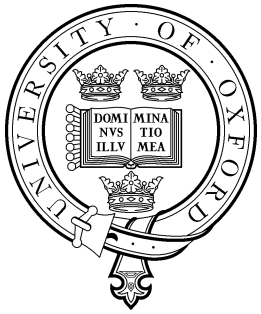
Model-Driven Data Migration



Mohammed A Aboulsamh

St Catherine's College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Hilary 2012

Abstract

Information systems often hold data of considerable value. Their continuing development or maintenance will often necessitate *evolution* of the system and *migration* of the data from one version to the next: a process that may be expensive, time-consuming, and prone to error. That such a process remains a source of challenges, is recognised by both academia and industry. In current practice, data migration is often considered only in the later stages of development, leaving critical data to be transformed and loaded by hand-written scripts, long after the design process has been completed.

The advent of model-driven engineering offers an opportunity to consider the question of information system evolution and data migration earlier in the development process. A precise account of the proposed changes to an existing system model can be used to predict the consequences for existing data, and to generate the necessary data migration implementation.

This dissertation shows how automatic data migration can be achieved by extending the definition of a data modeling language to include model-level operations, each of which corresponds to the addition, modification, or deletion of a model component. Using the Unified Modeling Language (UML) notation as an example, we show how the specification of these operations may be translated into an abstract program in the Abstract Machine Notation (AMN), employed in the B-method, and then formally checked for consistency and applicability prior to translation into a concrete programming notation, such as Structured Query Language (SQL).

Contents

Chapter 1

Introduction

1.1 Problem and motivation

Information systems are becoming more pervasive, and the data that they collect is becoming more detailed, more complex, and more essential to our daily lives. It should come as no surprise to find that this data will typically be more valuable than the components of the system itself. These components are easily replaced, easily upgraded: the cost of hardware and software continues to fall. The data, however, may be irreplaceable: the context in which it was collected no longer exists.

Measures can be taken to protect against the consequences of catastrophic data loss: should a hard disk fail, another may hold a copy of the same data; should a server fail, or a smartphone be lost, the data might be restored from a recent backup. More insidious, more damaging, and more likely is a loss not of data, per se, but of integrity. If this happens, any further data collected might be improperly or inconsistently stored; any further work done using the system may be wasted. If an integrity issue is not detected quickly, then considerable value may be lost.

The integrity of data—the *semantic integrity*—is expressed partly in terms of the values of different data items, and partly in terms of the relationships between them. For example: if a data item represents a person's age, then we would expect it to be in the range 0 to 120, perhaps—certainly not -53, or "yellow"; if one data item represents the number of children a person has, and another represents the number of male children, we would not only expect both items to be numeric, we would expect one to be numerically greater than the other.

The question of integrity becomes particularly important when the system itself is changed. If information is to be recorded in a different way, then both data values and relationships may need to be updated to match: existing data may need to be transformed, and new values generated, in order that the system might continue to

behave in a consistent fashion. This process of *data migration* can be costly, error-prone, and time-consuming. Several "dry runs" may be required, and even then there may be significant interruptions in service while data is transferred from the old to the new system, and checks are performed to confirm—as best as can be determined—that semantic integrity has been maintained.

The model-driven approach to systems development offers significant promise of improvement in this situation. If the design of the system, and the data representation in particular, is characterized by an abstract model, then we may be able to determine the data migration requirements by comparing two models: one of the existing version of the system, another of the new version that is to be installed. More than this, we may be able to adopt a model-driven approach to the data migration process: by constructing an *evolution model* that describes how the new model is obtained from the old, we might obtain a basis for the automatic generation of an appropriate data migration program.

This is the thesis of the present dissertation: that such a model can be constructed, written in a language that can itself be automatically derived from the language of the system model; and that an appropriate program can be generated. The program is appropriate not only in that it accurately reflects the proposed changes to the system, but also that we can guarantee, in advance, that the transformed data will meet the integrity constraints of the new system. This proposition is explored in the context of the modeling framework of MOF (Meta Object Factory), that underpins the widely-used UML (Unified Modeling Language) notation: a realistic context in which to examine the challenges of data migration.

The approach to the generation of an appropriate program—in particular, the determination of a sufficient condition for the program to guarantee the integrity of the transformed data—is based upon the formal, mathematical foundation of the AMN (Abstract Machine Notation), and the supporting framework of the B-method. The applicability of the approach is demonstrated by means of a transformation from the evolution modeling language into the widely-used Structured Query Language (SQL), perhaps the most common implementation platform for data migration.

1.2 Trends and challenges

In data-intensive systems such as information systems, the complexity of queries and updates to be specified may be low, but the data they act upon are typically very large, long-lived, sharable by many programs, and subject to numerous rules of

consistency [43]. While many years of research on database schema evolution provide solid theoretical foundations and interesting methodological approaches, information systems evolution activities are still a source of challenges [77]. As observed by [136], one important drawback of most schema evolution approaches is the lack of abstraction which remains largely unsolved. As a result, typically data conversion tasks in information systems evolution projects are left to programmers [41, 23]. Basic questions such as how to reflect in the existential level (i.e.data level) the changes that have occurred in the conceptual schema of a database, in such a way that consistency and validity of data are guaranteed, still represent a common problem in most information systems evolution settings [77].

Most often designers are basically "left alone" in the process of evolving an information system. Based only on their expertise, those designers must figure out how to express System changes and the corresponding data migration —not a trivial matter even for simple evolution steps. Given the available tools, the process is not incremental and there is no system support to check and guarantee consistency preservation, nor is support provided to predict or test consequences of evolution on existing data.

Model-driven engineering paradigm can be an important technique to reduce the complexity of information systems evolution : by employing abstraction and considering the various system artifacts as models, our ability to deal with the complexity of information systems evolution activity can be significantly improved. The abstraction promise offered by model-driven engineering paradigm has recently been realized in concrete results in the use of models as central elements in various software development and maintainance activities. For example, models can be used to capture design requirements and properties [18]; to map one software artifact to the other as in model transformation [45], to trace changes of a software artifacts as in model weaving [57], to reorganize and improve the quality of software as in model refactoring [166] or they can be used to generate platform-specific code and implementations [112]. These and other approaches in model-driven engineering may help in characterizing information systems evolution problem, however they remain largely general-purpose and offer no specific support for information systems evolution specific tasks such as data migration.

If models are to be treated as programs and compiled, or used as the basis for automatic generation of any kind, as advocated by model-driven engineering paradigm, then the abstract models must have a precise, formal semantics [46]. Assigning precise meaning to software models can be done by giving formal description to the abstract concepts and relationships that they present. This formalization helps in analyzing

and reasoning about the domain that the model represents. This way, formality can contribute to software development productivity and efficiency by bringing analysis and verification activities into the design phase rather than relying on implementation testing.

1.3 This dissertation

In this dissertation, we show how information systems evolution challenges can be addressed through a formal, model-driven approach. We start by showing how a model evolution language (in the form of a metamodel) may be derived for any Meta-Object Facility (MOF)-based data modeling language such as UML. We show how the evolution modeling language is adequate for the description of changes to models conformant to the selected data modeling language. We show also that the language can be given a formal semantics using the Abstract Machine Notation (AMN) of B-method, sufficient for the expression of consistency conditions of the language and data integrity constraints of the data model.

Our proposed language of model evolution, properly mapped to a formal semantics in B-method, can be used for the analysis of proposed evolutionary changes to data models. Using the supporting framework of B-method, we show how the applicability of a sequence of model evolution steps may be determined in advance, and used to check that a proposed evolution will preserve model consistency and data integrity. This applicability information can be fed back to system designers during the modeling activity, allowing them to see in advance how the changes they are proposing would affect the data in the existing system.

Using B-method refinement mechanism, we show how operating successive transformations on our abstract evolution specifications can be translated into an application in an executable language such as Structured Query Language (SQL), which is the implementation language we have chosen for describing data migration implementations. As a product of formal refinement, the generated code is guaranteed to preserve various consistency and integrity notions represented by the data model.

Each of our models or representations—the sequence of proposed changes, the AMN representation and the SQL implementation—conforms to a specific metamodel, each of which conforms in turn to MOF meta-metamodeling standard. Where a data modeling language, within MOF framework, is subject for evolutionary changes, we show how our approach may be generalized to account for data modeling language

evolution and predict consequences of such evolution on existing data models and corresponding data instances.

This dissertation is structured as follows. In Chapter 2, we provide a brief background on the main definitions, concepts and terminology we use in the rest of the dissertation.

Chapter 3 discusses requirements and motivations for a data model evolution language. The main idea of this chapter is to establish the need for a model evolution language applicable to data models written in a standard modeling notation and investigate the main features which such a language should present. Following a critical review of relevant literature, the chapter concludes with the identification of key requirements for a data model evolution language.

Chapter 4 presents a characterization of a subset of UML adequate for data modeling. This characterized subset is then extended with evolution operations so that we can specify evolution of data models written in UML. In particular, we show how, given a standard MOF-based data modeling notation (e.g. UML), the basic elements of an evolution language can be automatically generated. We then elaborate on the main components of the generated evolution language and discuss how it can be used as a basis for generating data migration implementation.

In Chapter 5, we use B-method to extend our proposed data model evolution language with appropriate relational semantics, mapping well-formed UML data models to structured constraints upon object properties and their values and mapping model evolution operations to appropriate substitutions upon object property values. We demonstrate how the proposed formal notation is sufficient for specifying data model consistency, an important pre-requisite for determining domain of applicability of model evolution operations. Using a proposed characterization of consistency constraints, we also show how conformance can be checked at two levels of MOF abstraction hierarchy — at model-level (between the data model and its modeling language) and at data-level (between data instances and the data model used for their collection and persistence).

Chapter 6 shows how the refinement of data model abstract evolution specifications into a formal description of the Structured Query Language (SQL), supports the generation of data transformations between successive versions of a system. In particular, we present a formal characterization of an SQL-metamodel and and demonstrate how the generated data migration (in SQL) are consistency-preserving

This dissertation ends with a discussion chapter summarizing research contributions, comparing these to related work and highlighting limitations and opportunities

for future work.

In the appendices of this dissertation, we describe the main components of a prototype implementation of the model-driven approach to data migration proposed by this thesis. We also include a complete description of B-method abstract, refinement and implementation machines we have developed to support the formalization of the presented approach. Finally, main proof activities are summarized at the end of this document.

Chapter 2

Background

Model-Driven Engineering (MDE) aims to raise the level of abstraction by which software systems are developed. This is done by emphasizing the need for thorough modeling. MDE relies on the use of models to describe software development activities and artifacts [18]. In MDE, models are abstractions of the artifacts developed during the software engineering process and can be used by domain experts to represent a variety of problem domains (e.g. data models, business processes, etc.). The basic assumption in MDE is that models are considered first-class entities: models are software artifacts that can be updated, analyzed or processed for different purposes [18].

A software model description can be formalized by giving precise interpretations to the classifications, associations, and constraints that the model presents. These precise interpretations lead to the formal analysis and verification of MDE. For example, using formal metamodeling we can ensure that a model defined with the intention of being conformant to a specific metamodel is, indeed, conformant, check that the various elements of a model are consistent, or guarantee correctness of a generated implementation.

Information systems frequently change throughout their lifetime. This change may occur due to various reasons such as changes in end user requirements or to improve the quality and organization of the system. These changes can be accommodated by changing the behavior of the system: the way the methods were implemented or by modifying the structure: the data model that was used to collect and persist the data.

The central theme of this thesis revolves around the problem of *information systems evolution*: how can we specify changes of an evolving information system and use these specifications to reason about evolution and data stored against an old sys-

tem model so that all structural and integrity constraints imposed by a new model remain valid?

We start this chapter by considering alternative data modeling notations. Using a running example, we explore the strengths and weaknesses of each alternative before we motivate our selection of UML. We then discuss some MDE fundamental concepts based on which we will build various elements of our proposed approach. Such concepts include metamodeling hierarchy and model transformation. In the second part of this chapter, we discuss the importance of integrating data modeling languages (UML in our context) with a precise formal notation. We discuss the applicability of a number of model-oriented notations, before we motivate our selection of B-method.

2.1 Data modeling approaches

When designing an information system for a particular application, we may create a design model of the application area. Technically, the application area being modeled is called the 'application domain' and typically is *part* of the *real world* [75].

In the field of information systems, we make the fundamental assumption that an application domain consists of a number of *objects* and the *relationships* between them, which are classified into *concepts*. The state of a particular domain, at a given time, therefore consists of a set of objects, a set of relationships, and a set of concepts into which these objects and relationships are classified [126]. For example, in the domain of a company, we may have the *concepts* of a customer, a product and a sale. At a given moment, we have *objects* classified as customers, *objects* classified as products, and *relationships* between customers and products classified as sales.

Building a good model for the application domain requires a good understanding of the world we are modeling, and hence is a task ideally suited to people rather than machines. Accordingly, the model should first be expressed at the *conceptual* level, in concepts that people find easy to work with. Implementation concerns are, of course, important, but should be ignored in the early stages of modeling. Once an initial conceptual design has been constructed, it can be mapped down to a logical design in any modeling language we like. This added flexibility also gives us a separation of concerns and makes it easier to implement and maintain the same application on more than one kind of implementation platform.

Although most information systems we conceptually model involve processes as well as data, given the context of our research, we will focus on the information (and hence on the data). This focus on date will, in turn, require us to concentrate on *data*

models, defined as collections of concepts that can be used to describe the structure of databases, i.e. data types, relationships and the constraints that should hold for the data [117].

There is a great diversity in conceptual data models, and they may be more or less useful in particular situations or for particular purposes. However, all of them are based on the fundamental assumption that we have mentioned above, which we shall attempt to clarify in the remainder of this section.

Within the context of this research, based on the work of [77, 74, 153, 19, 20], we have considered three main conceptual modeling approaches: Entity-Relationship(ER) modeling, Fact-Oriented modeling and Object-Oriented modeling. In our discussion of these modeling approaches, we do not intend to describe them in their entirety. We will focus our discussion on the main principles of each approach to confirm the fundamental assumption we stated at the beginning of this section and present the modeling approach which we will adopt throughout this thesis.

Running example. In this chapter and throughout the remaining chapters of this thesis, we will use a simplified Employee Information System (EIS) data model as a running example to illustrate our ideas and explain alternative approaches. In our EIS example, a company workforce is represented by employees. The company stores each employee's name and age. The company is organized into departments and each employee works for a department which is managed by an employee of the company. In addition, employees' personal data maintained by the company is organized in personal files which are linked to corresponding employees. In the remaining parts of this section, we will use the above example to illustrate the concepts of alternative modeling approaches which we have considered.

2.1.1 Entity-Relationship (ER) modeling

Entity-Relationship (ER) was introduced by Peter Chen in 1976 [36] and is one of the most widely used approach for data modeling. It pictures the world in terms of *entities* that have *attributes* and participate in *relationships*. Over time, many different versions of ER emerged, and today, there is no standard ER notation [153].

The basic object that the ER model represents is an entity. An entity may be an object with a physical or a conceptual existence. Each entity has attributes—the particular properties that describe it. A particular entity will have a value for each of its attributes. Each entity in the database is described by its name and attributes.

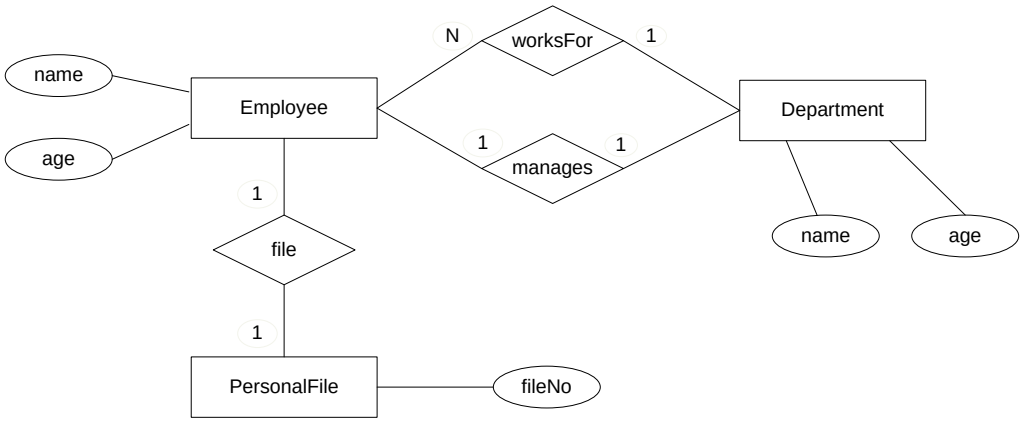


Figure 2.1: An ER Schema diagram for a simplified Employee Information System

Example. Figure 2.1 depicts our EIS running example using Chen's ER notation. As the diagram shows, an entity is represented in ER notation, as a rectangular box enclosing the entity name. Attribute names are enclosed in ovals and are attached to their entity by straight lines. An important constraint on an entity is the key or uniqueness constraint on attributes. Such an attribute is called a key attribute, and its values can be used to identify each entity. Whenever an attribute of one entity refers to another entity, some relationship exists. In the ER model, these references should not be represented as attributes but as relationships. In ER diagrams, relationships are displayed as diamond-shaped boxes, which are connected by straight lines to the rectangular boxes representing the participating entities. The relationship name is displayed in the diamond-shaped box. Relationships can also have attributes, similar to those of entities. The cardinality ratio for a relationship specifies the maximum number of relationship instances that an entity can participate in. For example, the cardinality ratio of 'n and 1' between Employee entity and Department entity is interpreted as 'many to one' (each employee works for at most one department, but many employees may work for the same department).

To its credit, this ER diagram portrays the application domain in a way that is independent of the target software platform. For example, classifying a relationship end as mandatory is a conceptual issue. There is no attempt to specify here how this constraint is implemented (e.g. using mandatory columns, foreign key references, or object references). Conversely, the ER diagram is less than ideal for validating the model with the domain expert, and the conceptual step from the data output to the model can, sometimes, be ambiguous and may only be guessed by the model's creator. For example, in ER diagrams the direction of relationships is undecided (i.e.

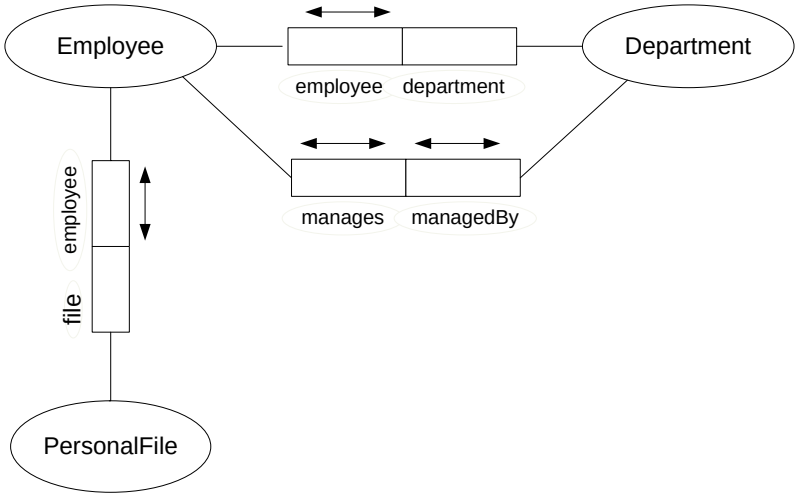


Figure 2.2: An ORM model diagram for a simplified Employee Information System

the diagram does not explicitly state whether the employee works for the department, or the department works for the employee). Outside academia, Chen's notation seems to be rarely used nowadays. One important reason may be that there are so many ER notation versions, with no single standard.

2.1.2 Fact-Oriented modeling

Object Role Modeling (ORM) is one of the most commonly cited Fact-Oriented modeling method [27]. It began in the early 1970s as a semantic modeling approach that views the world simply in terms of objects playing roles (taking parts in relationships). ORM has appeared in a variety of forms such as the natural-language information analysis method (NIAM) [74]. ORM includes various procedures to assist in the creation and transformation of data models. A key step in its design procedure is the verbalization of information examples relevant to the application, such as sample reports expected from the system. ORM sentence types (and constraints) may be specified either textually or graphically.

Domain concepts are shown in ORM as named ellipses and must have a reference scheme, as an abbreviation of the relevant association (e.g., Employee has a name). In ORM, a role is a part played in a relationship or association. A relationship is shown as a named sequence of one or more role boxes, each connected to the object type that plays it.

Apart from object types, the only data structure in ORM is the relationship type. In particular, attributes are not used at all in base ORM. This is one of the

fundamental differences between ORM and and ER. Wherever an attribute is used in ER, ORM uses a relationship instead. An ORM model is essentially a connected network of object types and relationship types.

Example. To understand how ORM works, we use the same running example of EIS depicted earlier in ER. In an ORM diagram, as shown in Figure 2.2, roles appear as boxes, connected by a line to their object type. Roles are simply the line ends, but may optionally be given names. ORM allows associations to be objectified as first class object type. In ORM this is captured as a mandatory role constraint, represented graphically by a black dot. The lack of a mandatory role constraint on the left role indicates it is optional.

2.1.3 Object-Oriented modeling

Although many object-oriented approaches exist, the most influential is the Unified Modeling Language (UML), which has been adopted by the Object Management Group(OMG) [124].

UML is a general-purpose modeling language. It offers elements of a graphical concrete syntax to create visual models of software intensive systems. UML synthesized notations of the Booch method [22], the Object-Modeling Technique (OMT [140]) and Object-Oriented Software Engineering (OOSE [86]) by fusing them into a single, common and widely usable modeling language; several other methods have also influenced the UML, as for instance Entity-Relationship modeling.

The UML rapidly gained popularity in industry, as it facilitated the communication between diverse stakeholders at different phases of software development, on one hand, and, on the other, it provided several points of views onto that system, including its context.

UML takes up central ideas from the ER model and put them into a broad software development context by proposing various graphical sublanguages and diagrams for specialized software development tasks. UML concepts relevant for data modeling include concepts typically used for modeling structural aspects such as objects, classes, properties, generalization and association. More precision to UML diagrams may be added by using textual constraints of Object Constraint Language [121].

Example. Figure 2.3 shows how our EIS data model can be represented in a UML class diagram. Each concept in the model is represented by a separate class. As such, we have a class for Employee, Department and Personal File. Each class

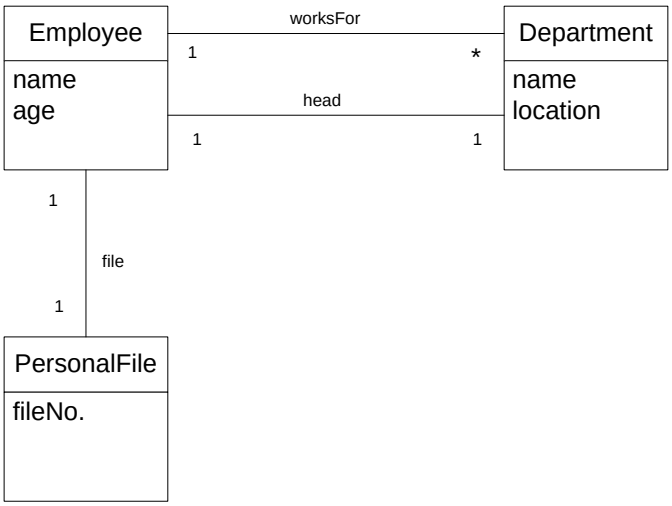


Figure 2.3: A UML model for a simplified Employee Information System

includes a number of attributes to help define properties of objects of each respective class. For example, for each employee object we are interested in capture name and age attributes. In addition, the diagram shows a number of associations describing relationships between different classes. For example, head is a relationship between Employee and Department classes. Based on such relationship, we expect each department object to be associated with an employee object acting as a head of that department.

2.1.4 Choosing an appropriate data modeling approach

It is important to note that, when describing data models, although terms used by each modeling approach vary, the meaning of concepts remains very close. For example, the concept of *entity* proposed by Chen [36] to describe object with a physical or a conceptual existence is similar to the concept of *object type* used by ORM and the concept of *Class* used by UML. Similarly, the concept of *relationship* proposed by Chen to make the connection between entities and the cardinality constraints that may accompany such description. This concept can be intuitively mapped to the concept of *association* in UML.

It is equally important to note that these modeling methods differ in representing some of the application domain concepts. As we highlighted above, unlike UML, ER models do not explicitly show the direction of entity relationships. In addition, unlike ER and UML, ORM makes no use of attributes in its models. All facts are represented in terms of objects (entities or values) playing roles. As such, compared to

ER and UML models, ORM models loose an important advantage: ability to produce compact models, which can complicate the task of the modeler.

Considering the relative strengths and weaknesses of different approaches outlined above, in this thesis we have selected to use UML for data modeling. Our decision is based on three main considerations:

1. *Extendability.* As we will show in Chapter 4, a main component of our model-driven approach is an evolution model. This evolution model represent abstract description of data model changes. To capture such data model evolutionary changes, we need to extend the modeling language in which the data model itself is written. This requires the modeling notation to be extendable. Extension to the UML may be achieved in two different ways [87], as a *lightweight* or *heavyweight* extension. The first of these involves the definition of a UML 'profile', using extension mechanisms included within the standard UML profiles package ([124], pp.179). This is the approach we followed in [2], but soon discovered that it is limited in its applicability. Instead, here we present the *heavyweight* extension, similar to the approach we followed in [3]. Unlike UML, other modeling approaches such as ER and ORM do not offer similar extendability mechanisms, limiting our ability to model the data model and its abstract evolutionary changes in the same modeling notation. This topic is discussed further in Chapter 4.

2. *Standardization.* Unlike UML, none of the other modeling approaches which we have considered has been adopted as a standard modeling notation. The use of a standard modeling framework such as UML for data and evolution modeling makes our approach relevant for potential usage by a wide range of audience in software engineering community in both academia and industry.

3. *Tool support.* One aim of this thesis is to realize a solution for data model evolution. Such a solution can be used by information system designers in performing data model updates, analyzing some of the consequences of their updates and subsequently migrating the data. As we will explain in Chapter 5, the realization of such solution requires integrating a number of metamodels in a model-driven engineering chain. Although tool support exists for ER and, at a wider extent, for ORM, tool support in the model-engineering community for UML exceed by far any other modeling approach. This abundance in tool support increases our opportunity to integrate with a large number of existing supporting tools towards making our approach more usable.

2.2 MDE of information systems

Model-Driven Engineering (MDE) is a unification of initiatives that aims to improve software development by employing high-level domain models in the design, implementation, testing and maintenance of software systems. The basic assumption in MDE is to consider models as first-class entities. The main implication of this assumption is that models are software artifacts that can be modified, updated, or processed for different purposes. Different operations can be applied on models. This differs from the traditional view of software development where models are used essentially for documentation.

MDE of information systems applies the principles of MDE in the development of information systems: an information system is specified as a set of models that are repetitively transformed and refined until a working implementation is obtained.

Within the domain of information systems, the notion of MDE has been a subject of increasing attention. Models have been widely used for the design and development of databases [47], schema and data integration [96], [132] and for data transformation [67]. In this work we use MDE paradigm to cater for the evolution and maintenance of information systems.

In MDE models are placed at the center of software development. A *model* represents a problem domain such as software, business processes or user requirements. The different entities of the problem domain are captured by model elements. The model elements have a set of properties, and may have relations between themselves. The nature of the model elements, i.e. their type, set of properties, and possible relations, are defined in a *metamodel* to which the model needs to conform. The primary responsibility of the metamodel layer is to define a language for specifying models. As such, a metamodel may be considered as the type of a given model, because it defines a set of constraints for creating the model. UML [120] and the OMG Common Warehouse Metamodel (CWM) [154] are examples of metamodels. A metamodel, in turn conforms to a *meta-metamodel*. The primary responsibility of this layer is to define the language for specifying a metamodel. The meta-metamodeling layer forms the foundation of the metamodeling hierarchy, established by OMG [124]. This layer is often referred to as M3, and MOF is an example of a meta-metamodel. The metamodel, the model and user data object layers are referred to as M2, M1 and M0 respectively.

As we described in the previous section, although different modeling notations can be used in MDE of information systems such as Entity-Relationship (ER) diagrams

[153], Object Role Modeling (ORM) [75], we focus on using the Unified Modeling Language (UML) [120], which is a representative notation of object modeling paradigm. In information systems design, UML models are normally augmented with OCL constraints [121] to express data integrity properties, pre and postconditions of update and query operations [51].

From an MDE perspective and considering the four-layer metamodeling architecture outlined in Section 2.2.1, UML and OCL element used in an IS data model correspond to layer [M1] and must conform to metamodels of UML and OCL at layer [M2]. Data models, in turn, can be instantiated into object diagrams representing data instances of real life systems [M0].

2.2.1 Metamodel hierarchy

An illustration of how these meta-layers relate to each other is shown in Figure 2.4 which is adapted from [124]. A metamodel (at level M2) is an instance of a meta-metamodel (at level M3), meaning that every element of the metamodel is an instance of an element in the meta-metamodel. Similarly, a model (at level M1) is an instance of a metamodel and a user run-time object (at level M0) is an instance of a model element defined in a model defined at the upper abstraction layer.

The relationship between meta-metamodel, metamodels and models can be confusing. A major source of that confusion is the fact that the UML meta-metamodel (MOF) is subset of UML itself (corresponding roughly to class diagrams). To help clarify this confusion, we may use object-based representation technique as outlined below.

2.2.2 Class-based and object-based representation of metamodels and models

As already stated, a *model* that is instantiated from a metamodel can, in turn, be used as a *metamodel* of another model in a recursive manner. As such, A MOF metamodel (at level M2) can either be represented in a *class-based* : by providing type descriptions via a set of model elements, as shown above in Figure 2.4, or in *object-based* representation: by a set of MOF objects that instantiate the MOF model elements, as shown in Figure 2.5. This *dual* representation is often left implicit when using the MOF with users generally employing the class-based representation [133].

To illustrate this concept, in Figure 2.5, we show an object-based representation of the same metamodel hierarchy shown in Figure 2.4. The metamodel in level M2

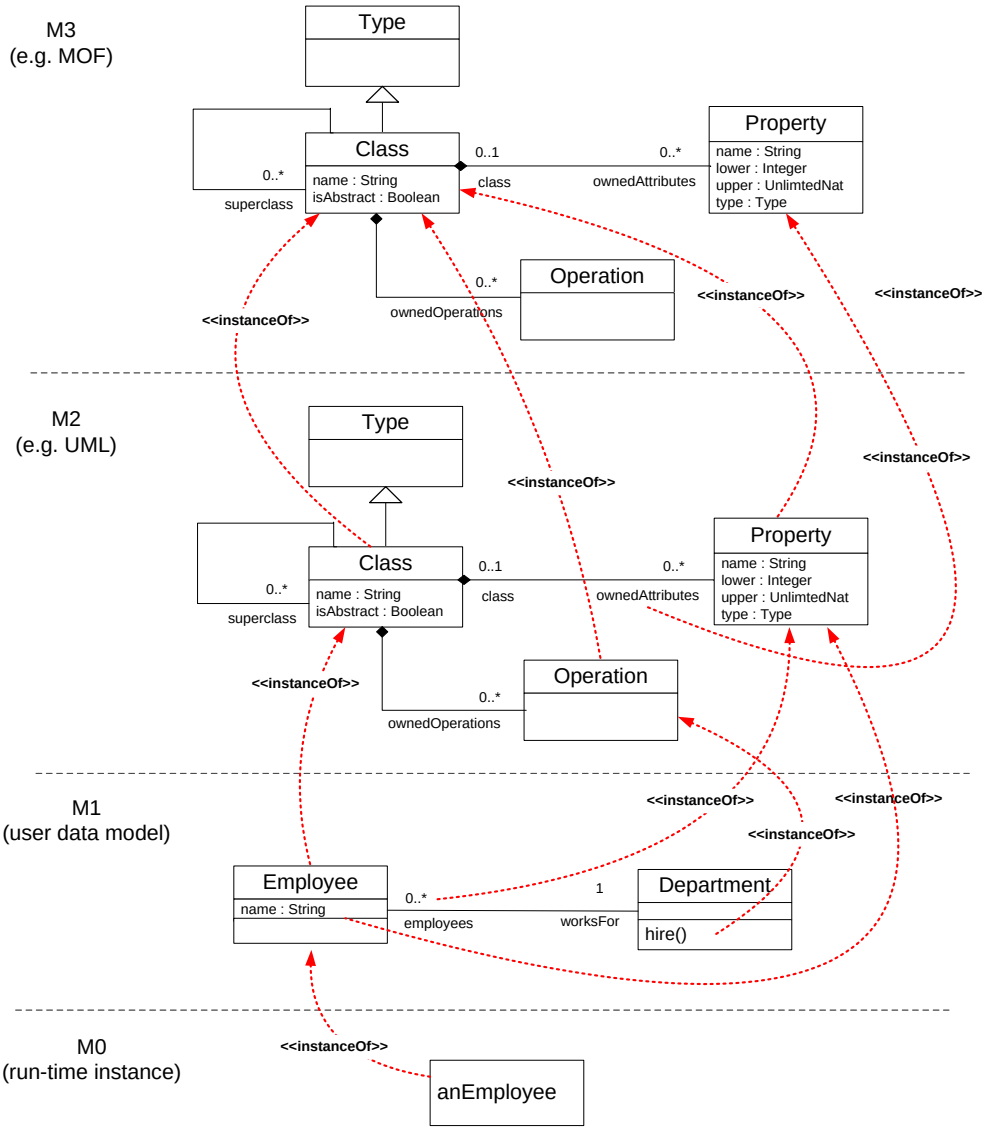


Figure 2.4: Illustration of OMG metamodel hierarchy

is defined as a collection of MOF class objects. Objects `C1`, `C2` and `C3` are instances of MOF Class concept in level M1 and are used to denote Class, Property and Operation concepts of UML metamodel. Objects `P1`, `P2` and `P3` are instances of MOF Property and are used to denote *name*, *isAbstract* and *ownedOperations* properties of UML Class concept. Similarly, at level M1, we show part of the user data model in an object-based representation. Objects `C1` and `C2` are used denote Employee and Department classes and objects `P1` and `P2` are used to denote the two associations in the model: *worksFor* and *employees*. The bottom layer of the diagram shows an instance of the Employee class.

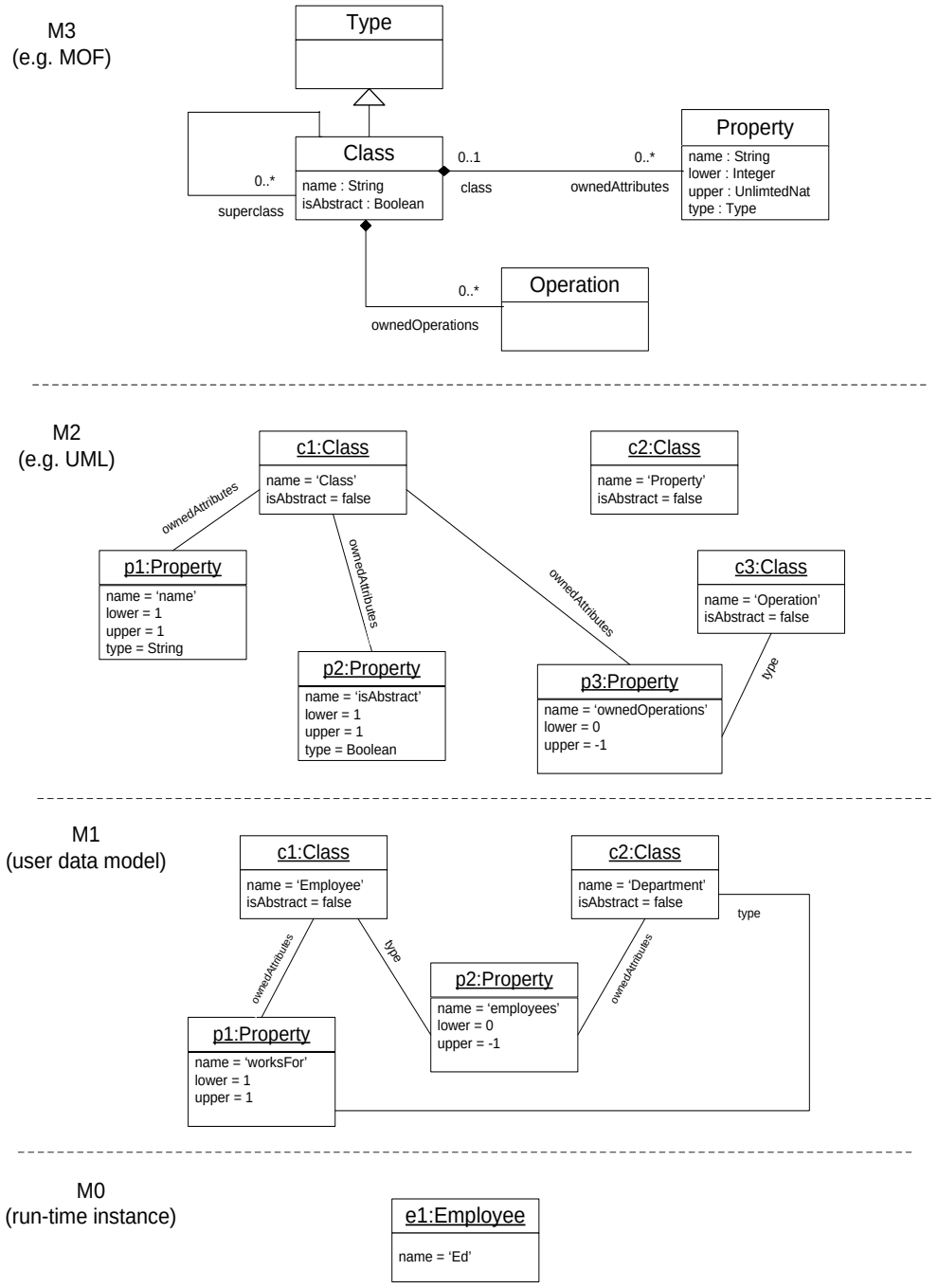


Figure 2.5: Object-based representation of metamodel hierarchy, shown in Figure 2.4

2.2.3 Model-Driven Architecture (MDA)

An attempt to implement a development process based on the concepts of metamodel hierarchy outlined above has been proposed and standardized by OMG under the name of Model-Driven Architecture (MDA) [125]. Such an approach is based on

three fundamental abstraction layers, each of which related to specific artifacts. In particular, the highest abstraction level deals with the description of business logic details through a Platform-Independent Model (PIM) and a Computation-Independent Model (CIM). It is not important to know further information about the implementation platform. Such description is delegated to the lower level of abstraction described as a Platform-Specific Model (PSM).

2.2.4 Model transformation

Model transformation is at the heart of MDE [142]. Model transformation techniques can enable a wide range of different automated activities as domain models can be transformed into other models at the same level of abstraction *(horizontal model transformation)* or at a lower level of abstraction *(vertical model transformation)* [45]. Examples of horizontal model transformation include model refactoring and model weaving while model refinement and code generation are considered examples of vertical model transformation.

Model transformations define a mapping between a source and a target model through a set of rules. Each rule links one or more model elements of the input with the corresponding target. Obviously, this description has to be given in a generic way; therefore, it can not be based on a particular model instance but on the abstract specification of such instances, that is, on the *metamodels*.

Model transformations can vary from simple element correspondences (translations) to intricate mappings which require a deep knowledge of source and target domains and complex navigations and analysis of the given abstractions. As a consequence, increasing development of model transformations demands a growing precision of model specifications. As we will be explaining in Section 2.3, this precision may be achieved by augmenting MDE techniques by formal semantics.

A considerable amount of research has been produced in the area of model transformation. Some of this research has developed into integrated approaches addressing different aspects of model transformation technique and accompanied by tool support. Some of such research efforts include for instance, ATL [90], GReAT [7], C-SAW [166] and VIATRA [44]. Our early investigation of information systems evolution used ATL [90] tool suit to build a prototypical solution, please see [1].

2.2.5 Information systems evolution

Information systems evolution can be driven by changes in various dimensions: *end user requirements* (e.g. integrating new concepts and new business rules often translates into the introduction of new data structures and behavior) and *technological advancements* (e.g. enhanced capabilities of DBMS with the increasing requirements for real time answers of unplanned queries may induce modifications to an information system existing data structures and functions) [73]. Another dimension of changes involves fixing error conditions (e.g. due to misunderstanding of initial requirements).

For the purpose of this research, the information systems evolution scenario we are interested in here may be characterized by the continuous introduction of small and incremental changes and refinements to the underlying object models employed in the development of information systems. These changes are normally induced by new user requirements and specified after initial system deployment. As a result, we may expect to see a series of object model versions representing a natural progression: or at least, one in which the differences between successive model versions can be captured and explained.

We may describe the information system evolution problem as a variant of *coupled transformation* problem outlined by [97] which occurs when multiple instances of software artifacts need to adapt to conform to an updated version of a shared specification so that they remain consistent with each other. This problem can be tracked in several areas of computer science such as database schema evolution, grammar evolution and format evolution.

Obviously, among these manifestations of the coupled transformation problem, database schema evolution is more closely related to the work we present here. A detailed review of relevant database schema evolution approaches is included in the next chapter.

2.3 Formal foundations for model-driven engineering

As highlighted in Section 2.2, the use of MDE paradigm in the design and development of information systems has attracted an increasing attention in both academia and industry. Using abstract models to describe various system concerns, to facilitate communication among stakeholders and to generate working information system

implementations promise not only to increase system quality but also to reduce system development time and cost. However, the lack of precise semantics of various modeling notations employed by MDE continued to cause difficulties. An informal description unavoidably involves ambiguities and lacks rigor, and thus precludes the reasoning, early simulation and automated analysis of system design expressed by a model. For example, it would be impossible to prove the consistency of a non-trivial system purely described in a data model. Equally, within the context of MDE model transformation process (outlined in section 2.2.4), it would be extremely difficult to ensure that a target (generated) software model is valid. We need to define validation properties to ensure the syntactic and semantic correctness of the generated model are in accordance with the specified metamodel.

These problems have been widely recognized [64, 17, 24, 38], and have led to the development of a number of approaches to improving the precision of modeling notations. The most common approach to the problem has been to make modeling notations more precise and amenable to rigorous analysis by integrating them with a suitable formal specification notation [24]. A number of integrated UML and formal notations have been proposed, see [29] for a recent survey.

Formal specification methods can provide a precise supplement to modeling language description and can be rigorously validated and verified, leading to early detection of error conditions. A method is *formal* if it has a sound mathematical basis, typically given by a formal specification language. This basis provides the means of precisely defining notions like consistency, completeness and correctness. It provides the means of proving that a specification is realizable, proving that a system can be implemented correctly, and proving properties of a system without necessarily running it to determine its behavior.

2.3.1 Integrating modeling approaches and formal methods

An important aspect of this thesis is to investigate the formalization of UML data models within the context of information systems evolution and data migration. We aim to show how the assignment of precise meaning to UML data models can be achieved by mapping a core part of UML modeling language constructs to appropriate mathematical constructs in an appropriate formal notation. This mapping is a formal equivalent of the informal semantics of UML modeling language itself: an explanation of what each construct in our selected modeling language means. Once our UML models have formal semantics, we are able to exploit formal reasoning techniques associated with the formal method that we have employed. Since UML is a large

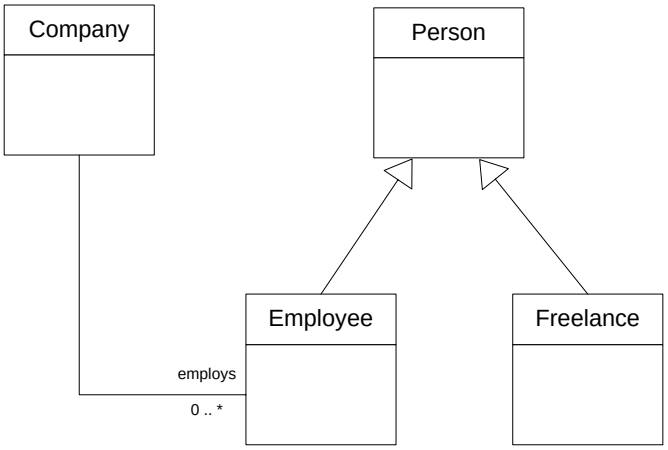


Figure 2.6: A simplified company example

modeling language, we restrict ourselves to considering formalization techniques for a core part of its static model.

Example. Figure 2.6 shows an example inspired by [17]. A UML data model can provide a visually expressive and intuitive description of an information system. However, it is less effective when it comes to answering important questions about the system it represents. In particular, it is not possible to reason (in a precise manner) with the model or deduce properties about it. For example, what is the relationship between the Company and the Person?. Furthermore, does a company employ any of its Freelance staff?. Based on informal arguments, these questions might be answered like: 'some persons must be employed by the company, as some employees are persons', or 'clearly, there is no relationship between the company and freelance staff - they are not connected', or 'surely, some freelance staff can be employed - the data model does not forbid this'. By developing a precise description of what a UML data model means, we can develop sound rules for reasoning with UML models.

Accordingly, exploring the semantic base of UML with formal techniques can be beneficial for a number of reasons [31]:

1. Formalization allows one to explore consequences of a particular design. Such exploration can uncover problems related to incomplete, inconsistent and ambiguous specification.

2. Variants of the semantics can be obtained by relaxing and/or tightening constraints on semantic models. This paves the way for performing various analysis

and simulation techniques in different contexts and can yield significant insights.

3. System verification is the process of showing that a system satisfies its specification. Formal verification is impossible without a formal specification. Although we may never completely verify an entire system, we can certainly verify smaller, critical pieces.

Naturally, based on the above benefits, we should now ask the question: which formal method should we integrate with our core part of UML data modeling?. Given the diversity of formal method notations, there is no immediate answer to this question. Each formal method provides different concepts and is well-suited to a particular context of use. In the following section we take a closer look into our formalization requirements and map them into the characteristics of some of the well-known formal methods.

2.3.2 Choosing an appropriate formal method

To be able to identify an appropriate formal method to use, we need to establish the main characteristics of information systems data models that we need to capture and formalize. The main purpose of a data model is to represent structural concerns which involve identifying appropriate static concepts, their properties and relationships, as opposed to concepts representing other concerns such as real-time or distributed systems.

Accordingly, an appropriate formal method that we can choose needs to provide the capabilities to represent various concepts of static structures such as classes, properties and associations with a high level of abstraction. Such capabilities can typically exist when the method provides mechanisms well-suited for capturing modular specifications. In addition, given our focus on refining abstract specifications into an executable code, the formal method we need to choose should have a well-established refinement mechanism. Ideally, such a mechanism should be an integrated part of a method capable of representing the overall life-cycle of software system development.

Formal methods differ because their underlying specification languages have different *syntactic* (specific notations with which a specification is represented) and/or *semantic domains* (a 'universe of objects' that is used to describe the system). An important consideration prior to applying formal methods is that some languages may be more suitable to one type of specification than to others [163]. For example, a formal method might be applicable for describing sequential programs but not parallel ones, or for describing message-passing distributed systems but not transaction-based

distributed databases. Without knowing the proper domain of applicability, a user may inappropriately apply a formal method to an inapplicable domain.

From a high-level perspective, we can classify three broad classes of formal methods : *model-oriented*, *property-oriented* and *process-oriented* [163]. Using a *model-oriented* method, a user defines a system's behavior directly by constructing a model of the system in terms of mathematical structures such as tuples, relations, functions, sets, and sequences. Using a *property-oriented* method, a user defines the system's behavior indirectly by stating a set of properties, usually in the form of a set of axioms, that the system must satisfy. Examples of property-oriented languages include Larch [65] and OBJ [68]. Concurrent systems are described using *process-oriented* formal specification languages. An implicit model for concurrency is typically the basis for these languages. In these languages, processes are denoted and built up by elementary expressions which describe operations of simple processes that are combined to yield new potentially more complex processes. An example of this category of languages include Communicating Sequential Processes (CSP)[28].

The main criteria we outlined above allow us to eliminate *property-oriented* and *process-oriented methods*. The emphasis of these methods on specifying system properties and behavioral concerns can prevent us from building an explicit model of system structure. This leaves us with formal methods that fall within the third category of model-oriented languages. In the following section, we concentrate on presenting some well-known or commonly-used *model-oriented* formal methods.

Model-oriented languages aim to capture software systems structure and state abstractly and succinctly. They are based on the idea of data abstractions and most of them favor a style of specification based on abstract data types (ADTs). State-based languages that favor a model-oriented style of specification include the languages of VDM, Z, Object-Z and B. They all follow a similar approach to model software systems, based on the concepts of set theory and predicate logic.

VDM (Vienna Development Method) [89] started in the early 1970s. Initially, the language was intended to be used to model the semantics of programming languages, but it was later adapted to model software systems in general. The latest version of the language, VDM-SL, is documented in ISO standards [82]. VDM favors a style of specification based on abstract data types (ADTs), providing special syntax to support this style of specification (record types).

Z is based on typed set-theory and first order predicate calculus, enriched with a structuring mechanism based on *schemas*. Z is flexible and adaptable to suit different styles of specification, but the standard style is based on ADTs specified using

the schema calculus. The latest version of Z is in ISO standards [83]; the version documented in [147] is still popular.

Object-Z is an extension to Z to facilitate the structuring of a model in an Object-Oriented style. The reference language definition is [143]. It provides the class schema as a structuring mechanism (a collection of an internal state schema and operation schemas) and support for Object-Oriented notions such as object, inheritance, and polymorphism. Object-Z is the most successful OO extension to Z, among the ones that emerged in early 1990s (see [149] for a survey).

B [6] is a state-based model-oriented language and a method designed by Jean Raymond Abrial, one of the early contributors to Z. It includes a notation based on abstract machines, and a method to refine abstract models into code in a stepwise-manner. Formal verification of proof obligations ensures that a specification is consistent throughout its refinements. B, like its predecessor, Z [147], is based on set theory and first-order predicate logic. For refinement, B requires a 'refinement relation' as part of its invariant predicate, which is analogous to an 'abstraction relation' schema in Z. The reference version of the language is documented in [6].

Comparison — VDM, Z, Object-Z and B are state-based model-oriented languages, which usually model a system by representing its state as a collection of state variables, their values and some operations that can change system state. All are based on set theory and mathematical logic. We can eliminate VDM and Object-Z. According to [89] and [143], these two methods are better-suited for describing programs written in Object-Oriented, while our main focus is on describing database data models. Z notation is a strongly typed specification language. It is not an executable notation; it cannot be interpreted or compiled into a running program [85]. Z and B converge on their approach to model software systems, which reflects Abrial's influence on both of them, but they diverge on the level of abstraction. Z is more abstract, focusing on modeling software systems, and B is more like an abstract programming language with a very strong emphasis on refining models into code. Both languages also differ on their semantics, Z has a denotational semantics, and B has a semantics based on weakest-preconditions.

At an early stage of this research our initial conclusion was that either language would be applicable. However, upon further investigation and considering our intended development context, we favored B for three main reasons. First, B modular Abstract Machine Notation (AMN) and Generalized Substitution Language (GSL) mapped appropriately with our intended representation of data model and evolution

MACHINE M
SETS S
VARIABLES v
INVARIANT I
INITIALIZATION T
OPERATIONS
out<-op =
…
/* GSL Operators */
…
END

(a) Abstract Machine clauses

GSL Operator	Substitution
SKIP	Immediate termination
$\mathbf{S_1}$ **[]** $\mathbf{S_2}$	CHOICE $\mathbf{S_1}$ OR $\mathbf{S_2}$ END;
P | S	PRE **P** THEN **S** END**;**
P ==> S	SELECT **P** THEN **S** END
$\mathbf{S_1}$ **;** $\mathbf{S_2}$	do $\mathbf{S_1}$ then $\mathbf{S_2}$
$\mathbf{S_1}$ **||** $\mathbf{S_2}$	do $\mathbf{S_1}$ and $\mathbf{S_2}$

(b) partial list of GSL Operators

Figure 2.7: Main elements of B-method AMN and GSL notations

model respectively. The lower level of abstraction provided by these two notations ensured that representing the semantics of our data and evolution modeling concepts would not be cumbersome. Second, B refinement mechanism which covers both data and operation refinement can be used to generate executable code from specification. Finally, compared to Z, B is better-equipped with tool support. It is supported by tools for proof and refinement. B proof tools such as Atelier B [13] and B toolkit [102] are well-documented and can be used at various development steps to ensure consistency of both abstract specification and related refinement.

In this thesis, we therefore propose to use B-method to formalize UML data models. Below we highlight some of the previous work that used B-method to formalize UML models

2.4 Formal modeling with B

A B model is constructed from one or more abstract machines. Each abstract machine has a name and a set of clauses that define the structure and the operations of the machine. Figure 2.7(a) shows main clauses of B AMN where clause SETS contains definition of sets; VARIABLES defines the state of the system, which should conform to properties stated in the INVARIANT clause. INITIALIZATION of variables and variable manipulations in OPERATIONS clause should also preserve invariant properties. OPERATIONS are based on GSL whose semantics is defined by means of predicate transformers [71] and the weakest precondition [52].

A generalized substitution is an abstract mathematical programming construct, built up from basic substitution, for example, the Assignment operator takes the form

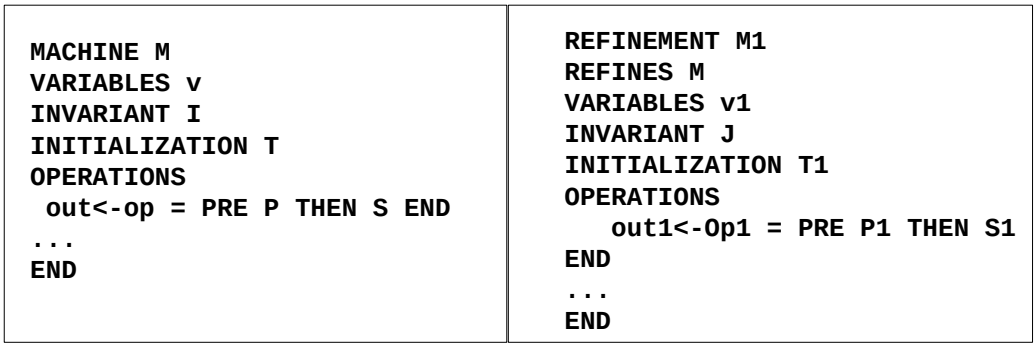


Figure 2.8: Example of Data Refinement in B

x := E, corresponding to assignment of expression E to state variable x. The Preconditioning operator P|S executes as S if the precondition P is true, otherwise, its behavior is non-deterministic and not even guaranteed to terminate. The statement @x.(P ==>S) represents an unbounded choice operator which chooses an arbitrary x that satisfies predicate P and then executes S with the value of x. Other GSL operators include SKIP, Bounded Choice, Guarding, Sequential and Parallel composition as can be seen in Figure 2.7. [6] provides more details on GSL and AMN.

B is a proof-based development method which integrates formal proof techniques in the development of software systems. At each step of the development process, B gives rise to a number of so-called *proof obligations* which guarantee consistency of specifications and correctness of subsequent implementation. Such proof obligations can be discharged by a B proof tool using automatic or interactive proof procedures supported by a proof engine, for example [102].

The B-method supports the notion of data refinement. Design decisions that are more concrete (closer to an executable code) are stated in *refinement* machines as opposed to *abstract* machines that include abstract design decisions. A *refinement* machine must include a *linking invariant* which relates abstract state to a refinement state. In addition, a refinement machine will have exactly the same interfaces as the machine it refines. This means that it will have the same operations as the abstract machine with exactly the same input and output parameters. Furthermore, operations in the refinement machine are required to work only within the preconditions given in the abstract machine, so those preconditions are assumed to hold for the refined operations [141].

Assume that the refinement machine M1 below is a refinement of the abstract machine M:

Machine M1 might then contain new variables as well as replace the abstract data structures of machine M with the concrete ones. The invariant of M1 - J defines not only the invariant properties of the refinement machine, but also the *linking invariant* connecting the state spaces of M and M1.

For a refinement step to be valid, every possible execution of the refined machine must correspond (via J) to some execution of the abstract machine. To demonstrate this, we should prove that initialization *T1* is a valid refinement of *T*, each operation of M1 is a valid refinement of its counterpart in M. In other words, we must discharge the following proof obligations to ensure that the transformation preserves the properties of the abstract level:

1. [T1] $\neg$ T $\neg$ J
   This proof obligation states that every initial state [T1] in the Refinement machine must have a corresponding initial state [T] in the Abstract Machine via the linking invariant J.

2. I $\wedge$ J $\wedge$ P $\Rightarrow$ [S1] $\neg$ [S] $\neg$ J
   This proof obligation states that every possible execution [S1] of the Refinement Machine must correspond (via the linking invariant J) to some execution [S] of the Abstract Machine. We require this to be true in any state that both the Abstract Machine and the Refinement Machine can jointly be in (as represented by the invariants I $\wedge$ J) we require this to be true only when the operation called within its precondition.

3. I $\wedge$ J $\wedge$ P $\Rightarrow$ [S1 [out1/out]] $\neg$ [S] $\neg$ ($J \wedge out1 = out$)
   this proof obligation has exactly the same explanation provided in proof obligation 2 above with the added condition that output `out1` of the Refinement operation, must also be matched by an output `out` of the Abstract Machine operation.

To carry out these proofs, the B-toolkit [102] includes two complementary provers. The first one is automatic, implementing a decision procedure that uses a set of deduction and rewriting rules. The second prover allows the users to enter into a dialogue with the automatic prover by defining their own deduction and/or rewriting rules that guide the prover to find the right way to discharge the proofs.

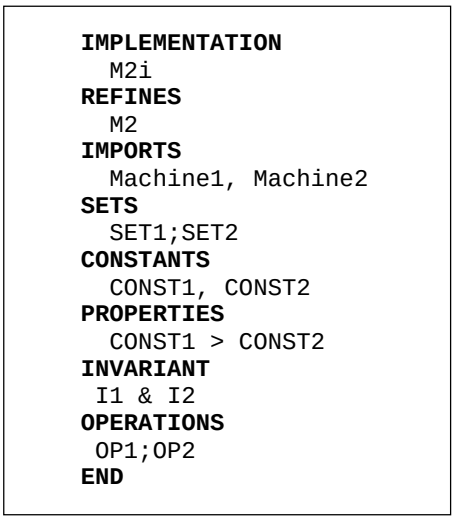


Figure 2.9: Main clauses of B Implementation Machine

An *implementation* is a particular type of a B machine. When a B-developed system is to be implemented in executable code, it is typically refined to an *implementation* which IMPORTS an abstract specification of an existing (coded) development to implement the operations of a refined component.

Figure 2.9 shows the main clauses of an Implementation machine. REFINES clause names the abstract machine or the refinement being implemented; IMPORTS names an abstract machine whose operations are used in the implementation; SETS defines local sets which must be fully enumerated; CONSTANTS defines single scalar values which are fully defined in the PROPERTIES clause; INVARIANT clause links the states of imported and refined machines. OPERATIONS lists operations from the refined machine which are implemented in terms of operations from the imported machine.

The proof obligations for an implementation are the same as those for refinement. In addition, given that an implementation machine is closer to concrete implementation than the machine it refines, it only allows a restricted number of substitution forms compared to those used in an abstract machine or in refinement. Those forms include simple variable assignment, IF THEN ELSE END, VAR IN END and sequential composition. One form of substitution that only exists in implementation is WHILE-loop. This is an important form of implementation substitutions with its own proof obligations. A WHILE-loop takes the form of

```
[WHILE P DO S INVARIANT I VARIANT V END] R
```

The operation of the loop is controlled by a while-test, denoted by `P`. The *body* of the loop, denoted by `S`, is executed repeatedly as long as the predicate `P` is true.

As soon as the predicate becomes false, the loop terminates. A loop is normally preceded by a sequence of instructions called loop *initialization* that prepares values in variables that will be used in the loop. The loop *variant* is an expression that denotes a natural number and represents the maximum number of iterations of loop body. The loop *invariant* is a predicate that makes a statement about the values of variables in the loop body. The *invariant* must be true before the loop starts, while the loop progresses and after the loop terminates, as elaborated further in loop correctness proof obligations below.

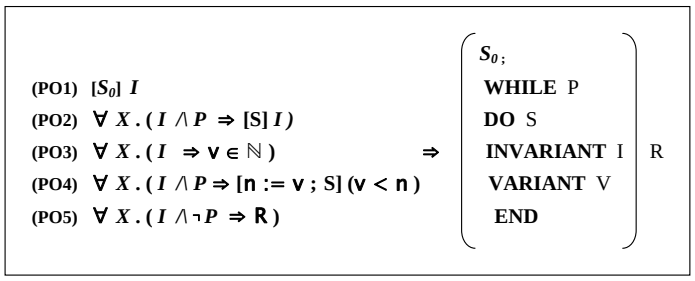


Figure 2.10: Proof obligations of B implementation loop

Figure 2.10 states the rules of loop correctness in the form of proof obligations. The first Proof Obligation (PO1) states that the loop invariant `I` holds on entry to the loop; (PO2) states that the body of the loop `S` preserves the invariant; (PO3) states that the loop variant `V` is a natural number; (PO4) states that the variant strictly decreases on each loop iteration; finally, (PO5) states that the desired result R holds on exit from the loop.

Note: throughout this dissertation, we describe our B-method formalization using machine-readable ASCII symbols. A brief mapping of the commonly used symbols in this dissertation to corresponding representation of mathematical operators can be found in Appendix B.

Chapter 3

Towards a Data Model Evolution Language

3.1 Introduction

In this chapter, we take first steps towards designing a data model evolution language. The main purpose of this language is to allow information system designers to abstractly specify data model changes and subsequently use this abstract specification as a basis for generating platform-specific data migration programs.

One of our aims in this thesis is to look into information systems evolution problem from a model-driven perspective. This aim implies a basic assumption: the information system under consideration has been developed in a model-driven fashion, i.e. it has been specified as a *model* which, consequently, transformed and refined until a working implementation is obtained.

With this aim in mind, at the beginning of this chapter, we depict the basic idea of our approach in Section 3.2. Using MOF abstraction layers as a starting point, we investigate main considerations for designing a data model evolution language and categorize these considerations into two key design requirements, revolving around precise specification of changes and managing change consequences on consistency conditions and data instances.

We then look into relevant state of the art literature. We have identified database schema evolution and model-driven engineering as the main areas of related research. Following [55], a database schema is a description of the information content of a database. As such, we consider Entity Relation (ER) diagrams, Relational models and object models all to be data models, regardless of the underlying database implementation technology. Accordingly, we review schema evolution approaches in object

and relational databases to confirm our identified key requirements and to develop a list of key features that a data model evolution language should present.

The model-driven engineering paradigm treats abstract models as the source code for a range of testing, implementation, and configuration processes. Although little has been done in terms of the generation of data migration implementations, there is clearly related, general purpose work in the areas of model refactoring, model comparison, model weaving and metamodel evolution as we are discussing later in this chapter.

The outcome of the above investigation will be utilized in two ways: first, we develop a list of features that a data modeling language should present. Second, we also identify the kinds of changes typically required when evolving data models. This includes compound evolution steps that modelers typically need to perform and commonly presented in literature. Additionally, we identify techniques used to manage the effect of data model evolution. Particularly, mapping changes to instances in the underlying database.

3.2 Towards a Data Model Evolution Language

Figure 3.1 below illustrates the main ideas of the approach that we are going to investigate in this chapter. Following abstraction hierarchy established in [120], the figure depicts three abstraction layers. A data model, in M1 layer, specifies the structure (data types, relationships and constraints) and operations of an information system [117]. A data model is defined using specific data modeling language (also referred to as metamodel), shown in M2 layer. This modeling language provides the modeling constructs and Well-Formedness Rules (WFRs) that a data model must preserve. Layer M0 represents user data objects or instances that has been collected and persisted.

As data models evolve from one version to another, we would like to be able to describe changes to an existing data model. A data model evolution language thus needs to provide support for describing data model changes. Such changes typically involve *primitive* operations: adding, deleting or modifying model elements (e.g. classes and attributes of a data model). However, in many cases a data modeler needs to change a collection of related model elements. Hence, a data model evolution language would need to provide support for *compound* operations: referring to multiple model elements (e.g. extracting a superclass or merging two classes together).

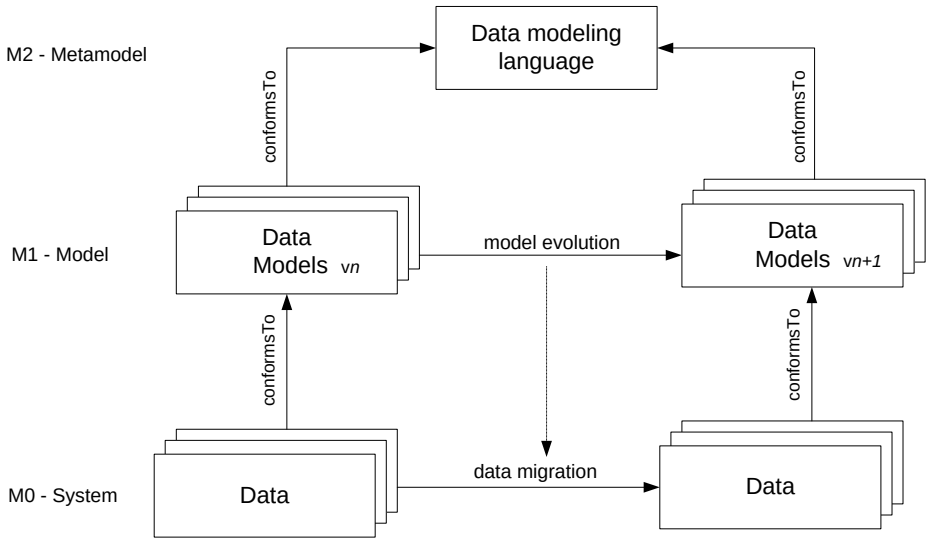


Figure 3.1: overview of the approach

Data models are used for collecting and persisting user data objects. When a data model evolves, a natural consequence would be to propagate data model changes to data instances. A key requirement for data model evolution language is thus to propagate the data model changes to the database, i.e. to derive and execute data migration correctly by implementing the changes specified in data model evolution specification. In many model evolution and data migration scenarios, values of newly introduced or modified attributes or association can be represented in relation to values existing in a source data model. For example, in our evolution specifications of a student management system, we may wish to express that a data value of 'A' in a student grade data element in the evolved model is equivalent to a data value of 'outstanding' in the existing model. It would, therefore, be beneficial if the language can provide support for annotating attributes and associations with expressions representing their new intended value in relation to existing ones.

It is important to note here that the *conformsTo* relationship in Figure 3.1 requires satisfying two kinds of constraints, at two different levels of abstraction. The *conformsTo* relationship between data models and the modeling language (M1 to M2) requires data models to satisfy constraints defined at the modeling language level. Such constraints normally specify how a valid data model can be formed. In addition, the *conformsTo* relationship between data instances and data model (M0 to M1) requires data instances to satisfy constraints defined at data model level. Such constraints represent integrity properties that may restrict values that a data element may have or the way a data element may relate to another data element. As a re-

sult, while evolving data models, a data model evolution language needs to provide support for preserving both kinds of constraints.

Finally, following an evolution, we may want to analyze some of the functionality or properties of the evolved model. For example, we may wish to determine whether the evolved model still conforms to the language in which it was initially written or whether the evolved model still preserves the properties of the source model. To be able to address such requirements, a data model evolution approach needs to provide precise semantics to the data model and to its related changes. Such precise semantics can be exploited in an appropriate analysis framework and provide answers to the kind of questions addressed above.

In summary, from the above explanation, we can synthesize a number of key requirements that need to be addressed by a data model evolution language. These key requirements can be categorized in *two* main categories: *specifying evolution* and *managing the effect of evolution*. The first key requirement deals with the ability to precisely specify different kind of changes to a data model. The second key requirement deals with the ability of the language to maintain model consistency so that an evolved model still conforms to the modeling language. In addition, as the change specifications need to be mapped to executable data migration programs, the language needs to provide mechanisms for confirming that the migrated data instances preserve the integrity rules of the evolved data model.

3.3 State of the Art

Considering the two key requirements which we have identified above, here we examine relevant approaches in two main research areas: database schema evolution and model-driven engineering.

As Figure 3.2 shows, each of the two main research areas include a number of subareas where different aspects of our research have been addressed. In our investigation of these research efforts, we focus on the two key dimensions we have identified: specifying evolution and managing the effect of evolution.

Within database schema evolution, we review object and relational database schema evolution approaches. However, we put more emphasis when reviewing object databases. This is mainly due to the similarity between data models proposed in object database approaches and the underlying data model we consider in this thesis. In addition, by closely reviewing object database schema evolution approaches, we

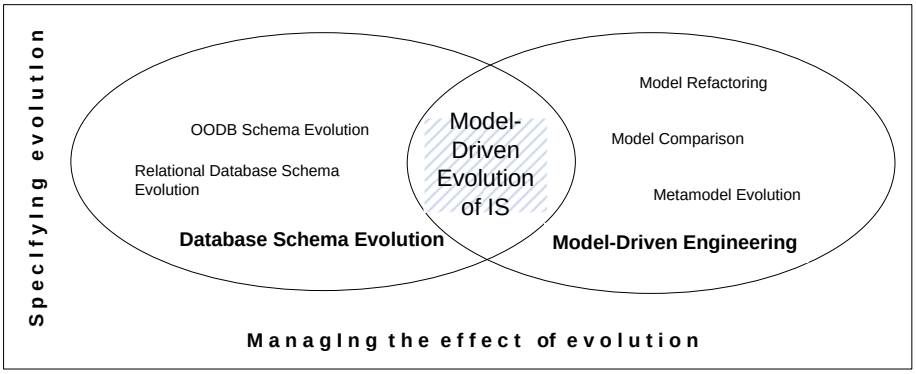


Figure 3.2: Overview of relevant research areas with a focus on two key dimensions

aim to achieve another, more specific, goal. Given that *primitive* evolution operations are largely the same regardless of the object model employed, we aim to identify composition requirements and the kind of *compound* evolution operations that our data model evolution language needs to support.

3.3.1 Database Schema Evolution

Once a database system is developed, various reasons can require changes to be made to the database system (e.g. due to a change in user requirements, a fix to an erroneous condition, or a need to support new applications). These changes can imply modifications to the conceptual schema, logical schema or database state. In literature this process is termed *schema evolution*. It is defined as the process of applying changes to a schema in a consistent way and propagating these changes to data instances while the database is in operation [50]. We start our review of schema evolution literature by looking into approaches that investigate the evolution of object database schemas.

Object Database Schema Evolution Approaches. Several object data model representations have been proposed in literature with corresponding schema evolution approaches, for example [14, 60, 100, 131, 39, 40, 134].

The underlying object data models in most of object-oriented database approaches are, to a large extent, similar. Data models of earlier approaches like [14, 60] consisted of a collection of classes organized in a lattice (rooted and connected directed acyclic graph). Each class has at least one superclass (except object class). Both instance variables and methods of a class can be inherited and overridden. Object data models of more recent approaches such as [39, 40, 134] were based on ODMG standard [165].

Most of Object-Oriented database approaches establish a *taxonomy* of pre-defined schema changes such as addition or deletion of classes and attributes. While some of these approaches did not define any compound evolution operations in their taxonomies like [14]. Other approaches [26, 100, 134] introduced a set of high-level operators. Table 3.1 shows a list of taxonomies proposed by selected object database schema evolution approaches.

In our work and similar to [100] and [134], we assume that compound data model evolution operations can be decomposed into simpler primitive evolution operations. A pre-requisite step here is the provision of composition operators that can allow building compound operations on top of the primitive ones. [26], in particular, showed how to implement compound operators using primitive operators on O2 object model. [134] based the definition of composites on the notion of *link* between classes. Another kind of compound evolution operations, focussing on evolving relationship constructs, was introduced by Claypool et al [40].

Maintenance of data model integrity constraints was an active area of research within the object database schema evolution community. In earlier approaches like [14], integrity maintenance was achieved by using a set of rules for selecting the most meaningful way to preserve the invariant properties for each of the schema changes. A similar approach was adopted by O2 [49]. In more recent approaches like [39], schema invariants are maintained through the use of contract (pre and post condition) that are associated with each evolution operation.

Relational Database Schema Evolution Approaches. The relational data model organizes data in the form of relations (tables). These tables consist of tuples (rows) of information defined over a set of attributes (columns). The attributes, in turn, are defined over a set of atomic domains of values. Many relational database schema evolution approaches such as [42] rely on the Data Definition Language (DDL) statements from SQL (CREATE, DROP, ALTER) to perform schema evolution.

Schema evolution primitives in the SQL language are primitive in nature. Unless there is an extension to the language, each statement describes a simple change to a schema (e.g. adding or dropping individual tables or columns). One such extension is the Schema Modification Operators (SMO) proposed by the PRISM approach [113].

In SMO each statement represents a common database restructuring action that requires data migration. In addition, in PRISM, each SMO statement and its *inverse* can be represented as a logical formula in predicate calculus as well as SQL statements that describe the alteration of schema and movement of data. Another SQL extension

Approach	Primitive evolution operations	Compound evolution operations
ORION - Banerjee et al, (1987) [14]	• Changes to a node (add, drop, rename) • Changes to an edge (making a class a superclass, removing a class from the superclass list, changing the order of superclasses) • Changes to an instance variable (add, drop, rename, change of domain, change inheritance, change default value)	
OTGen - Lerner and Haberman (1990) [101]	• Class (add, delete, rename, add superclass, delete superclass) • Instance variable (add, delete, rename, change type)	
O2 - Ferrandina and Zicari (1995) [60]	• Creation, modification, renaming and deletion of a class • Creation, modification, deletion and renaming of an attribute • Creation and deletion of an inheritance link between two classes	• Extract superclass • Extract subclass • Extract class • Merge classes
SERF [39]	• Add-class • Destroy-leaf-class • Add and delete-ISA-edge • Add and delete-attribute	• Merge-classes-union • Merge-classes-difference • Inline-class • Encapsulate-class • Generalize and specialize-classes
ROVER [40]	• Add and delete-class • Add and delete-ISA-edge • Add and delete-reference-attribute • form and drop-relationship	• Change-cardinality-1m, • Change-cardinality-m1, • Change-relationship-name • Change-type
Schema modification by catalog [134]	• Add and delete class • Add and delete attribute • Add and delete inheritance link	• Specialize super type • Extract class, super and subclass • Inline class, super and subclass • Move feature over reference • Merge classes • ...

Table 3.1: An outline of selected database schema evolution approaches

to support relational schema evolution was proposed by HECATAEUS [129]. Here, a central construct is an evolution policy, which is a syntactic extension to SQL. DB-MAIN is a conceptual modeling platform that offers services that connect models and databases. Based on DB-MAIN platform, a number of proposals such as [41] and [42] were made to ensure that changes to a model should propagate to the database in a way that evolves the database and maintains the data in its instance rather than dropping the database and regenerating a fresh instance.

In a relational model, integrity constraints can take various forms. The most popular types are *entity integrity* and *referential integrity* constraints [55]. The *first* guarantees that no two tuples belonging to the same relation refer to the same real-world entity. In other words, it guarantees uniqueness of keys. The *second* constraint makes sure that whenever a column in one table derives values from a key of another table, those values must be consistent. A great deal of research work has been carried out on the enforcement of integrity constraints in relational databases. In the pioneering work of [35] a general framework is described for transforming constraints into active rules for constraint maintenance. In Trker and Gertz [158] a set of system-independent rules were proposed to implement triggers based on constraint specifications by using Tuple Relational Calculus.

This concludes our review of related approaches in database schema evolution area. We now take a closer look into main relevant approaches in Model-Driven Engineering.

3.3.2 Model-Driven Engineering

In Model-Driven Engineering, we have model refactoring, model weaving, model comparison and model co-evolution as potentially relevant sources of research relevant for the main focus of our thesis.

Model Refactoring. Opdyke [127] and Roberts [139] work focused on refactoring object-oriented programs in such a way that it does not alter the external behavior of the code, yet improves its internal structure. Fowler et al [62] promoted the notion of refactoring as a means of revising design models and hence code. They presented a catalogue of refactoring rules. Although Fowler did not provide a precise definition of his refactoring rules (e.g. using a formal notation), we find his work relevant to to the work we carry out in this thesis : his work can be an appropriate source of candidate model evolution patterns.

Category	Refactoring rules
Moving features between objects	• Move field • Extract class • Inline class
Organizing data	• Replace Data Value with Object • Change Value to Reference • Change Reference to Value • Change Unidirectional Association to Bidirectional. • Change Bidirectional Association to Unidirectional. • Replace type code with class • Replace type code with subclasses • Replace type code with state / strategy • Replace subclass with fields
Dealing with generalization	• Pull up Field • Push down Field • Extract Subclass • Extract Superclass, • Collapse Hierarchy • Replace Inheritance with Delegation. • Replace Delegation with Inheritance

Table 3.2: A selected list from Fowler's refactoring catalog

In his book, Fowler presented an extensive catalogue of refactoring, primarily based on industrial experience. Sixty-eight refactorings were presented and organized into six categories according to the kind of refactoring. Since Fowler's focus in his book was on refactoring design models of object-oriented code which is different from our focus on evolving data models, only a subset of Fowler's refactorings would be relevant for the work we present here. Therefore, when reviewing Fowler's refactoring catalogue, we limited ourselves to those refactoring rules that we find suitable for data modeling and may have a direct impact on persistent data. For example, we can exclude categories such as *composition of methods* which is related to the re-organization and re-structuring of methods. Fowler's refactoring rules that we find relevant were categorized under three main categories: *moving features between objects*, *organizing data* and *dealing with generalization*. Inside these three categories, we had to go one step further and exclude those refactoring rules that deal with the way the code is organized (e.g. *pull up constructor body*) or those which deal with dynamic behavior of an object-oriented program (e.g. *introduce foreign method*). As a result of our

review, we extracted a list of Fowler's refactoring rules which is presented in Table 3.2.

Since our focus is on data model evolution, we are particularly interested in investigating the application of the refactoring notion to models. Most of the work on model refactoring was based on Fowler's refactoring catalogue. Approaches such as [69, 21, 108, 151] proposed different ways for refactoring UML Class diagrams and state chart models. However, since these approaches do not contribute directly to our identified two key requirements, we take note of them here without going further in the review of their details.

The refactoring of a model might result in the evolution of persistent data. Thus, persistent data needs to co-evolve. This aspect of refactoring is relevant to our second key requirement which deals with managing the effect of evolution. We may note that many of the changes made to schemas in Object Oriented Databases (OODBs) as discussed in Section 3.3.1 are similar to refactoring. In spite of this close relation between OODB schema evolution and model refactoring, little was done by the model-driven engineering community to support data migration as a result of data model refactoring [156].

Although the notion of refactoring has been widely investigated, a precise definition of behavior preservation is rarely provided [111]. Fowler, for example, did not offer any characterization to prove this essential refactoring property. Rather, he presented these refactoring steps in an informal way providing step-by-step guidelines for their applications. A similar approach was adopted by Ambler [10] in his work on database refactoring. In model refactoring literature some efforts were made in order to characterize the notion of behavior preservation. For example, in [151], for class diagram refactoring, it was argued that the refactoring steps carried out on the model do preserve the behavior (e.g. creating a generalization between two classes does not introduce new behavior).

As the essence of refactoring is behavior preservation, pure refactoring is insufficient to assist in data model evolution as model evolution could result in introducing new behavior. Therefore, we continue to investigate the literature on other model-driven engineering approaches that involve other structural changes to models.

Model Comparison. Model comparison can be used as a first step towards model evolution. Instead of editing a source model in order to obtain a target (evolved) model, we may wish to build the target model in isolation of the source model and use model comparison technique to identify the difference between the two model

versions. This difference can then be used to derive the migration of the underlying data.

Model comparison approaches aims to realize how a model was evolved and generate a difference between two model versions. Epsilon Comparison Language (ECL) was presented in [94] as a task-specific language of the generic model management language, Epsilon Object Language (EOL) [95]. ECL features a rule-based metamodel agnostic syntax. The main focus of ECL is on matching element(s) of a left model to element(s) of a right model. ECL proposes no specific representation of the result of model comparison (e.g. difference model) nor any integration with transformation techniques. In [70] the capability to find mappings and differences between models was termed *Model differentiation*. The authors presented a metamodel-independent algorithm and a tool (DSMDiff) for detecting mappings and differences between domain-specific models. The algorithm presented determines only if the two models are syntactically equivalent. Structural differences in DSMDiff are shown in a tree browser with coloring and iconic notations which lack the ability to integrate into other MDE processes (e.g. model transformation).

EMF Compare [80] is another model comparison tool which is part of the larger Eclipse Modeling Framework Technology (EMFT) project. The objective of this component is to provide a generic and scalable implementation of model comparison. Although EMF Compare tool represents model difference as a model which makes the tool appealing for the purpose of our approach, it offers limited support considering our data migration scenario. The generic capabilities of the tool implies that it is at a too high level of an abstraction that makes it inflexible to domain specific customization required for model-driven data migration approach.

In general, while model comparison and difference representation approaches address a similar problem : structural evolution of models, there are three important differences from the work we are investigating. First, in these approaches, a primary assumption is that a target model already exists and the focus is on calculating and characterizing the difference between a source model and a target model. Second, model comparison and difference representation approaches seem to limit their scope to the structural elements of a model with no consideration to integrity constraints or well-formedness rules. Third, these approaches tend to be generic and do not focus on information system evolution specific requirements such as data migration.

Metamodel Evolution and Model Co-Evolution. A number of approaches proposed to address the problem of metamodel evolution and model co-evolution i.e.

adapting (migrating) models that conform to an older version of a metamodel to a newer version of the metamodel.

[78] Investigated the potential of automating model migration in response to metamodel changes by analyzing the evolution history of two industrial metamodels. A classification of metamodel changes based on potential automation was presented as a basis for specifying the requirements for effective tool support. Another classification of metamodel changes in EMF/Ecore was presented by [32] which introduced a process model that defines the necessary steps to migrate model instances upon an evolving metamodel. Wachsmuth [160] proposed an operator-based approach for metamodel evolution and classifies a set of operators according to the preservation of metamodel expressiveness and existing models. Cicchetti et al. [37] proposed to represent metamodel changes as difference models conforming to a difference metamodel to identify semi-automated countermeasures in order to co-evolve the corresponding models. Here, the co-evolution of models induced by the metamodel changes are reduced to three cases which are: $\Delta MM[resolvable]$ model containing resolvable changes (e.g. extracting abstract superclass) , $\Delta MM[unresolvable]$ with unresolvable changes (e.g. adding obligatory meta-property) and non-breaking changes that do not need to be propagated and are ignored (e.g. adding non-obligatory meta-property).

In [78] and [37] metamodel evolution is specified by a sequence of operator applications. To support the co-evolution of models, each operator application can be coupled to a model migration separately. Operator-based approaches generally provide a set of reusable operators which work at the metamodel level as well as at the model level. At the metamodel level, a coupled operator defines a metamodel transformation capturing a common evolution. At the model level, a coupled operator defines a model transformation capturing the corresponding migration.

Our work aims to deal with data model changes (M1) which has consequential effects on data instances (M0) and require explicit treatment of data model constraints to ensure the semantic integrity of the data is preserved. Moreover, with the exception of [37], classified changes are not presented in models; hence lack the ability to integrate into other MDE processes (e.g.model transformation).

3.4 Our approach

In this section we synthesize the key requirements that need to be addressed by a data model evolution language and highligh, at a high level, the main components

Specifying evolution

- Ability to express primitive evolutionary changes
- Support for compositionality
- Support for semantic linkage of data

Managing the effect of evolution

- Maintenance of model consistency
- Maintenance of data integrity
- Support for data migration

Figure 3.3: Main requirements of data model evolution language

of our approach. These components will be elaborated in great detail in subsequent chapters.

3.4.1 Synthesizing design requirements

As we have seen throughout the previous section; there is a considerable body of related and applicable work addressing different aspects of information systems evolution and data migration. Database schema evolution approaches that we have reviewed in section 3.3.1 provide solid theoretical foundations and interesting methodological efforts, the lack of abstraction was observed in [137] and remains largely unresolved after many years. Conversely, model-driven engineering approaches which we reviewed in section 3.3.2, promote the idea of abstracting from implementation details by focusing on models as first class entities. Approaches such as model weaving, model refactoring, model comparison and metamodel evolution may help in characterizing information systems evolution however, for our purpose, they remain largely general-purpose and offer no specific support for information systems evolution tasks such as data migration.

From the review of the above literature, we can synthesize requirements that need to be addressed by a data model evolution language. We use the two main categories we identified earlier in section 3.2 to classify a number of more specific and desirable features for a data model evolution language. This list of requirements is presented in figure 3.3 and briefly explained below.

Under evolution specification category, a data modeler must be able to precisely specify two kinds of changes on a data model: primitive and compound. Given the constructs of a particular data modeling language, a model evolution language should enumerate possible changes and derive a basic notation for the evolution of data models in that language. This can be achieved by considering the ways in which each kind of model element may be added, removed, or modified. Additionally, a data model evolution language should define combinators. Using combinators, a modeler can compose compound evolution operations to describe changes to a number of different model elements. In addition, the language needs to include a facility for annotating attributes and associations with expressions representing their new intended value in relation to values existing in a source data model.

From the perspective of managing the effect of evolution, it is important to note that a data model rarely exists in isolation. On one hand, a data model must be a valid instance of a data modeling language, and on other hand, valid data models are used to collect and persist user data objects. A data model evolution language thus needs to provide mechanisms for maintaining the integrity of both *model* and *data* integrity. Within the domain of information systems evolution, the value of any data model evolution language is greatly minimized if it does provide ways for supporting data migration. Data model changes can have direct impact on existing data instances. As a result, upon data model evolution, data migration needs to be specified. This can be achieved by describing corresponding data transformations by which we can adapt data collected and persisted based on a source data model so that it conforms to a target data model.

3.4.2 Main elements of our approach

The key requirements we have synthesized above may be met by a model-driven approach to data model evolution and data migration. As a primary technique, our proposed approach uses metamodeling and model transformations to develop data migration implementations. Figure 3.4 shows an overview of our proposed approach. The figure presents a package diagram representing metamodels defining the main components of our approach. All these metamodels are modeled using Meta-Object Facility (MOF)[122], as a metamodeling language. Our metamodeling approach consists of a number of metamodels. Each metamodel defines a component of the approach and is represented in a separate package. The combination of these packages defines our model-driven approach to information system data migration.

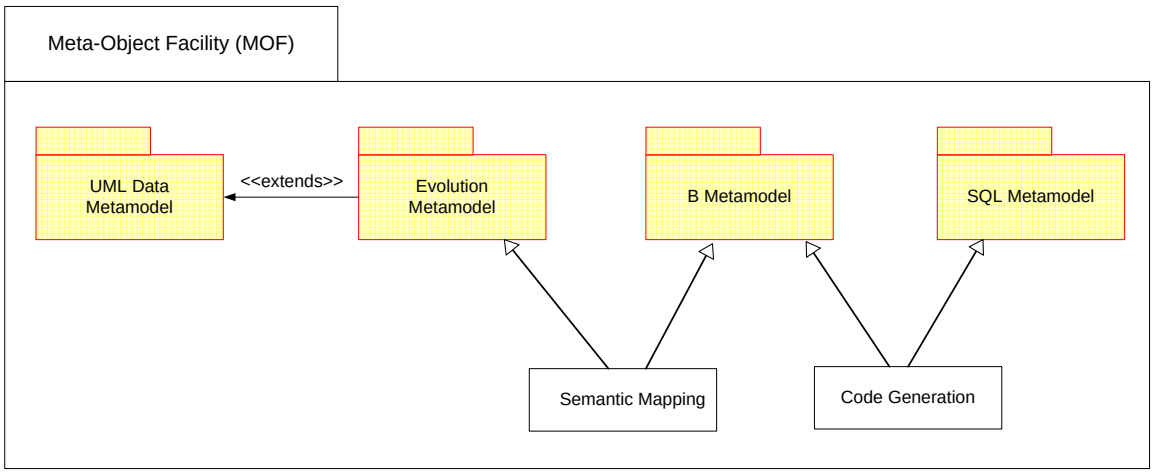


Figure 3.4: Overview of the metamodeling approach

In our approach, each development step is represented by a model, which is an instance of the corresponding metamodel. We move from one development step to the other using model transformation. By expressing our solution at a metamodel level, in essence, we are proposing a general reusable solution for describing the evolution of all data models conforming to our selected data metamodeling language (e.g. UML).

Because these metamodels and model transformation artifacts are the basic essential elements of our approach, we elaborate more on their contents and use below. As shown in Figure 3.4, There are three main packages and two model transformation algorithms:

- **UML Data Metamodel** : is a subset of UML core package that we have defined to model domain state of an information system. Although, as we outlined in Section 2.2, different modeling notations can be used in Model-Driven Engineering of information systems, we focus on using the Unified Modeling Language (UML)[124] for describing model elements and the Object Constraint Language (OCL) [121] for describing constraints. This metamodel is discussed in greater details in Section 4.2.

- **Evolution Metamodel** : is an extension of the UML Data Metamodel. It defines the abstract syntax of our proposed evolution modeling language. This metamodel will also be presented in more details in Section 4.3.

- **B-method Metamodel** : building on the introduction of the main concepts of B-method language we presented in Section 2.3, this MOF-based metamodel

encapsulates the abstract syntax of B-method language constructs. This metamodel is used for giving precise semantics to both our data modeling and evolution modeling languages.

We explain the way we use B-method to assign formal semantics to our data and evolution metamodel in Chapters 5. In Chapter 6, we explain how we transform an abstract representation of B-method abstract machines into refinment and implementation constructs. In Appendix A, we elaborate on the use of B-method metamodel within a model-driven engineering context.

- **SQL Metamodel** : to generate an appropriate implementation of our model evolution and data migrations, we require a mapping from our abstract model operations to operations upon a specific, concrete platform. Given the prevalence of relational database implementations, we assume that the data we want to evolve is persisted in a relation database and use a metamodel for the Structure Query Language (SQL) to generate executable data migration code.

  In Chapter 6, we explain how the main elements of SQL metamodel may be given formal semantics in B-method framework. This step is necessary in order to establish a formal refinement relation between proposed abstract evolution specifications and corresponding SQL code representation. In Appendix A, we elaborate on the use of SQL metamodel within a model-driven engineering context by describing how the required SQL migration code is generated.

- **Semantic Mapping** : we use model transformation technique to link the abstract syntax of the Evolution Metamodel to the abstract syntax of B Metamodel. This mapping gives the evolution language its meaning in terms of B-method constructs. As we will explain in greater details in Chapter 5, we distinguish two types of semantics: static and dynamic. Static semantics refers to precise description of data model constructs and consistency conditions of constructs in the data metamodeling language, and is specified as invariant conditions that must hold for any data model created using the evolution language. Dynamic semantics refer to the interpretation of a given set of model evolution operations in the context of data models.

- **Code Generation** : this is a model-to-text transformation to transform an SQL model, a valid instance of SQL Metamodel, into a corresponding collection of SQL statements. The generated statements include both Data Definition

Language (DDL) statements to update table and column structure and Data Manipulation Language (DML) statements to migrate data records.

Although Figure 3.4 shows only two mappings defined by model transformation algorithms, other mappings exist in the approach and are not shown in the diagram for clarity of presentation. For example, as we will show in the implementation section in Appendix A, we map the abstract syntax of the Evolution Metamodel into a context-free EBNF grammar to generate an editor that can be used to edit data models, conformant to our UML Data Metamodel. Another example is the refinement transformation we perform in Chapter 6. In this transformation activity, we map an instance of B Metamodel constructs corresponding to (an object-oriented) evolution model to another instance of B Metamodel corresponding to (a relational) SQL model.

Chapter 4

Modeling Data Model Evolution

In this chapter, we define two modeling components of our model-driven approach : data model and evolution model. In our context, data models include information about structural aspects of an information system design. Evolution models, on the other hand, include abstract description of data model changes. As UML is our selected modeling language, naturally, these two modeling components need to be defined in UML.

Our proposed UML data model will use those UML concepts which are essential for modeling aspects related to the structural concepts and their relationships. Since semantic integrity is an important concern in data modeling, these structural concepts will need to be augmented with a constraint language, precisely defining how data elements may be related or restricting the domain of certain data values.

In the context of model-driven engineering, an evolutionary step in the design of an information system corresponds to a change in the system model: removing or adding features, changing properties or associations. To capture such evolutionary changes, we must extend the language in which the data model itself is written; in this case UML.

Since our ultimate goal is to migrate data collected against an old data model into a form suitable for storage against a new data model, it may not be sufficient to record information about the intent of the evolution at the data model level. If we are able to define the ways in which the values of attributes in the new model are related to the values of attributes in the old, we can generate an appropriate data transformation, applicable at the instance level, which can be transformed into an appropriate data migration implementation

This chapter is structured as follows. Section 4.1 briefly describes our modeling approach in general. Section 4.2 discusses our data modeling approach. Since we use

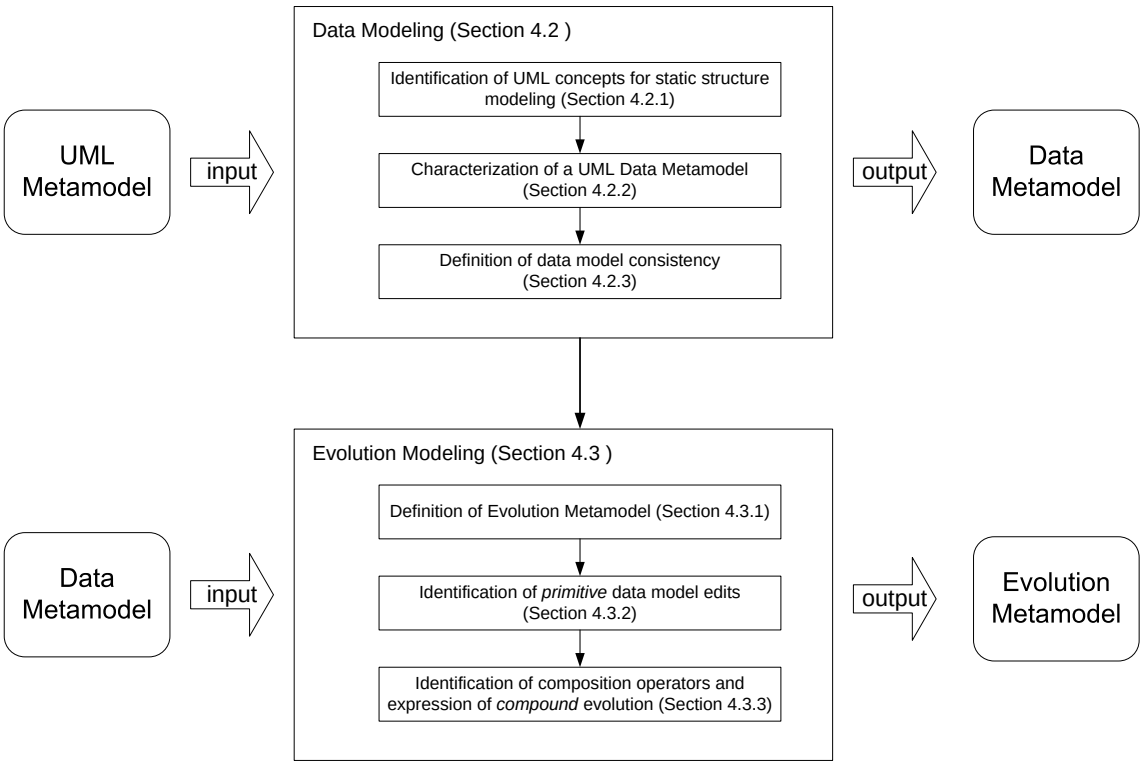


Figure 4.1: Main steps of data and evolution modeling approach

UML as the modeling language, we first identify UML concepts relevant for structural modeling. We then explain the method we followed to characterize a UML metamodel subset, appropriate for data modeling and we discuss conditions required for data models to be consistent. Section 4.3 proceeds with a definition of an abstract syntax of evolution models, in the form of an Evolution Metamodel. In this section we first discuss how a UML-based metamodel (similar to our Data Metamodel) may be extended. We then present an algorithm for generating an evolution model, suitable for capturing changes to data models, written in our proposed data modeling language. Section 4.4 presents 'induced data migration' as an important realization of our modeling approach. The chapter closes with a discussion that establishes the need for extending our modeling approach with appropriate formal semantics.

4.1 Modeling Approach

Our approach to model data model evolution [1, 3, 2] can be summarized in the following two steps (see Figure 4.1):

Step 1 : Data model structural concepts are described using a UML Data Metamodel. This metamodel includes concepts typically found in a UML class diagram such as classes, properties and associations. We have followed a systematic method to identify these elements within UML packages relevant for describing static structures. This metamodel (at level M2 of MOF hierarchy) can be instantiated into data models (at level M1) describing user data applications from various domains (e.g. employee information, student registration, part ordering etc). Once data models are in place, they can be used to collect and persist data objects (at level M0). For a data model to be consistent, elements at various abstraction levels, need to satisfy a set of consistency constraints. As we will elaborate in Section 4.2.3, two kinds of consistency constraints need to be preserved : constraints defined by the modeling language (i.e. UML Data Metamodel) need to be respected by data models; while constraints defined by the data model need to be respected by data objects at the instance level.

Step 2 : Evolutionary steps of UML data models are described using an Evolution Metamodel. This evolution metamodel (at level M2 of MOF hierarchy) is an extension of the Data Metamodel, defined in step 1 and can be instantiated in an evolution model (at level M1), abstractly describing data model changes. Using model transformation, the abstract evolution model can be transformed into a program in an implementation language (e.g. SQL) to migrate the data (at level M0) from one data store to the other.

Our modeling approach has several benefits. The most important among them are the following:

1. *Decoupling from underlying implementation platform.* Both modeling components which we describe in this chapter are abstractly defined in UML. As such, they are not tight to any underlying implementation technology. Using appropriate model transformation techniques (see Section 2.2.4), information system designers can translate these modeling artifacts into executable programs in a chosen platform-specific implementation. This separation between modeling and implementation concerns can enable *induced data migration*, as we will explain in Section 4.4.

2. *Reusability.* Defining the approach at a metamodel level of design implies that resulting metamodel artifacts are reusable : they can be generically used as

templates to describe various data models and different data model evolution scenarios.

3. *Relevance.* The use of a standard modeling framework such as UML for data and evolution modeling makes our approach relevant for potential usage by a wide range of audience in software engineering community; and integration with a large number of existing supporting tools.

4. *Extendability.* While the set of primitive model edits that make up an evolution model can be identified : each data model element may be added, modified or removed, compound evolution scenarios can be fairly large. Using our evolution metamodel, new evolution patterns can be defined by composing primitive model edits or existing evolution patterns to describe new and emerging evolution scenarios.

4.2 Data Modeling

Although UML is mainly known for designing object-oriented program code, it is becoming a popular tool in data modeling community [138],[115] and [116]. Yet, there is no generally accepted standard for data modeling based on UML. In the absence of a standard UML data modeling extension, we characterize an adequate metamodel for modeling information systems data using a subset of UML constructs augmented with language-level constraints relevant for data modeling. In this section, we elaborate on the steps we followed to define a UML Data Metamodel.

4.2.1 UML concepts for static structure modeling

The first step our UML data modeling is to identify concepts relevant for modeling structural aspects of an information system design. We have identified *objects*, *classes*, *properties*,*generalization*, and *association* as the fundamental concepts needed for developing data models using UML. As these concepts are the essential building blocks of our Data Metamodel and have a direct influence on the way we define our data model evolution, it is important to describe what do they mean within the context of UML structural modeling.

Object is the most fundamental concept of UML for describing structural aspects of a system. Objects do not only represent real world entities (such as employees, departments, etc.), but they also describe abstract concepts like employment or assignment. Structural properties of an object are described as a set of attributes, which

can be instantiated into slots holding data values. Objects are described by classes. A class describes the common properties of a set of objects of a domain. When a class is abstract, it cannot have any objects. Objects do not stand alone; they are always tied to their classes. The terms *instance* and *object* are largely synonymous and, from now on, will be used interchangeably.

Common properties of a set of classes can be placed in a parent class which inherits these properties to its child classes. This forms a generalization relationship between the parent class (superclass) and the child classes (subclasses). Closely related to generalization is the notion of inheritance, mainly known from object-oriented programming languages. The child inherits structure, behavior, and constraints from all its parents. In a simple case, a class has a single parent. In a more complicated situation, a child may have more than one parents. This is called multiple inheritance (see [152] for a comprehensive overview). In our defined Data Metamodel, we only allow single inheritance.

Associations are the UML primary constructs for expressing relationships in a model. An association has a name and two or more association ends defining roles played by classes participating in the relationship. An association end can be paired with an opposite association end to form a *bidirectional* association. In such a case, both ends need to be owned by the same association. In addition, each association end has a minimum and maximum multiplicity bound. The minimum multiplicity bound indicates that all objects of the owning class must be related, at least, to a minimum objects of the associated class. The maximum multiplicity bound defines that an object of the owning class cannot be related with more than a maximum objects of the associated class. At the instance level, an association is instantiated into a link which holds references to objects of classes participating in the association.

4.2.2 Characterization of a UML Data Metamodel

Now that we have identified the basic UML concepts that are needed to describe data models, we need to present them in a data modeling language (in the form of metamodel). Defining a UML-based metamodel supporting the structural modeling concepts which we have identified requires mapping these basic concepts into appropriate modeling elements in UML metamodel. This mapping would provide us with a set of elements with their associated UML definitions and characteristics.

UML metamodel is organized into a vast number of packages from which various diagrams may be constructed to model different perspectives and views of an application. Since the focus of our work is on data modeling, as we discussed above, we only

Element	**Data Metamodel**	**Essential**	**Persistable**
Element	✓	✓	
NamedElement	✓	✓	
Type	✓	✓	
TypedElement	✓	✓	
Class	✓	✓	✓
MultiplicityElement	✓	✓	
Operation			
Parameter			
Property	✓	✓	✓
DataType	✓	✓	✓
Enumeration	✓	✓	✓
EnumerationLiteral	✓	✓	✓
PrimitiveType	✓	✓	✓
Comment			✓
Package			✓

Table 4.1: Elements of UML Core::Basic package

consider aspects related to modeling static structures. UML elements corresponding to our identified structure modeling concepts are defined in the *Core* package of the infrastructure library ([124], pp.27) which define a meta language core that can be reused to define a variety of metamodels. The Core package is subdivided into a number of sub-packages: *Basic*, *PrimitiveTypes*, *Abstractions*, *Constructs* and *Generalization*. As a convention, we use '::' to denote navigation from one package to its sub-package. Core::Basic is of particular interest to our work as it provides a minimal class-based modeling language on top of which more complex languages can be built.

Table 4.1 lists all elements (metaclasses) of the Basic package. For each element, the table shows whether the element is included in our UML subset (Data Metamodel column). The selection is based on two criteria. First, whether the element is required for modeling structural aspects (Essential column) i.e. does it map directly to one of our identified structural modeling concepts?. This is a compulsory criterion that determines whether an element can be included or excluded from our subset. Second, whether the element can persist data items (Persistable column). This is an important but not a compulsory criterion. This criterion is important because we are going to depend on it to identify elements that can be instantiated and subsequently, present a corresponding instance layer. But it is not a compulsory criterion since although abstract metaclasses cannot have direct instances, they can play an important role in structuring and organizing the data model (e.g. MultiplicityEelement metaclass).

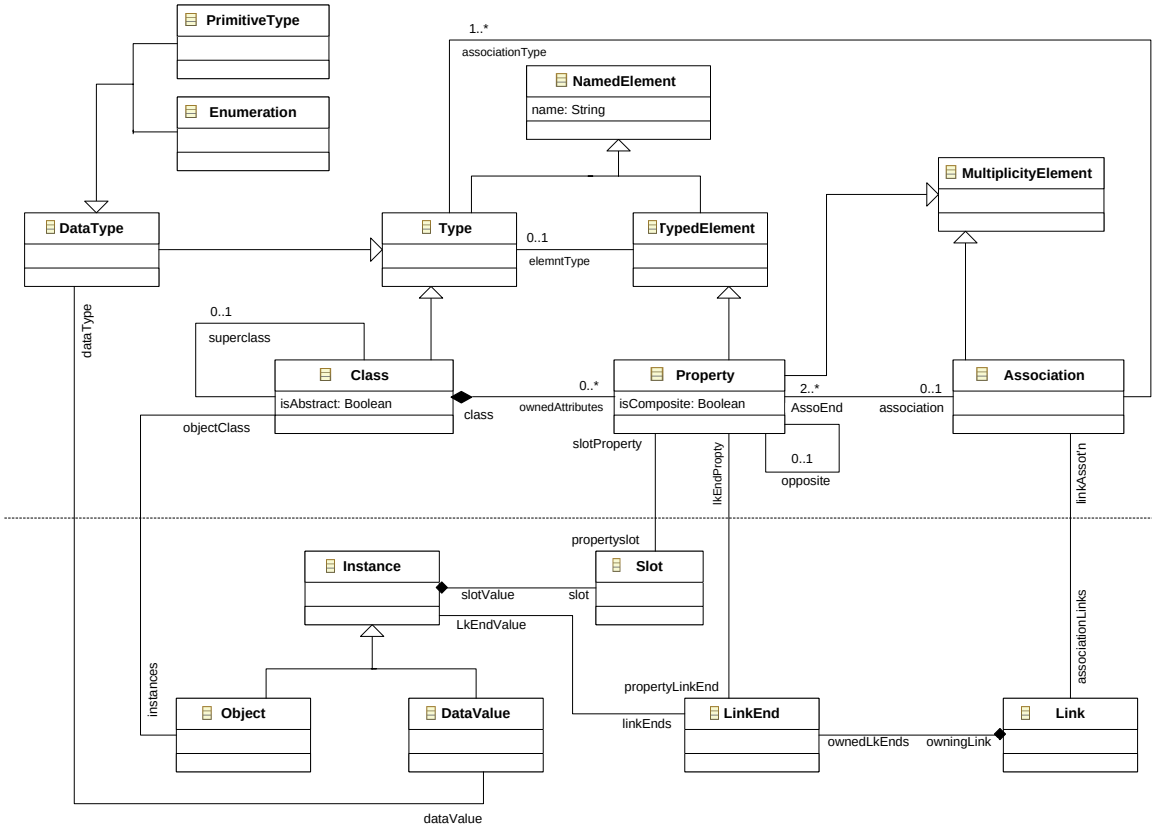


Figure 4.2: A subset of UML metamodel

The table shows the result of applying both criteria to elements of the Core::Basic package and accordingly, were included in or excluded from our metamodel subset.

From a UML data modeling perspective, it is important to note here that the elements above which we extracted from the Core::Basic package are closely related to concepts modeled in other Core sub-packages. For example, the Property element can be typed by a one of the types pre-defined in the Core::PrimitiveType package and Class element (as a Classifier) can be organized in an inheritance hierarchy based on concepts defined in Core::Abstraction::Generalization package.

Table 4.1 describes the elements that we selected from UML to include in our data modeling language (i.e. Data Metamodel). However, the table does not explain how these elements may be used or how they are related to each other. Figure 4.2 shows the resulting subset in a UML class diagram that shows how these elements are modeled and the relationship between them. A brief summary of the elements included in this figure follows. Model elements with names can be represented by the abstract NamedElement class. A Type is a named element that is used as the type

for a TypedElement and constrains the set of values that a typed element may refer to. A Class is a type that has objects as its instance. A Class may participate in an inheritance hierarchy using *superclass* property. A MultiplicityElement is an abstract metaclass that defines an inclusive interval of non-negative integers beginning with a lower bound and ending with an upper bound. A Property is owned by a class and is a typed element that may be typed by a data type and shown as an *ownedAttribute* of the class or typed by another class and shown as *associationEnd* of Association. A property may be paired with an *opposite* property to represent a bidirectional association. A DataType is a class that acts as a common superclass for different kinds of data types. A DataType is specialized in PrimitiveType that includes Integer, Boolean, String and UnlimitedNatural. An Enumeration is a set of literals that can be used as values.

In the UML, the data held in a system can be characterized as a model instance, represented by elements at the lower part of figure 4.2. More specifically, using concepts from UML Instance package ([124], pp.53), the lower part of figure 4.2 shows how system data can be presented as instances of a data model. An Instance is a model element that represents modeled system instances. The kind of instance depends on its corresponding data model classifier element. An Object is an instance of a Class; Slot is an instance of a primitive-typed Property; and a Link is an instance of an Association. A Slot is owned by an instance. It specifies the values of its defining Property. A link contains two LinkEnds, each, in turn, contains a reference to an object participating in an association. Objects, slots, values and links must obey any constraints on the classes, attributes or associations of which they are instances.

4.2.3 Consistency of UML data model

For a model to be *consistent*, it has to *conform* to its metamodel definition. A model conforms to its metamodel definition when the metamodel specifies every concept used in the model definition and the model uses the metamodel concepts according to the Well-Formedness Rules (WFRs) specified by the metamodel [128]. Consistency can be described by a set of constraints between a model and its corresponding metamodel. When all consistency constraints are satisfied, a data model is said to conform to its metamodel.

The conceptual data metamodel we have presented in section 4.2, which is based on UML metamodel and its structural properties, thus, provides a unifying means to allow us to differentiate consistent from inconsistent data models. More specifically,

Syntactic consistency

```
--Name Uniquness
context Class inv :
   self.allInstances()->forAll(c1,c2 |
      c1.name = c2.name implies c1 = c2)

--Absence of circular inheritance
context Class inv :
   self.allInstances()->forAll(c|
      not (c.AllParents()->includes(c)))

--Existential dependency of composition
context Property inv :
   self ->allInstances()->forAll(p |
      Class->allInstances()->exists(c|
         c.ownedAttributes->includes(p))

--Association bidirectionality
context Association inv :
   self.associationEnds->forAll(ae1,ae2|ae1<>ae2
      and ae1.association = ae2.association
      implies ae1.type = ae2.class and
                  ae2.type = ae1.class)
```

Figure 4.3: OCL characterization of data model syntactic consistency

as our presented metamodel characterizes two layers of abstraction: data model and instance model, we can distinguish between two sets of consistency constraints. First, consistency constraints that are defined by the modeling language and need to be satisfied by the data model. Such constraints can be used to check data model *syntactic conformance*. Second, consistency constraints that are defined by the data model and need to be satisfied by the instance model. Such constraints can be used to check data model *semantic conformance*.

Following the discussion of the structural properties of our selected UML subset above, we can synthesize a number of data model consistency constraints in terms of syntax and semantics. Below we list a number of these consistency constraints. The set of constraints we list below is by no means an exhaustive list of consistency constraints for data models, but it lays a foundation for demonstrating how our proposed approach may be used to maintain consistency of evolving data models.

Definition 1 (Syntactic Consistency). A data model is syntactically conformant when it fulfills WFRs defined by the modeling language (UML). A metamodel for UML contains considerable number of structural properties. However, for explaining how we may maintain syntactic conformance, we consider the consistency constraints stated in Figure 4.4 (using OCL) and explained below:

Semantic consistency

```
--Instance conformance:
context Object inv:
   self.class.ownedAttributes->forAll(a|
      self.ownedSlots->exists(s|s.property =a))

--Link conformance
context Object inv :
   self.class.oppositeAssociationEnds()->
      forAll(ae|exists(le|self.selectedLinkEnds(ae)->
         size()>= ae.lower and
            self.selectedLinkEnds(ae)->size()<= ae.upper))))

-Value conformance:
context Slot inv:
   self.value.oclIsTypeOf(DataValue).
      dataType.checkDataType()
```

Figure 4.4: OCL characterization of data model syntactic consistency

- *Name uniqueness*: this constraint involves data model *classes*, *associations* as well as *properties* (*attributes* and *associationEnds*). The scope of name distinction of a property is the union of native and inherited properties of its owning class.
- *Absence of circular inheritance*: a class cannot generalize itself directly or indirectly, i.e., no cycles should exist within class inheritance hierarchy.
- *Existential dependency of composition*: a *composite* determines the life span of its *components*. When a composite instance is deleted, all of its components are recursively deleted.
- *Association bidirectionality*: this involves two association ends paired together to form a bidirectional association. In such a case, both ends need to be owned by the same association and each association End becomes an opposite of the other end.

Definition 2 (Semantic Consistency). An instance model consisting of objects, slots and links must satisfy all consistency conditions defined by its corresponding data model. This denotes *semantic consistency* of an instance model to its data model. This *semantic consistency* can be characterized by a number of consistency constraints that link instance-level elements to corresponding model-level elements, thereby establishing a certain set of conditions that must be satisfied for any valid

instance model. If an instance satisfies all these constraints, we say that the instance semantically conforms to its data model. Based on UML structure properties we described above, we can synthesize the following *semantic consistency* conditions:

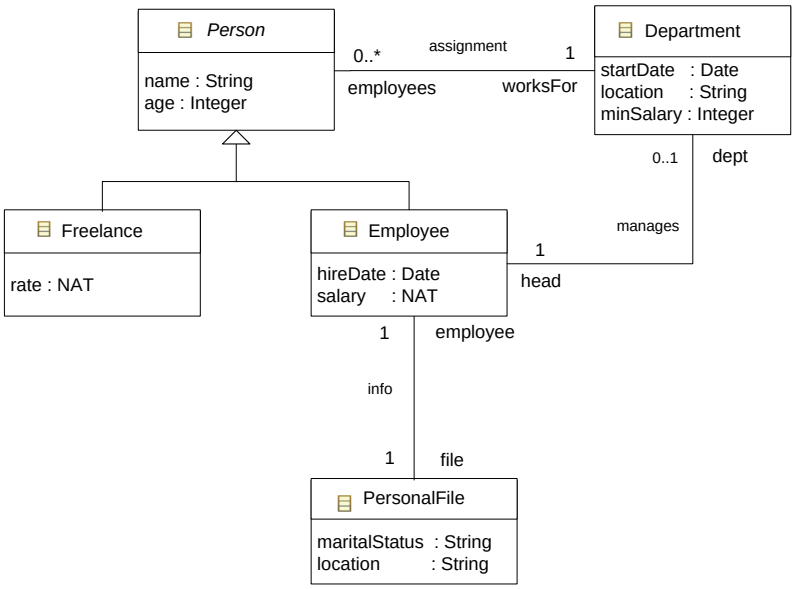
- *Instance conformance:* an instance conforms to its type if each instance slot conforms to instance's class property which is defined by the class or its superclass. The instance should also fulfill all further WFRs which are defined in the context of the instance's class or its superclasses.
- *Slot conformance:* a *slot* conforms to its *property* if the value of the slot is consistent with the data type of the property.
- *Link conformance:* a *link* conforms to its *association* if the objects it references are consistent with the association type and multiplicity, and the objects it references satisfy the bidirectional association defined at the data model level.

Finally, we should note that there is no reason to separate an instance model from its data model by disregarding semantic consistency constraints. Such an instance model, in most cases, would not exhibits integrity or contains enough information to be understood.

Example. Keeping with the running example we have introduced at the beginning of Chapter 2, Figure 4.5 represents the initial version of our simplified EIS data model. The workforce in the company presented in this data model can either be employees, represented by class Employee, or contractors, represented by class Freelance. Both kinds of workers are considered persons, represented by Person class. A person is assigned to a department, represented by Department. In addition, employees' personal data maintained by the company is represented by PersonalFile class.

Each of the above classes contains a number of attributes. In addition, the model has a number of associations to represent relationships between persons and the department which they work for, an optional manager relationship and the personal information held about an employee.

The current version of the model is restricted by a number of constraints, written in OCL, shown below the model. This initial version of the model was instantiated into a valid model instance shown in Figure 4.6.




```
/* An employee's salary must be greater than the department minSalary*/
context Department inv C1 : self.employees ->forAll(e|e.salary >=
self.minSalary)

/* employees and department are bi-directional associations*/
context Employee inv C2 : self.department.employees ->includes (self)

/* An employee's hire date must be after the department start date*/
context Employee inv C3 : self.department.startDate < self.hireDate
```

Figure 4.5: Data model of a simplified Employee Information System

4.3 Modeling evolution

In the previous section, we have characterized a subset of UML metamodel as a data modeling language that can be used to describe data models. In this section, we describe Evolution Metamodel : the second part of our modeling approach, presented in Figure 4.1. In particular, we present the modeling language based on which we can write evolution models describing how data models, conforming to our Data Metamodel can be evolved.

To be able to evolve data models written in our UML subset, we extend our Data Metamodel with a set of model-level edits in the form of model operations. Such operations are drawn from the model elements of the UML Data Metamodel itself : each element in the Data Metamodel may be added, modified or removed. Manual specification of such operations can be error-prone and time consuming activity, hence we show how such set of model operations can be automatically generated. Before, we do that, however, we first give an account of how a UML-based metamodel such as our Data Metamodel may be extended.

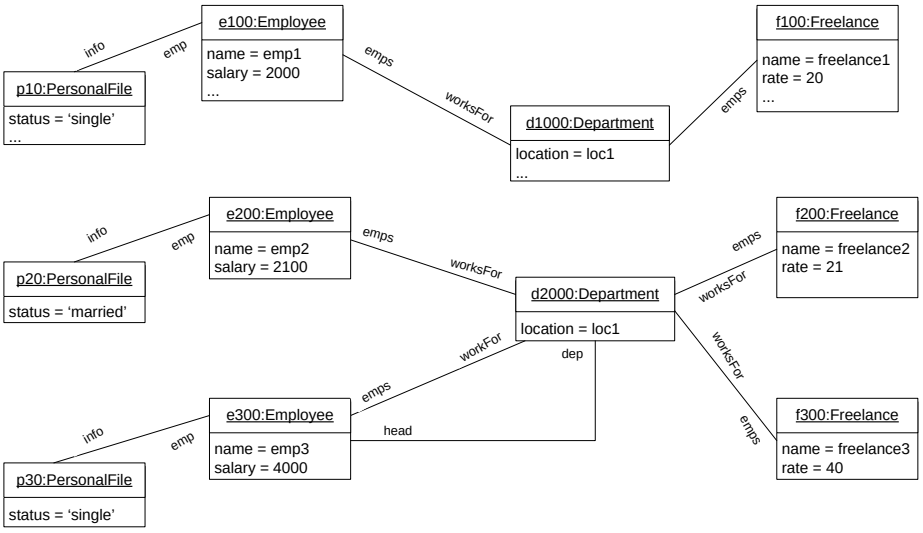


Figure 4.6: Instance model of the simplified Employee Information System

4.3.1 Definition of evolution metamodel

UML is frequently used as a general-purpose modeling language, but it can also be extended to meet the requirements of specific domains. In the context of our work, we extend UML metamodel, part of which is shown in Figure 4.2 , to cater for the domain of information systems evolution.

An extension to the UML metamodel may be achieved in two different ways [87], as a *lightweight* or *heavyweight* extension. The first of these involves the definition of a UML 'profile', using extension mechanisms included within the standard UML profiles package ([124], pp.179). This approach has been used successfully for notations like CORBA [119], but is limited in its application: any new concept or construct needs to be encoded in terms of the existing concepts within UML, creating problems with interpretation, expressiveness, and flexibility. Instead, here we present a *heavyweight* extension, similar in the nature to the ones defined for the Common Warehouse Metamodel [154] and the Software Process Engineering Metamodel [155].

Definition 3 (**Metamodel Extension**).
In line with the OMG metamodel hierarchy concepts introduced in Section 2.2.1, to extend a metamodel, we need to consider its meta-metamodel (at level M3 of MOF hierarchy). Considering the metamodeling language (in this case MOF) allows use to distinguish the different concepts, properties and the relationships the modeling language includes. Accordingly, we can declare *extend* as a mapping function

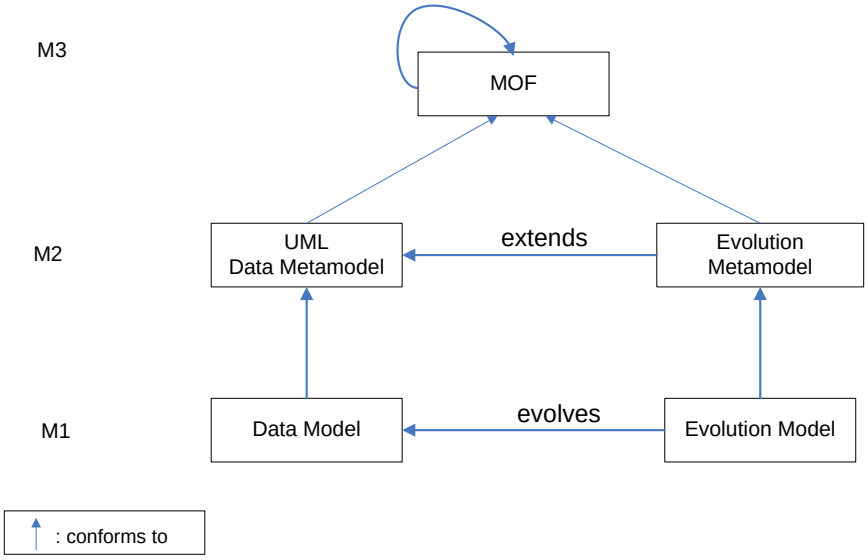


Figure 4.7: Illustration of UML metamodel extension

that takes metamodeling language concepts (at level M3) together with their metamodel instances (at level M2) as input and return an extended (evolution) metamodel consisting of the original concepts of the input metamodel with their corresponding evolution operations. More specifically, this function can have the following type:

$$extend : (\mathcal{MMM} \times \mathcal{MM}) \nrightarrow \mathcal{MM}$$

where $\mathcal{MMM}$ denotes the metamodeling language (M3) and $\mathcal{MM}$ a modeling language (M2). *extend* is a partial function (denoted by $\nrightarrow$) since it only operates on a subset of the specified metamodel concepts (e.g. abstract classes are not associated with evolution operations).

Using MOF, we can extend UML metamodel by introducing evolution operations. As such, each UML metamodel concept will be associated with primitive evolution operations to add, delete or modify instances of that concept at the model-level. The primitive evolution operations will be combined into an evolution metamodel and instantiated into an evolution model as illustrated in Figure 4.7.

Algorithm 1 shows part of the algorithm used for generating evolution metamodel. This algorithm takes as input the UML Data Metamodel, its corresponding meta-metamodel (MOF) and OCL Metamodel and returns as output an evolution metamodel with the operations required to evolve elements of the Data Metamodel. The algorithm generates the basic model edits as instances of MOF Operation Class. It also generates operation parameters (as instances of MOF Parameter Class) and

Alg. 1 An algorithm for generating evolution metamodel

Input: DataMM, MOF, OclMetamodel
Output: evolutionMM

```
1   // create empty evolutionMM
2   evolutionMM = new MM( )
3   // create Evolution Class
4   MOFClass Evolution = new MOFClass
5   MOFClass ModelEdit = new MOFClass
6   ...
7   for each element in DataMM do;
8       element = findMOFClass(DataMM, MOF);
9           //generate model edits for metamodel class
10          if (element is instanceOf MOFClass);
11              if (element.isAbstract = false);
12                  //generate addClass edit
13                  MOFOperation addClass = new MOFOperation;
14                  MOFClassProperties = findMOFClassProperties(MOFClass);
15                  for each property in MOFClassProperties do
16                      MOFParameter = new MOFParameter();
17                      MOFParameter = MOFClassProperty;
18                      add MOFParameter to addClass Operation
19                      add addClass to ModelEdit Class
20                  endFor
21                  //generate modifyClass edit
22                  MOFOperation modifyClass = new MOFOperation;
23                  ...
24                  //generate deleteClass edit
25                  MOFOperation deleteClass = new MOFOperation;
26                  ...
27              endIf
28          endIf
29          //generate evolution primitives for metamodel properties
30          if (element is instanceOf MOFProperty);
31          ...
32          endIf
33  endFor
```

appropriately assigns them to the respective model edit operation. The basic idea is that for every evolving (non-abstract) element of the Data Metamodel, we identify its MOF class (line 8). Every property of the identified MOF class as well as properties of its superclass appears as a parameter in the model edit, respecting the parameter data type. For example, `addClass` edit (line 9-20) will have parameters corresponding to the properties of its corresponding MOF Class (also called Class) and its superclasses (e.g. NamedElement), and takes the form : `addClass(name:String, isAbstract:Boolean, superclass:Class)`. Applying the algorithm above, we obtain a MOF-compliant evolution metamodel: an extension of the UML Data Meta-

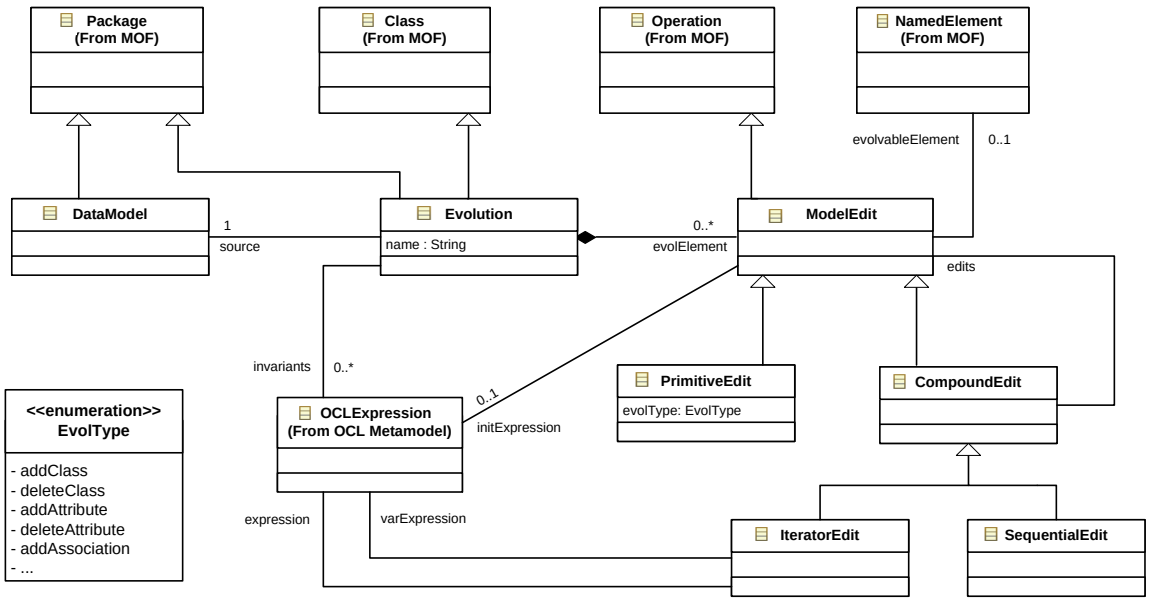


Figure 4.8: Evolution Metamodel

model defined at level (M2) of the four-level MOF architecture and applied to data models at level (M1).

Figure 4.8 shows the main concepts of the generated evolution metamodel. Classes drawn from MOF and OCL are annotated accordingly. Remaining classes represent the main concepts in our proposed evolution metamodel. An Evolution defines how a source data model can be evolved into a target data model. Evolution is a subclass of both Package and Class. As a package it provides a namespace for its ModelEdits and as a class it can define properties and operations. An Evolution can refer to OclExpression to specify invariant properties and WFRs of its instantiating models (i.e. evolution models). Each Evolution contains a set of ModelEdits. A ModelEdit may be subclassed to address different types of changes. These changes could be primitive, primarily affecting one element of a data model, and represented by PrimitiveEvolutionOperations, for example, `addClass()` and `deleteAssociation()` or compound, affecting multiple elements in a data model, and represented by CompoundEvolutionOperations, for example, `extractClass()`. Here we have assumed only that a suitable enumeration can be provided. Each ModelEdit is associated with a set of typed parameters and specify a number of evolution operations. Attribute and Association addition and modification edits take an additional parameter: an OCL expression, to describe the initial value of attribute or association.

With the introduction of the Evolution Metamodel, it is now possible to evolve

Evolution of Data Model Classes

- `addClass (className, isAbstract, superclass)`
- `modifyClass (className, isAbstract, superclass)`
- `deleteClass (className)`

Evolution of Data Model Attributes

- `addAttribute (className, attributeName, expression)`
- `modifyAttribute (className, attributeName, expression)`
- `deleteAttribute (className, attributeName)`

Evolution of Data Model Associations

- `addAssociation (assoName, srcClass, tgtClass srcProperty, tgtProperty, multiplicity, expression)`
- `addAssociation (assoName, srcClass, tgtClass srcProperty, tgtProperty, multiplicity, expression)`
- `deleteAssociation (assoName)`

Figure 4.9: Primitive Model Edits

and modify UML data models using a set of primitive model edits that can be used independently or combined together to describe an intended data model evolution scenario.

4.3.2 Primitive Model Edits

The presentation of model edits in the Evolution Metamodel primarily specifies their abstract syntax. The meaning of each edit is yet to be defined. In the subsequent chapter, we give precise semantics to each model edit. However, here it is important to briefly introduce the intended use of each edit, particularity, given our main focus on data migration, its impact on persistent system data.

Algorithm 1 has generated three primitive model edits corresponding to the addition, modification and deletion of each non-abstract metaclass in our UML Data Metamodel depicted in Figure 4.2. Considering all generated primitive edits would complicate the presentation unnecessarily. In the context of our dissertation, we are mainly concerned with data model changes that affect the form or validity of the corresponding data instances in a system. Therefore, we focus our presentation on edit operations that evolve the main structural elements presented in a model, that is, *classes*, *properties* and *associations*. Subsequently, we disregard changes to less important structural elements such as changes to data types and enumerations. Since data model constraints may need to change as a result of data model evolution, at the

end of next section we discuss the impact of constraint evolution on the existing data, although we do not generate operations corresponding to the evolution of constraints. Figure 4.9 shows the primitive model edits that we shall focus on.

Evolution Primitives for Data Model Classes. At the data model level (M1), adding a class will create a new class, specifying its name, whether it is an abstract class and if it is a subclass to any other class in the data model. At the instance level (M0), existing data are not affected by the class addition, and will continue to be conformant to the target (evolved) model.

Modifying a class so that it is *abstract* requires the designer to provide a subclass to which class objects need to be migrated. Consequently, mandatory attributes that are not available in the subclass would need to be created and initialized to default values. Modifying a class so that it is *concrete* (i.e. can have instances) does not imply changes to data instances. Modifying a class so that it is no longer a *superclass* implies removing slots of properties inherited from that class. Additionally, association ends pointing to the superclass, referring to objects of one of its subclasses, need to be removed.

Deleting data model classes requires deleting corresponding data instances. However, deletion of instance elements poses the risk of migrating to inconsistent state. For example, deletion of objects may leave association links pointing to non-existing objects. Therefore, class deletion is bound to metamodel level constraints: classes may only be deleted when they are outside inheritance hierarchies and are not targeted by associations. Such considerations are discussed further when describing the formal semantics of `addClass` in Section 5.3.1.

Evolution Primitives for Data Model Attributes. Attribute evolution primitives are parameterized by the owning class, the name and type of the new attribute. Furthermore, addition and modification primitives may include an OCL expression describing an initial value that the new or modified attribute can hold. Modifying attribute types require type conversion that maps the original set of values onto a new set of values conforming to the new attribute type. This conversion can be done directly or through an intermediate type. As type conversion is more of an implementation issue, it is not discussed in this thesis. Deleting an attribute, removes the named attribute from its woning class and the corresponding value from class objects.

Evolution Primitives for Data Model Associations. Adding an association requires an association name and two association ends, each owned by classes participating in the association relationship. The addition of an association may require initialization of corresponding links using default values or default value expressions. This can be achieved using an OCL expression, given as a parameter of the primitive.

An association may be modified so that it points to a class which is a subclass or a superclass of the original class. The former case requires deletion of links not conforming to the new association type. The latter case requires no data migration. Modifying an association so that it is an opposite needs a migration to make the link set satisfy this added constraint. Modifying an association so that it no longer has an opposite does not require instance-level migration.

Deleting data model association require deleting corresponding instances, such as links by the migration. However, deletion of links poses the risk of migration to inconsistent models: for example, deletion of a links may break object containment. Therefore, deleting an association is bound to metamodel level constraints: associations may only be deleted when they are neither composite, nor have an opposite.

Evolution of Data Model Constraints. Although many evolutions will involve restructuring and extension of models, many more will involve the introduction of additional constraints, or the modification of constraints already specified within the model —particularly where the evolution is taking place within a model-driven architecture, and the constraints may be used directly to determine the detailed behavior of a system. The simplest, and most common, constraint evolution involves a change to the multiplicity of some property: for example, we might decide that a one-to-many association needs instead to be many-to-many, or vice versa. In UML, this will correspond to a change to the `upper` value associated with one of the properties. More complex constraints may be specified as class or model invariants, describing arbitrary constraints upon the relationships between values of properties across the model.

If the conjunction of constraints after the evolutionary step is logically weaker than before, and the model is otherwise unchanged, then the data migration can be considered feasible. Whatever data the system currently holds, if it is consistent with the original model, then it should also be consistent with the new model. However, where the conjunction of constraints afterwards is stronger than before, or the evolutionary step involves other changes to structures and values, then data may not fit:

that is, the data migration corresponding to the proposed evolution might produce a collection of values that does not properly conform to the new model.

4.3.3 Expressing evolution patterns

While describing compound data model evolution using a set of basic model edits may be achieved by sequencing these edits, pure sequencing is not always powerful enough to express many desired evolution. For example, in generalize class scenario [100], we may wish to create a new superclass AB based on two input classes A and B. Superclass AB is constructed by collecting common attributes that exist in both classes A and B. There may not be any pure sequencing of evolution primitives that could characterize the logic of this scenario. Thus, we recognize the need for an iterator combinator. Hence, in the context of our work, combining edits can be done using one of two combinators :

- Sequential composition : describes the effect of two updates being performed one after the other, as part of the same compound evolution.
- Iterator composition : iterates over a sequence of objects to apply an edit.

With the help of these two combinators, we may compose primitive edits to describe evolution patterns. For example, we can use these combinators to define operations such as `inlineClass` [26] in terms of their component edits on model elements:

```
inlineClass(Source,Target,property) =
 ( forall p : Source.properties .
    addAttribute(Target,p,Source.p.type,
     Source.p.upper, Source.p.lower, Source.p) ) ;
 deleteClass(Source)
```

Here, we have assumed a language in which we may select the values of meta-properties (properties in the metamodel) with the usual `.` notation, as an OCL expression and the existence of an appropriate iterator `forall`. We have then used a sequential comosition operator, denoted by ';', to combine two primitive updates : `addAttribute` and `deleteClass` to define `inlineClass` compound operation.

Other commonly-used patterns include merging classes, introducing an association class and the movement of attributes up and down inheritance hierarchies. A wide range of patterns have been identified in the literature: see for example, [14], [60], and [39].

Example. Within the context of our example outlined at end of Section 4.2.3 and depicted in Figure 4.5, assume in a subsequent, due to change in requirements, it was decided that having a separate PersonalInfo class is unnecessary and that the multiplicity of manager association between Department and Employee classes now needs to be considered mandatory rather than optional. In addition, a new attribute named seniority is added to the Person class and initialized into 'senior' if a person's age is greater than 50, otherwise, the initial value is 'junior'. Suppose also that, to enhance employee assignment tracking capability, a new Project class was extracted from the Department class.

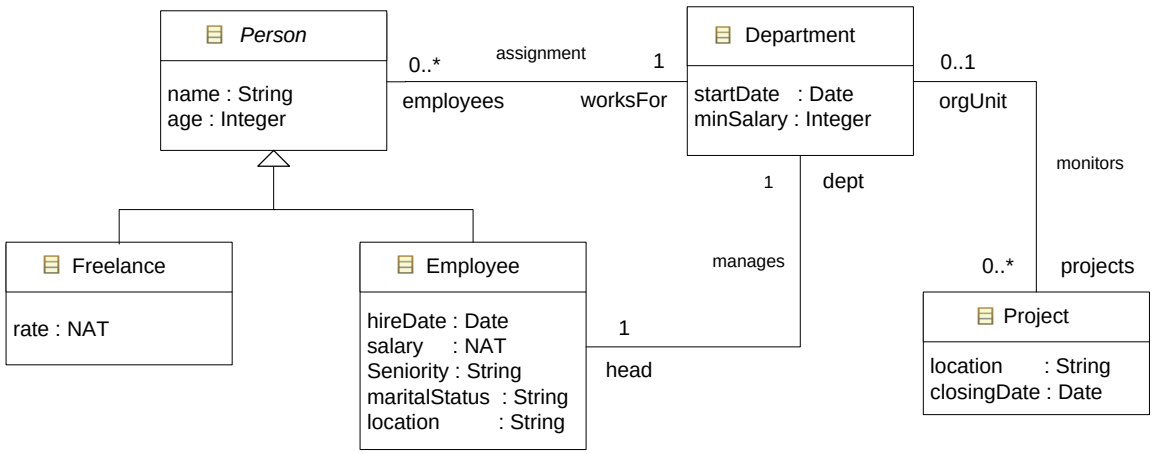
In order to capture these changes at the design level, we may write an evolution model representing the proposed changes as evolutionary steps described in terms of our proposed evolution metamodel.

```
inlineClass(Employee, employee, PersonalInfo);
addAttribute(Employee, seniority: String
   [if self.age > 50 then 'senior' else 'junior']) ;
modifyAssociation(Department, manager, Employee, 1,1) ;
extractClass(Department, Project, projects)
```

Note that each of `inlineClass` and `extractClass` is a compound edit which can be written in terms of the primitive edits. For example, the `extractClass` consists of the following operations, in part:

```
(addClass(Project);
 addAssociation (Department, projects : Project, 0, *);
 moveAttribute(
    addAttribute(Project, location,
          [Project.location = Department.location]);
    deleteAttribute(Project, location));
    moveOperation(
         addOperation(Project, setProjectAssigmt);
         deleteOperation(Department, setProjectAssigmt))
```

Figure 4.10 shows the evolved version of the data model after the above evolution operations have been performed.




```
/* An employee's salary must be greater than the department minSalary*/
context Department inv C1 : self.employees ->forAll(e|e.salary >=
self.minSalary)

/* employees and department are bi-directional associations*/
context Employee inv C2 : self.department.employees ->includes (self)

/* An employee's hire date must be after the department start date*/
context Employee inv C3 : self.department.startDate < self.hireDate
```

Figure 4.10: Employee Information System model (evolved)

4.4 Induced data migration

One important benefit of our proposed modeling approach is that data migration at the instance-level (M0) can be derived by data model changes at the model-level (M1). Modeling each concern at a separate abstraction level allows us to decouple the conceptual representation of the data from the physical representation at the implementation platform.

Figure 4.11 depicts the concept of induced migration, where each data model evolution has a corresponding data migration. Induced data migration consists of three components: (1) a definition of the data model evolution and (2) a mapping from data model evolution to data migration. (3) data migration rules that depends on numerous factors, such as how the data is stored, what platform is available to execute the data transformation and the quantity of the data. This component is therefore driven by the context. The question that remains though is: how to induce data migration from data model evolution?. Below, we characterize this question more precisely. Throughout the coming chapters, we demonstrate how this concept

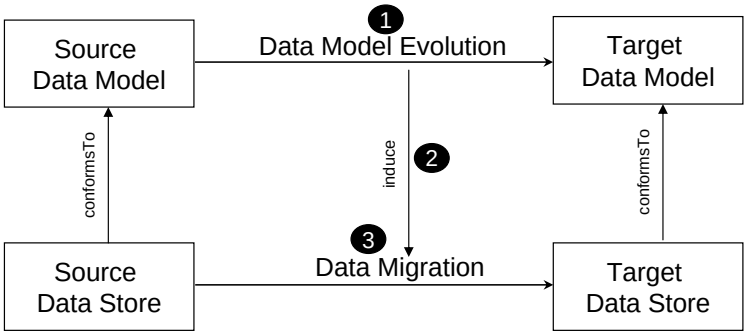


Figure 4.11: Induced Data Migration

can be realized using our model-driven approach.

We specify data model evolution as a collection of model edits. Each model edit specifies a relation between a source model and a target model in the same modeling language: $modelEdit \in Model \nrightarrow Model$, where *Model* represents the set of all data models.

Data migration, on the other hand, can be specified as a collection of data edits. Each data edit specifies a relation between a source data store and a target data store: $dataEdit \in Data \rightarrow Data$, where *Data* represents the set of all data stores.

We may use *modelEdit* as a basis for inducing *dataEdit* functions that, when applied to a data store $d_1 \in Data$ of the existing system model, will produce a target data store $d_2 \in Data$. To map data model evolution at the model level (M1 of MOF hierarchy) to data migration at the data instance level (M0 of MOF hierarchy), we define function *induce*: $induce : (Model \rightarrow Model) \rightarrow (Data \rightarrow Data)$
so that $d_2 = derive(ModelEdit \;\; m_1)(d_1)$

As stated above, instance level actions depend on the implementation platform. For example, if the data we are interested in migrating is persisted in a relational database, we may expect that the instance actions we need to migrate this data would consist of a collection of SQL statements. However, to abstractly demonstrate the concept of induced data migration, in this section we may assume that our instance data can be manipulated by a set of UML instance actions, defined in UML Action package ([120], pp.217).

Table 4.2, shows main actions defined in this package. As the table shows, Objects, attribute value slots and links can be manipulated by actions. Objects can be created and destroyed; attribute value slots can be set and links can be created and destroyed. For example, `CreateObject(c:Class):Instance` creates a new object and `DestroyObject (o:Instance)` destroys an object.

Action	Description
`CreateObject(c:Class):Instance}`	Creates a new object that conforms to the specified class. The object is returned as an output parameter.
`DestroyObject(o:Instance)`	Removes object o from its respective class. We assume that links in which o participates are not automatically destroyed.
`AddAttributeValue(o:Instance, att:Property,v: ValueSpecification)`	Sets the value v as the new value for the attribute att of the specified object o.
`RemoveAttributeValue(o:Instance, att:Property, v: ValueSpecification)`	Removes the value v of att in the specified object o.
`CreateLink(as:Association,p1:Property, o1:Instance, p2:Property,o2:Instance)`	Creates a new link in the specified association as between objects o1 and o2, playing the roles p1 and p2, respectively.
`DestroyLink(as:Association,p1:Property, o1:Instance, p2:Property, o2:Instance)`	Removes the link between objects o1 and o2 from `as`.

Table 4.2: Main Actions of UML Action Package

Each model edit at the model level may be interpreted as a series of instance actions. For example, the `inlineClass` compound edit, shown in Section 4.3.3 could be interpreted as:

```
1 forall object : Target .
2   addAttributeValue (object, sourceProperty, value)
3 forall s2t : sourceTotarget
4    destroyLink (sourceTotarget, sourceProperty, targetObject,
5                 targetProperty, sourceObject)
6 forall sourceObject : Source
7     destroyObject (sourceObject)
```

with `addAttributeValue()`, `destroyLink()` and `destroyobject` instance actions implementing `addAttribute()`, `deleteAssociation()` and `deleteClass` respectively.

Our proposed Evolution Metamodel may be used incorrectly by designer writing evolution models. In particular, a designer may describe evolution operations that violate data model syntactic consistency, semantic consistency or both. Accordingly, our specified model edits and data edits need to handle consistency constraints at two abstraction levels, that is, model-level and instance-level.

We may also note that some evolution operations may have a bearing upon the data integrity semantics. This leads to a non-trivial guard upon the overall data migration function. For example, the `modifyAssociation()` operation in the example at the end of Section 4.3.3 changes the multiplicity of the *manager* association from optional to mandatory. Accordingly, we would expect a non-trivial guard of the form:

```
forall dd : Department .
   dd.manager ->size() > 1
```

to arise upon the migration of existing data. This guard requires that the manager link should exist in all objects of class Department for the migration of the data to succeed. Although this guard is satisfied by Department instance `d2` in Figure 4.5 which has a *manager* link to Employee instance `e3`, Department instance `d1` does not have such a link and may not be migrated to the new model until such link is established. This can be achieved by providing additional input: a set of links or references that are to be created for the association in question or an expression explaining how the link is to be created, in order that the constraints of the new model should be satisfied.

Chapter 5

Specification and Verification of Data Model Evolution

When a data model evolves, one important consideration is to maintain data model consistency. As we discussed in Section 4.2.3, data model consistency can be characterized by a number of consistency constraints. Such constraints define conditions that must be satisfied by a data model and its corresponding data instances.

To be able to maintain the consistency of the data model within the context of evolution, two aspects of semantics need to be elaborated. First, we need to formally capture data model concepts at both levels of abstraction: model-level and instance-level and to precisely define how the two abstraction layers are related. Second, we need to specify the effect of evolution on the data model, showing the impact of evolution operations on model-level elements and on corresponding data instances, ensuring that the data model consistency is preserved.

To satisfy the above requirements, we need to have a formal representation of the data model. This formal representation provides us with a framework for characterizing data model concepts and the relationship that exists among them. It also helps us reason about data model evolution and the effect of such evolution on the consistency of the data model. Defining formal semantic for our data model evolution requires the specification of a semantic domain in a formal language [76]. We selected B-method [6] as a formal framework for the specification of our semantic domain.

Model-based formal methods such as B or Z [164] are well-suited to the expression of models with important static constraints, and to the verification of these constraints across state changes. In our work, we have favored B. As outlined in section 2.4, B-method, is a precise state-based formal modeling language with well-defined semantics and two main notations: AMN and GSL.

In this chapter we exploit B formal specification techniques, expressing both the data model and the evolution model. On one hand, semantics of UML data model (Section 5.1) describes the meaning of a UML data model in terms of the structure of a characterized B abstract machine: sets of elements and their relationships which are consistent with the Well-Formedness Rules (WFRs) defined by the UML abstract syntax. This part of the semantics is given in AMN notation of B method. On the other hand, semantics of model evolution operations (Section 5.3) describes the evolution of the state of the modeled elements and capture the applicability and effect of evolution primitives. This part of the semantics is given in GSL notation of B-method.

5.1 Semantics of UML data model

An appropriate B semantic domain for UML data models should be generic enough to express the structural aspects and properties of UML data models, which we outlined in Section 4.2.1. Since previous work such as [98, 144, 105, 107] has already investigated the mapping from UML to B, we do not investigate this problem. However, we still draw on previous work translating UML and OCL into B to define an appropriate B semantic domain, which is rich enough for our UML data models (instances of the specified UML subset). Below we show parts of the abstract machine specification. Full specification of the abstract machine can be found in Appendix C.

5.1.1 Type structure

UML Boolean type is mapped to `BOOL` (defined as the set TRUE, FALSE in the `Bool_TYPE` machine) in B. UML String type is mapped to `STRING` in B, defined in `String_TYPE` machine. The UML Integer type is translated to B's `NAT`, defined in `Int_TYPE` machine. These machines are system type machines that can be accessed using `SEES` clause in the main machine (lines 2-3, in table 5.1).

The types of other classes in the data model are represented by given sets in the `SETS` clause (lines 4-6), for example, `CLASS`, is a given set used to hold names of possible data model classes and `OBJECTID` is a given set used to define all object identifiers. A B given set may have enumerated values, or, as we use it here, may simply be an (implicitly) non-empty, finite set [[6], Sects. 2.2 and 4.17].

In general, `TYPE` is a given set that represents all types in the data model. It is categorized into `PrimitiveType` and `classType`, represented by two `CONSTANTS`

```
...
SEES
   Bool_TYPE,String_TYPE
SETS
   CLASS; OBJECTID; PROPERTY; ASSOCIATION;
   VALUE; TYPE; PrimitiveTYPE
CONSTANTS
   typeOf, primitiveType, classType
PROPERTIES
   primitiveType : PrimitiveTYPE --> TYPE    &
   classType     : CLASS         --> TYPE    &
   ran(primitiveType) /\ ran(classType) = {} &
   typeOf        : VALUE >-> TYPE
...
```

Table 5.1: Formalization of data model type structure

```
...
className          <: CLASS                    &
isAbstract          : CLASS   +-> BOOL         &
superclass          : CLASS   +-> CLASS        &
ownedProperties     : CLASS   <-> PROPERTY     &
...
```

Table 5.2: Formalization of data model classes

functions. Finally, `typeOf` is an injective function mapping each data model value to its corresponding type.

5.1.2 Data model classes

Table 5.2 shows the formal semantics given to data model classes. A class name is represented by `className` set, defined as a subset (denoted by `<:`) of `CLASS` : a given set that holds all possible class names. `isAbstract` is a function from `CLASS` to a boolean value. Class generalization is represented by the `superclass` function that relates a class to its immediate superclass. A data model class can own a number of properties (attributes and association ends). This is represented by `ownedProperties` relation that maps data model classes to its owned properties.

5.1.3 Data model properties

A UML property may represent an attribute or an association end [124]. Table 5.3 shows the formalization of data model properties. Each property has a name,

```
...
propertyName       <: PROPERTY                &
isComposite         : PROPERTY   +-> BOOL      &
propertyLowerMultip : PROPERTY   +-> NAT       &
propertyUpperMultip : PROPERTY   +-> NAT       &
owningClass         : PROPERTY   +-> CLASS     &
propertyType        : PROPERTY   +-> TYPE      &
opposite            : PROPERTY   +-> PROPERTY  &
...
```

Table 5.3: Formalization of data model properties

represented by `propertyName`, defined as a subset of `PROPERTY`: a given set that holds all possible property names. `owningClass` function maps a property to its owning class. `propertyType` function maps a property to a data type. When a property is an association end, it can represent composition (whole-part) relationship between the property owning class and a related class. This is mapped to `isComposite` function from `PROPERTY` to a boolean value. Lower and upper multiplicity of properties are represented by two functions : `propertyLowerMultip` and `propertyUpperMultip` respectively, mapping a property to a natural number. Function `opposite` maps a `PROPERTY` to a `PROPERTY` to represent the case where two association ends are paired together to form a bi-directional association.

5.1.4 Data model associations

```
...
associationName    <: ASSOCIATION                            &
association         : ASSOCIATION +->(CLASS +-> CLASS)       &
memberEnds          : ASSOCIATION +->(PROPERTY +-> PROPERTY) &
...
```

Table 5.4: Formalization of data model associations

An association can have a name, represented by `associationName`, defined as a subset of `ASSOCIATION`: a given set that holds all possible association names. An association relates two classes, as represented by `association` function. This function is indexed by an association name and yields a functional relation between two classes. Each association must have two association ends. This is represented by `memberEnds` function relating `ASSOCIATION` to a partial function between two properties. Other

characteristics of a member end (as a PROPERTY) is shown in table 5.3 above. For example, the multiplicity of each member end is represented by propertyLowerMultip and propertyUpperMultip.

5.1.5 Data model instances

```
1 ...
2 extent  : CLASS       +-> POW(OBJECTID)                &
3 value   : PROPERTY    <-> (OBJECTID +-> VALUE)         &
4 link    : ASSOCIATION <-> (OBJECTID +-> POW(OBJECTID)) &
5 ...
```

Table 5.5: Formalization of data model instance layer

In our data model semantics definition, we differentiate between two abstraction layers, the model layer and the instance layer. Table 5.5, shows the main elements of the instance layer semantic elements. The function extent maps a class to a set of object identifiers. The relation value returns the current value of an attribute property for each object of the property owning class. The relation link, indexed by an association name, yields a function between object identifiers, represented as a function from an object identifier to a set of object identifiers. Note that here we use relations (denoted by <->) rather than partial functions, to characterize values and links. This is mainly to capture data model inheritance at the instance level. A property can be owned by multiple classes: a superclass and its subclasses in an inheritance hierarchy.

5.2 Consistency of the data model

Our AMN formalization presented above captures the semantics of UML Data Metamodel subset which we use for data modeling, and thus provides the context needed for expressing consistency constraints for data models and, hence, allow us to analyze UML data models for consistency. In the previous chapter, Section 4.2.3, we introduced two kinds of data model consistency constraints dealing with data model consistency from syntactic and semantic views. To be able to verify model consistency checking, here, we formalize these consistency rules based on the abstract machine formalization we presented above. Data model consistency rules are defined as invariants in the abstract machine.

5.2.1 Syntactic consistency

```
1 INVARIANT
2 ...
3   !(c1,c2).(c1:className & c2:className &
4         c1 /= c2 & (c1|->c2) : closure1 (superclass) =>
5         ownedProperties[{c1}] /\ ownedProperties[{c2}] = {}) &
```

Table 5.6: Name uniquness for data model properties

Name uniqueness: for a data model to be valid all data model constructs such classes, properties and associations need to have unique names. This condition is captured by the fact that names of these constructs are defined as sets. However, in presence of data model inheritance, it is necessary to capture the condition that, within an inheritance hierarchy, a property name should uniquely refer to one property only. Where `!` denotes universal quantification, `closure1` denotes transitive closure operator, the invariant shown in table 5.6 expresses this requirement.

```
1 closure1(superclass) /\ id(CLASS) = {}
```

Table 5.7: Absence of circular inheritance

Absence of circular inheritance: this consistency condition requires that no cycles exist within class inheritance hierarchy. This is precisely expressed by the invariant presented in Table 5.7. The transitive closure (denoted by `closure`) of `superclass` function intersected with the identity relation on data model classes (denoted by `id`) should result in an empty set.

```
1 !(aa).(aa:dom(memberEnd) =>
2    !(ae).(ae:memberEnd(aa) & isComposite(ae) = TRUE &
3           association(aa)(owningClass(ae))= {}  =>
4             ownedProperties~[{ae}] = {}  ))
```

Table 5.8: Existential dependency of composition

Existential dependency of composition: in a composite association, the class that represents the *part* can only exist if the class that represents the *whole* exists. This is captured by the invariant in Table 5.8. After universally quantifying over data model associations which are in the domain of `memberEnds` function (line 1), we require

that, if the target class of the composite association got delete, the owning class of the association end should also be deleted.

```
!(aa1,p1,p2).(aa:dom(memberEnds)     &
              p1:dom(memberEnds(aa1) &
              p2:ran(memberEnds(aa1) &
              (p1|->p2) : opposite =>
                 #(aa2).(aa2:dom(memberEnds)) &
                  memberEnds(aa) = {(p2|->p1)})))
```

Table 5.9: Association bidirectionality

Association bidirectionality: two member end properties can be paired so that each end can be an *opposite* of the other. In such a case, both ends need to be owned by the same association. We capture this consistency condition by quantifying over an association and two opposite properties. The invariant requires that, for such an association, there exists another association name with the same member ends, going in the opposite direction.

5.2.2 Semantic Consistency

```
!(cc,oo,pp).(cc:className &
              oo:extent(cc)         &
              pp:PROPERTY           &
              oo : dom (value (pp)) &
              owningClass(pp) = cc
=>
   pp : ownedProperties[(closure1(superclass)\/id(className))[{cc}]])
```

Table 5.10: Instance conformance

Instance conformance: in UML data models, an instance conforms to its class if values of the instance slots are consistent with properties of the instance class and superclasses. This is captured by the invariant in Table 5.10. The invariant requires that properties defining instance values be owned the instance class or its superclasses.

Value conformance: in UML data models, a value conforms to its property if the value is consistent with the data type of the property. This is captured by the invariant in Table 5.11. The invariant requires that the type of any attribute value

```
1 !(cc,aa,oo).(cc:className &
2              aa:ownedProperties(cc) &
3              oo:extent(cc) =>
4                 typeOf(value(aa)[{oo}]) = propertyType(aa) )
```

Table 5.11: Value conformance

assigned to an object of the class to match the type of the property as defined in the object's class (line 4).

```
1 !(aa,pp).(aa:dom(memberEnds) & pp : dom(memberEnds(aa)) =>
2    (!(cc,oo).(cc:dom(association(aa)) & oo:extent(cc)   =>
3       (card(link(aa)(oo))) >= propertyLowerMultip(pp)    &
4       (card(link(aa)(oo))) <= propertyUpperMultip(pp) )))
```

Table 5.12: Link conformance

Link conformance: a link conforms to its association if the objects it references are consistent with the association type and multiplicity. The referenced objects must satisfy the bidirectional association defined at the data model level. The invariant in table 5.12 above requires that the number of objects participating in an association link at the instance level must be permitted by the multiplicity of corresponding member end at the model level.

Example. To show how our proposed abstract machine may be used, we may instantiate its variables from our Employee Information model and its conformant object model in Figure 4.5 and Figure 4.6 respectively, we get the following relations, where `a |-> b` denotes the pair (a, b):

```
className       = {Employee, Department, PersonalFile,... }
ownedProperties = {Employee |->hireDate
                   Employee |->,salary
                   Department |->location ,
                   PersonalFile|->maritalStatus,... }
propertyType    = {hireDate |->Date,
                   salary   |-> NAT,
                   location |->String,...}
association     = {assignment |->(Person|->Department),
                   manages    |->(Department|->Employee),...}
```

```
extent          = {Employee  |->{e100,e200,...}
                   Department|->{d1000, d2000},...}
value           = {name|->(e100|->emp1),
                   rate |-> (f100|->20)
link            = {assignment|->(e100|->{d1000}),
                   manages   |-> (e100 |->{d2000}),
```

Data model-specific constraints (for example, association multiplicity or user-defined constraints) that require particular representation of data instances will also be mapped into invariant properties constraining `extent`, `value`, and `link` variables in our abstract machine semantic domain above.

For example, In Employee Information model, the OCL constraint

```
context Employee inv C1 :
        self.department.employees->includes(self)
```

can be mapped to the AMN machine INVARIANT:

```
! ee : extent (Employee).
     ! dd : link (assignment) (ee) =>
        ee : link (assignment) (dd)
```

where `!` denotes universal quantification, `:` denotes set membership and `()` denotes function application.

5.3 Semantics of model evolution operations

We may give semantics to our evolution operations by mapping each to appropriate substitutions on machine variables, described in B-method *Generalized Substitution Language* (GSL). Each substitution will act upon the variables of the machine state, giving precise meaning to changes in data model elements and corresponding instances. This characterization will help us later reason about data model evolution. By reasoning about the substitution, we may determine whether or not the proposed change corresponds automatically to a successful data model evolution and corresponding data migration, or whether manual intervention will be required.

5.3.1 Specifying evolution primitives

Each of the model evolution operations which we presented in Section 4.3.2 of Chapter 4, may be given semantics as an operation upon the machine which we defined in section 5.1, explaining the applicability and intended effect of an evolution step upon data model elements and data instances contained in the system.

In general, the semantics of our evolution primitive can specified using a combination of preconditioned substitution `PRE..THEN..END` and unbounded choice substitution using `ANY..WHERE..THEN..END`, taking the form:

```
evolutionPrimitive(parameters) =
   PRE
      parameter typing & other B Predicates
   THEN
      ANY  local variable  WHERE
          B Predicates on local variable &
          B Predicates on other variables
      THEN
         localvariable   := B Substitution  ||
         other variables := B Substitution
      END
   END
```

The `PRE` operator in preconditioned substitution is used to assign types to operation parameters and to describe the applicability conditions under which the substitution statements can be executed correctly. Thus, the statement after the `PRE` operator can be seen as expressing the expectations or requirements of the specifier on the conditions under which the substitution will be executed.

The local variable following `ANY` is a variable disjoint from the state space of the operation, created only to execute the statement and discarded after the execution. In general, an `ANY` statement can make use of a list of local variables. The predicate after the `WHERE` clause is a predicate on the local variable(s) that provides a type to the variable(s) and may, in addition, provide some other constraining information. It may also refer to other state variables and relate them to the local variable(s). The body of the substitution comes after `THEN` clause. Here, we describe the effect of the operation. The substitution statements describing the effect of the operation may refer to local variable(s) specified after `ANY` or other machine state variables

```
1  addClass(cName,isAbst,super) =
2      PRE
3          cName  : CLASS - className &
4          cName /: dom(extent)       &
5          isAbst : BOOL              &
6          super  : className
7      THEN
8          className  := className   \/ {cName}             ||
9          isAbstract := isAbstract  \/ {cName |-> isAbst}  ||
10         superclass := superclass  \/ {cName |-> super }  ||
11         extent     := extent      \/ {cName |->{}}
12     END
```

Table 5.13: Part of the semantics of class addition

relevant to the operation. Typically, we use B parallel substitution (denoted by ||), to indicate an undetermined order of execution.

The general form of substitution for *add* operations is a set union (denoted by \/). This is used to add a new element to the set of known elements and to update functional relations in which the type of the element being added takes part. The general form of substitution for *delete* operations is either set subtraction (denoted by -) , domain subtraction(denoted by <<|) or range subtraction (denoted by |>>). These operators are used to remove an element from an existing set and to update functional relations in which the type of the element being removed takes part. The general form of *modify* operation substitution is function over-riding (denoted by <+): the current value of an element is replaced with a new value supplied by operation parameters.

The evolution primitive semantics may refer to either model-level or instance-level concepts. This feature allows us to specify how an evolution of a model-level construct may impact other model-level constructs and /or instance level constructs.

Below we show parts of the semantics of the evolution primitives. The complete description of the semantics of each evolution primitive can be found in Appendix C.

Class addition and deletion

The update `addClass` corresponds to the B-method operation shown in Table 5.13. The `PRE` operator (lines 2-6) provides typing information to the operation parameters. We then select a fresh class name (denoted by `className`)(line 3) to update the variables of the class construct in a series of parallel substitutions (denoted by ||) (lines 8-11). The class variables such as `className`, `isAbstract`, `superclass` are

updated based on the parameters of the operation using set union. Since the addition of a class does not affect existing data instances, at the instance level, we map the class identifier to an empty set of `extent`.

```
1  deleteClass (cName) =
2      PRE
3          cName  : className &
4          cName /: ran(superclass) &
5          ownedProperties[{cName}] /\ ran(opposite) = {}
6      THEN
7          ANY
8              cObjects, cProperties,cAssociations
9          WHERE
10             cObjects = extent(cName) &
11             cProperties = ownedProperties[{cName}] &
12             cAssociations = {asso|asso : ASSOCIATION &
13                 (dom(association(asso))\/
14                     ran(association(asso))) = {cName}}
15         THEN
16             className       := className - {cName}                ||
17             superclass      := {cName} <<| superclass |>> {cName} ||
18             propertyName    := propertyName - cProperties      ||
19             association     := cAssociations <<| association   ||
20             extent          := {cName}       <<| extent        ||
21             value           := cProperties   <<| value         ||
22             link            := cAssociations <<| link
23         END
24     END;
```

Table 5.14: Part of the semantics of class deletion

`deleteClass` operation has the class name `cName` as a parameter. The semantics of this operation is shown, in part, in Table 5.14. The precondition predicate (lines 2-5) ensures that the class name, provided as a parameter of the operation already exists in the data model classes. It also ensures that the deleted class is not one of the superclasses and that none of the deleted class properties is an opposite in a bidirectional association.

The body of the operation first declares three local variables for class objects, class properties and class associations and binds these local variables with their corresponding sets. These local variables are then used to update class variables in the machine such as `className`, `superclass` and `propertyName`...etc. This is done using set and domain subtraction. The remaining substitution statements remove

```
1  addAttribute(cName,attrName,type,exp) =
2      PRE
3          cName    : CLASS                    &
4          attrName : PROPERTY - propertyName  &
5          attrName /:ownedProperties[{cName}] &
6          type      : ran(primitiveType)      &
7          exp       : VALUE                   &
8          typeOf(exp) = type
9      THEN
10         ANY objectId
11         WHERE objectId = extent(cName)
12         THEN
13             propertyName     := propertyName    \/ {attrName}         ||
14             ownedProperties  := ownedProperties \/ {cName|->attrName} ||
15             propertyType     := propertyType    \/ {attrName|-> type} ||
16             value            := value \/  {attrName|->(objectId * {exp})}
17         END
18     END;
```

Table 5.15: Part of the semantics of attribute addition

```
1  deleteAttribute(cName,attrName) =
2      PRE
3          cName : CLASS &
4          attrName : PROPERTY
5      THEN
6          propertyName    := propertyName - {attrName}          ||
7          owningClass     := {attrName} <<| owningClass         ||
8          value           := {attrName} <<| value
9      END;
```

Table 5.16: Part of the semantics of attribute addition

owned class properties from relevant functions. Any objects corresponding to the class name are deleted from `extent`, `value` and `link` functions.

Attribute addition and deletion

The operation of `addAttribute` has the semantics shown in Table 5.15. The precondition of the operation defines the relevant type of each operation parameters (lines 3-9). The body of the operation consists of a series of parallel substitutions, creating a new attribute and setting up its `propertyName`, `propertyType` and *default* value based on expression provided by the operation parameters (lines 14-17).

```
addAssociation(assoName,
               srcClass,srcProp,tgtClass,tgtProp,isComp,exp) =
    PRE
        assoName: ASSOCIATION - associationName &
        srcClass : CLASS & srcClass /: dom(association(assoName)) &
        tgtClass : CLASS & tgtClass /: ran(association(assoName)) &
        srcProp  : PROPERTY - propertyName &
        srcProp /:ownedProperties[{srcClass}]&
        exp      : OBJECTID
    THEN
        ANY srcOID
        WHERE srcOID = extent(owningClass(srcProp))
        THEN
            associationName := associationName \/ {assoName}            ||
            propertyName    := propertyName    \/  {srcProp,tgtProp}    ||
            association     := association     \/
                                {assoName|->{srcClass|->tgtClass}}      ||
            memberEnds      := memberEnds      \/
                                {assoName|->{srcProp|-> tgtProp}}       ||
            ownedProperties := ownedProperties \/
                               {srcClass|->srcProp, tgtClass|->tgtProp} ||
            link            := link   \/ {assoName |-> (srcOID * {{exp}})}
        END
    END;
```

Table 5.17: Semantics of association addition

The semantics of `deleteAttribute` operation is shown in Table 5.16. This operation takes two parameters, `attrName`: the name of the attribute to be deleted and `cName`: the name of the attribute owning class. Both parameters are bound to corresponding attribute and class names. A parallel composition of substitution statements are then used to remove the attribute from relevant sets and relations using set and domain subtraction operators (lines 6-8).

Association addition and deletion

The operation of `addAssociation` has the description presented in Table 5.17. In addition to an association name, the operation takes a source and a target class names and a source and a target property names as two association ends to form an association. The parameters of the operation also specify whether any of the two ends is considered as a composite end of the association. Similar to attribute addition, the

```
deleteAssociation(assoName) =
   PRE
       assoName : ASSOCIATION
   THEN
       ANY
           srcClass,tgtClass, srcProp,tgtProp
       WHERE
           srcClass : dom(association(assoName))&
           srcProp  : dom(memberEnds(assoName)) &
       THEN
           associationName := associationName - {assoName}      ||
           propertyName    := propertyName - {srcProp,tgtProp}  ||
           association     := {assoName} <<| association        ||
           memberEnds      := {assoName} <<| memberEnds         ||
           ownedProperties := ownedProperties  -
                                {srcClass|->srcProp, tgtClass|->tgtProp} ||
           link            := {assoName} <<| link
       END
   END
END
```

Table 5.18: Part of the semantics of association deletion

parameters of `addAssociation` operation may include an expression giving a default value to the association being added.

The body of the operation binds a local variable of source object identifier to corresponding identifiers. This variable is used to initialize the link created at the instance-level when an expression is provided in the operation parameters. The substitution statements update relevant functional relations such as `associatioName`, `association` and `memberEnds` using set union. Note here that we assume that the source and target properties provided as parameters in the operation did not exist before in the model. Hence, we use a number of substitution statements to introduce them into the data model.

The operation of `deleteAssociation`, shown in Table 5.18, takes the association names as a single parameter. The `PRE` operator ensures that the association name is a member of the `ASSOCIATION` set. The body of the operation first introduces a number of local variables to identify source and target classes and association ends participating in the association to be deleted. These variables are then used to update association-related functions such as `association` and `memberEnds`using a combination of substitution statements.

5.3.2 From evolution primitives to evolution patterns

We may wish also to include model edits corresponding to more complex model evolution operations such as inlining an existing class, introducing an association class, extracting a class or specializing a reference type. These operations involve changes to a collection of related model elements.

Fowler's refactoring catalogue [62] and advanced schema evolution operations , similar to those outlined in [100, 39, 26], or more recently in [19] can be a starting point for candidate patterns relevant within the context of model-driven information systems evolution. Using our AMN representation, these data model evolution patterns may be given a formal semantics to precisely define their applicability and effect on an information system data model.

For example, we may introduce `inlineClass` compound operation. This operation [39] is used when a referenced class becomes redundant and we wish to retain some or all of its attributes as members of another class. For the arguments `srcClass`, `refClass`, and `srcAttribute`, this operation can have the semantics shown in Table 5.19.

```
1  inlineClass(srcClass, refClass, refProperty) =
2  PRE
3     srcClass    : className &
4     refClass    : className &
5     refProperty : propertyName &
6     refProperty : ownedProperties[{srcClass}] &
7     propertyType(refProperty) = classType(refClass)
8  THEN
9     ! attrib : ownedProperties [{refClass}] .
10        {addAttribute(srcClass, attrrib, type, exp);
11         deleteAttribute(refClass, attrrib)};
12      VAR assoName IN
13          assoName := dom(association~[{(srcClass|->refClass)}];
14      deleteAssociation(assoName);
15      deleteClass(refClass)
16 END
```

Table 5.19: Semantics of class inlining

As another example, we take `generalizeClass` compound operation [100]. This operation finds common attributes between two given classes and builds a new superclass. The semantics of this operation is defined in Table 5.20.

The precondition of this operation ensures that both subclasses exist in the data model. In the substitution of the operation, we first create an abstract superclass. We then define three local variables : to collect the attributes of each subclass and the common attributes that exist in both classes. We finally assign each common attribute to the new superclass and modify each subclass to point to the new class as its super.

```
generalize (subClass1, subClass2, superClass) =
PRE
   subClass1 : className &
   subClass1:  className &
   superClass: CLASS     &
THEN
   addClass (superClass, isAbstract) ;
      VAR newSuper, sc1Attribs, sc2Attribs, commonAttribs IN
         sc1Attribs := union (ownedProperties(subClass1)) &
         sc2Attribs := union (ownedProperties(subClass2)) &
         commonAttribs = sc1Attribs /\ sc2Attribs
      THEN
         ! attrib . (attrib : commonAttribs .
            ! oClass . (oClass = owningClass(attrib)
               ownedProperties := ownedProperties - {oClass |-> {attrib}}
                                                  \/ { superClass |-> {attrib}}
               owningClass (attrib) := superClass));
           modifyClass(subClass1, newSuper) ;
           modifyClass(subClass2, newSuper) ;
      END
END
```

Table 5.20: Semantics of class generalization

In the following section, we use the formal semantics we proposed here to show how a calculation of consistency constraints can be used to determine the domain of applicability of a proposed sequence of model changes: the set of data states from which a successful data migration will be possible—from the old version to the new version of the system. As the system model is edited and annotated to reflect proposed changes, the designer can be given an indication as to whether the data held in the existing system, transformed according to the specified evolution, would fit the updated model.

5.4 Verification of data model evolution

As we outlined in Chapter 2, within the context of this dissertation, the problem of model-driven data migration can be decomposed into two sub-problems: (i) specifying data model evolution and (ii) managing the effect of evolution.

The first sub-problem deals with the meaning of each evolutionary step on the overall data model and underlying data instances. For example, the deletion of a property in a class affects the owning class and its subclasses inheriting that property. This deletion also implies deleting corresponding data values from objects of the affected classes. In the previous section, we have assigned our evolutionary steps a precise semantics from B-method that deals with this sub-problem.

The second sub-problem requires techniques for maintaining both data and model consistency. As information systems evolve, it is important to be able to determine whether the proposed evolution preserves data model consistency and whether the corresponding data migration is going to succeed. In other words, we need to ensure that each model-level change and corresponding data-level values satisfy the invariants of the target model, hence we can determine the success of the data migration before it is executed.

Building on AMN formalization that we presented in the previous section, in this section we present three important verification activities. First, using B-method internal consistency proofs, we show how our model edits can preserve data model consistency. In our context, this is a pre-requisite for evolving data models, i.e. we need to ensure that by applying our model edits on a consistent source data model, we obtain a consistent target data model. Second, as another verification activity, using B-method refinement proofs, we show how it is possible to check whether an evolved data model is a refactoring of a source data model. Such a check would allow modelers to perform a wide range of edits and, at the same time, ensure that data migration can be performed successfully, given that the resulting model is a refinement of the source model. Finally, when data model evolution involves adding or removing data model features such that refinement cannot be established, we show applicability of data model evolution may be determined and use this as an indication for the success of data migration from the old version to the new version of the system.

5.4.1 Verifying data model consistency

The AMN formalization we presented so far can play a substantial role in verifying data model consistency, both syntactic and semantic. Hence, we are able to verify

the notion of conformance; a crucial principal in model-driven development. While most of the related work in this area, such as [128, 54, 25], focus on one aspect of conformance, that is, whether a model conforms to a metamodel (M1 to M2 levels of MOF hierarchy), we, in addition, are also interested in investigating another aspect of conformance, which is verifying that data instances conform to the data model (M0 to M1). In our work, checking both aspects of conformance resolves to discharging an internal consistency proof obligation in B-method. Below we explain the main principles based on which data model consistency proof obligations are raised. In Appendix D, we provide detailed explanation of how these proof obligations are discharged using B-method proof tool.

Based on our explanation in Section 4.2.3, we assume that all data model consistency constraints are properly mapped into abstract machine invariants, here denoted by INV. Abstract machine INV specifies what is required to be true of all possible states that can be reached during any model edit. We also assume that all data model edits are properly mapped into abstract machine operations, here denoted by $EDIT$. If invoked correctly (i.e. within their precondition), each model edit $EDIT$ must maintain machine invariants, and reach a final state in which INV holds. If we further assume that that an edit can only be called from a state in which the INV is true, the proof obligation on model edits to preserve the invariant is thus as follows:

$$INV \wedge P \Rightarrow [EDIT]INV \tag{5.1}$$

This states that if the machine is in a state in which INV and PRE are true, and the operation is supplied with inputs in accordance with PRE, then its behavior (described by $EDIT$) must be guaranteed to re-establish INV. If all of our model edits meet this requirement, and the machine starts in a state in which INV is true, then the invariant INV must indeed be true in every reachable state.

To establish that machine invariants hold when model edits are executed, we need to prove that

$$[EDIT](INV) \tag{5.2}$$

As the abstract machine includes a number of invariants given as conjunction, according to B-method proof generation technique, we need to establish that

$$[EDIT](INV_1) \wedge \cdots \wedge (INV_n) \tag{5.3}$$

This is equivalent to establishing each of the invariants separately and it allows us to give smaller proof obligations rather than one large proof obligation. In fact, only

	Syntactic Consistency				Semantics Consistency		
	Name Uniq.	Inher. Heirchy.	Bidirectionality	Exis. Depndncy.	Inst. Conform.	Value Conform.	Link Conform.
Class addition	x	x					
Class deletion		x	x	x	x		
Attribute addition	x	x				x	
Attribute deletion		x					
Association addition	x	x	x				x
Association deletion		x	x	x	x		

Table 5.21: Potential violations of machine invariants by model edits

the invariants that include any of the variables that *EDIT* acts on as a free variable will raise proof obligations, i.e. if i is a free variable in INV_1 and it is being updated by *EDIT*, we will have to prove that $[EDIT]\ (INV_1)$. For the rest of the invariants INV (excluding INV_1), we have that $[EDIT]\ (INV) = INV$; since there is no free occurrence of i that *EDIT* substitutes.

Table 5.21 shows a high-level overview of the proof obligations raised by B-method prover for our model edits. Columns correspond to data model consistency constraints, both syntactic and semantic; rows to our generated model edits. A cross in a cell represents that there is a potential conflict between the corresponding consistency invariant and the specific edit, meaning that the consistency constraint can potentially be violated when the edit executes.

An operation whose proof obligation is not true highlights possible conflict between the machine invariant and the operation, which will need to be resolved during the design process. There are a number of ways that the conflict can be resolved, depending how it arose. It may be that the machine allows the operation to be invoked when it should not be. This is controlled by the precondition to the operation, and it can be corrected by strengthening the precondition to exclude the states and inputs where it should not be invoked. Consider an operation whose body is `PRE P THEN EDIT END` in a machine whose invariant is INV. The proof obligation for the operation is:

$$\begin{aligned} INV \wedge P &\Rightarrow [EDIT]INV \equiv \\ P \wedge INV &\Rightarrow [EDIT]INV \equiv \\ P &\Rightarrow (INV[EDIT]INV) \end{aligned}$$

$INV[EDIT]INV$ represents the weakest value of P that will ensure that $EDIT$ will preserve invariant INV. This gives us a justification to use proof obligations to complete preconditions.

For example, since the typing invariants of variables representing model elements is denoted by partial functions, in order to ensure that such typing invariants are maintained after the execution of relevant model edits, we identify a precondition, which states that if a new model element is added by the operation, this element should not be in the domain of the partial function. For instance, `owningClass`, which is a partial function from `PROPERTY` to `CLASS` is updated by `addAttribute` operation. In `addAttribute` operation, we identify `attrName /: dom(owningClass)` as a precondition to be added to the operation.

The fact that our model edits are proved to preserve data model consistency constraints gives us a guarantee that when we use these edits to evolve a consistent source data model, we will end up with a consistent target data model. With this guarantee, we proceed to another kind of check involving source and target data models, that is to check whether the target model is a refinement of the source model.

5.4.2 Verification of data model refactoring

Our focus is on refactoring UML data models used within the context of information systems development. We have shown how a refactoring step such as `extractClass` can be applied to an information system data model.

The essence of refactoring is to improve the design of a software artifact without changing its externally observable behavior [111]. Although the notion of refactoring has been widely investigated, for example [62], a precise definition of behavior preservation is rarely provided [111].

The refinement proof in B establishes that all invariant properties of a current data model and that all pre-post properties of model operations are also valid in the target (evolved) model. Thus, it can give us a means to verify the data model refinement property and, at the same time, guarantee that the data migration will succeed: that is, the data values after the changes will satisfy the target model constraints. This enables designers to perform a wide range of data model changes (e.g. those characterized as refactoring) while ensuring that data persisted under a current model can be safely migrated.

In Section 3.3.2, we mentioned that a refactoring should not alter the behavior of software. As we are interested in applying the refactoring notion to data models as software artifacts, we need to capture the behavior aspect of data models and

prove that the source model behavior (before refactoring) is semantically equivalent to the target model behavior(after refactoring). This can be achieved by translating the source model to an AMN *abstract* machine and the target model to an AMN *refinement* machine. We use a *linking invariant* in the refinement machine to relate the state space of the two machines. By discharging B-method refinement proof obligations we outlined in section 2.4, we can prove that the target model is a refinement of the source model and hence establish that the behavior of the two models is equivalent.

The mapping from UML to B described in the previous chapter can be used for such verification. We map the current data model to an AMN abstract machine and the target model to an AMN refinement machine. In the refinement machine, we need to define a *linking invariant*: an invariant property that relates data in both machines and acts as data transformation specification. This invariant can be mapped from initial value expressions provided by designer in the evolution specifications (e.g., those provided with `addAttribute()` and `addAssociation()` primitives).

By discharging B refinement proof obligations we can prove that a target model, after evolution, is a data refinement of the current model and, hence, establish that the behavior of the two models is equivalent and that the data migration can be applied. Below, we state the proof obligations in general then apply them to an example.

Example. Figure 5.1(a) shows a partial mapping of the initial version of UML data model, presented in Figure 4.6(a), to an AMN abstract machine. Figure 5.1(b) shows a similar mapping of the same data model after applying `extractClass()` refactoring step, as shown in Figure 4.10, to an AMN refinement machine. Note that the last invariant conjunct in the refinement machine (line 9 in Figure 5.1(b)) describes a linking invariant in the form of data transformation of the *team* association. While in the source model, this association was a relation between Department and Employee classes, in the target model it is a *relational composition* through Project class. Applying refinement proof obligations to the example above, we get the following outcome:

1. **Initialization**. This condition holds since the empty sets in Initialization clauses can be related to each other.

2. **Operations**. When considering `setProjAssgmt()` operation (line 16 in Figure 5.1(b)), this condition holds since this operation in the refinement machine has no explicit precondition (it works under the assumption of the precondition of

```
1  MACHINE   EmployeeTracking
2
3  SETS      EMPLOYEE; DEPARTMENT
4  VARIABLES employees, team
5  INVARIANT
6     ...
7     employees : DEPARTMENT <-> EMPLOYEE &
8     team      : DEPARTMENT <-> EMPLOYEE
9     ...
10
11 INITIALISATION
12    employees := {} || team := {}
13
14 OPERATIONS
15    ...
16    setProjAssgmt (emp,dep)=
17      PRE
18         emp: EMPLOYEE & dep: DEPARTMENT
19      THEN
20         team := team \/ {dep |-> emp} END;
21
22    response<--getProjAssgmt(emp)=
23      PRE
24         emp: ran (team)
25      THEN
26         response:= team~[{emp}] END
27 END
```

(a)

```
Refinement EmployeeTrackingR
REFINES    EmployeeTracking
SETS       PROJECT
VARIABLES  employeesr, projectsr, teamr
INVARIANT
   ...
   teamr     : PROJECT <--> EMPLOYEE &
   projectsr : DEPARTMENT <-> PROJECT &
   team =    (projectsr ; teamr)
   ...
INITIALISATION
    employeesr :={} || projectsr := {}||
    teamr := {}
OPERATIONS
   ...
   setProjAssgmt (emp,dep)=
      BEGIN
         ANY  pp WHERE pp:projectsr[{dep}]
         THEN teamr := teamr \/ {pp|->emp} END
      END;

   response<--getProjAssgmt(emp)=
      BEGIN
      VAR ppr
      IN ppr:= teamr~[{emp}];
         resp:= projectsr~[ppr]END
      END
```

(b)

Figure 5.1: Mapping of data models in Figure 4.6(a) and Figure 4.10 to AMN (partial)

corresponding `setProjAssgmt()` operation in the abstract machine) and every execution of this operation in the refinement machine [S1] updates `teamr` relation which is (according to linking invariant J) an equivalent relation to `team` which is updated by `setProjAssgmt()` operation in the abstract machine (line 16 in Figure 5.1(a)).

3. **Operations with outputs**. Applying this proof obligation on `getProjAssgmt()` operation (line 22 in Figure 5.1(b)), we find that it holds. Every execution of this operation in the refinement machine generates a `response` output assigned to an input employee. This is matched by the execution of the corresponding operation in the abstract machine via the linking invariant `team = (projectsr ; teamr)`.

We conclude that a refinement relation exists between the two machines.

We have shown how an information system data model may be evolved by introducing a refactoring steps. The applicability and effect of such refactoring steps were precisely specified using our formal semantics. This precise definition allowed us to apply refinement checks to establish whether a target data model is a refinement of a source data model.

5.4.3 Checking applicability of data migration

Where the evolution represents a change in requirements, we should not expect the new model to preserve the behavior or establish the invariant properties of the current model; to the contrary, we should expect to find that different properties are now constrained in different ways.

In this section, we present a technique for computing the pre- and postconditions of a collection of evolution operations. This technique is important for two main reasons. First, it allows us to specify an evolution pattern to an existing data model design as a composition of evolution primitives and then to check the legality of the composition and calculate its overall precondition. Second, since we aim to check data model evolution statically, this technique will help us decide, in advance, if it is meaningful or safe to perform a sequence of evolution or not and, as a result, avoid invoking a kind of roll-back mechanism to restore the system to a previous state when one evolution operation cannot be executed. We start from the basic assumption that it is easier to check if a sequence of model evolution and corresponding data migration can be successfully applied on a given data model before execution and that such check can be simpler if we can represent such composed evolution by one pre-post pair.

As we stated previously, in our work, we have two main ways in which we can compose evolution:

1. Grouping or combining (using sequential compositions).

2. Set iteration.

Chaining using a *sequential* composition (denoted by `;`) is where a sequence of evolution primitives is applied one primitive after the other. For example, the following chain adds attributes `attr1` and `attr2` to the class `E`.

```
addAttribute(E,attr1);
addAttribute(E,attr2)
```

Set iteration is where an evolution or an evolution chain is performed on a set of model elements. For example, the following set iteration copies all the attributes of the class `E` to the class `D`.

```
forAll attr:Attribute & owningClass(attr) = E  . {
   addAttribute(D,attr)}
```

A chain of evolution primitives may be of any length, but we can simplify the computation of its pre- and postconditions by observing that we need only to solve the problem for a chain of a length of *two* primitives. This procedure can then be iteratively applied to the remaining chain until the full pre- and postconditions have been computed.

Below we present how an overall pre- and postcondition of a chain of evolution may be calculated in three steps. First, we need to establish that the chain of evolution primitives is legal, i.e. there is no logical contradiction between one evolution primitive and the other. Second, since an evolution chain may contain a set iteration, we explain how the pre- and postconditions of a set iteration can be computed . Finally, we perform the computation of the overall chain of evolution.

Step 1 : determining the 'legality' of a compound evolution

To provide more precise description of evolution composition, we introduce the following notation:
$state_i$ - set of all variables in a particular system state i.
pre primv ($state_i$) - precondition of an evolution primitive evaluated on state i.
post primv ($state_i$) - postcondition of an evolution primitive evaluated on state i.

Initially, one might think that two evolution primitives can be composed simply by combining their pre and postconditions using the AND operator. In some cases this is correct. In some other cases, combining evolution primitives may introduce logical contradictions that may not seem obvious. for example,

```
addClass(F);
addAttribute(F,attr)
```

Considering the semantics of `addClass` and `addAttribute` primitives we discussed in Section 5.3, ANDing the preconditions of these two primitives produces, among other precondition clauses:

```
F  : CLASS - className  & ...
F  : className & ...
```

even though this chain seems perfectly correct. The source of this contradiction lies in the fact that the two preconditions should be valid at different points in data model evolution.

In other compositions, the chain may simply be illegal, although the overall precondition may not indicate contradiction e.g.,

```
deleteClass(C);
addAttribute(C,attr)
```

ANDing the preconditions here gives `C : className` even though this chain is illegal. Although the precondition for `addAttribute` is valid at the start of the chain, `deleteClass` breaks it, so this composition of evolution primitives can not be correct. Therefore, for our chain of evolution to be legal, we require that every precondition should be valid at the point where it is applied.
If *eval* is a function of type *eval* : PREDICATE → BOOLEAN, we may write :
$eval\ [pre\ primv_i\ (state_i)] = true$
If this condition is not satisfied then we have one inapplicable evolution step and we can conclude that the whole chain is inapplicable.

The second condition that must be satisfied is that if the postcondition of a primitive description evaluates to true then the precondition of a subsequent evolution primitive must also evaluate to true, i.e.:
$eval\ [(post\ primv_i(state_i, state_{i+1}) \Rightarrow pre\ primv_{i+1}\ (state_{i+1})] = \text{true}$
Therefore, before we continue with composing evolution primitives, assuming a chain of two primitives, we need to analyze the consistency between postcondition of the first and precondition of the second evolution primitives. With this analysis we detect cases in which, after completion of the first evolution primitive, we cannot continue with the second evolution primitive due to unsatisfied precondition.

Step 2 : computing pre- and postconditions for set iteration

As stated earlier, a set iteration has the following form:

```
forAll x:Element & Pred(x,...) . {
   primitive (x,. . . )}
```

where `Element` is some kind of model elements, `Pred` is some predicate with an argument from the model elements or the evolution. If the set of `x` of type `Element` that satisfies `Pred(x,...)` is given as `{x1,x2,...,xn}`, and writing $primv_k$ as a shorthand for $primitive(x_k, ...)$, then this iteration may be viewed as the following chain: $primv_1, primiv_2, ..., primv_k$.

To be able to calculate the overall pre and postcondition, we must first establish whether such an iteration is legal, i.e. no contradiction exists between the postcondition of any evolution primitive and the precondition of a subsequent evolution primitive within the set iteration. This can be done using the same criteria we presented in step 1 above.

Once the legality of set iteration is established, we can compute its precondition. This can be done by logically ANDing the preconditions of its primitives

Now that we are able to determine the legality of a chain of evolution and express the pre and postcondition of a set iteration, we are in a position to compute the overall pre and postcondition of the chain.

Step 3: Computing overall pre- and postcondition

Assuming the chain is legal, its precondition is obtained by logically ANDing *pre primv*1 with whatever parts of *pre primv*2 that are not contradicted by *post primv*1. If a contradiction arises in this evaluation, the chain is illegal. This results in aborting the calculation. Once we establish that no logical contradiction exists, we can compute the overall precondition of the chain. This is obtained by evaluating:
*pre primv*1 $\wedge$ (*post primv*1, *pre primv*2) $\wedge$ (*post primv*2, *pre primv*3)$\wedge$ ...

In our approach a postcondition is described as substitution statements. An expression `E` can be substituted for a free variable `x` (i.e. one that is not in scope of quantification) in a predicate `P`, by replacing all free occurrences of x with the expression `E`, i.e. `P[E/x]`. As well as single substitutions that replace a single variable in a predicate with an expression, we also use multiple substitutions. These simultaneously replace a collection of variables with a corresponding collection of expressions, i.e. `P[E,...,F / x,...,y]`. Any machine variable not mentioned in the postcondition of an evolution primitive is implicitly not affected by the evolution. The postcondition of a compound evolution is obtained by considering the overall variable substitution described in the postconditions.

An example of the application of the technique we described above is given next.

Example

In this example we take a typical compound evolution that involves both chaining and set iterations and compute its pre- and postconditions. We will use `inlineClass` operation which was part of the data model evolution in our example at the end of Section 4.3.3. Figure 5.2 depicts the overall effect of this operation. The semantics of this operation is defined, in part, as:

```
inlineClass(srcClass, refClass, refProperty) =
<forAll attrib : ownedProperties [{refClass}] .
     addAttribute(srcClass, attrrib,...);
     deleteAttribute(refClass, attrrib)}
```

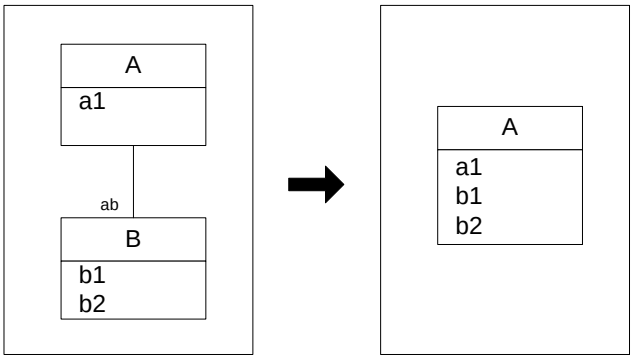


Figure 5.2: Class inlining

```
deleteAssociation(assoName);
deleteClass(refClass)>
...
>
```

Computing the pre-and postconditions of this compound evolution proceeds in several steps:

1. Computing pre and postcondition for the set iteration

This involves first rewriting the precondition of `deleteAttribute` operation with the postcondition of `addAttribute`:
evaluate([*post addAttribute*], [*pre deleteAttribute*]) =

```
attrib : ownedProperties[{refClass}] & ...
```

and then ANDing this with the precondition for `addAttribute()`, which is:

```
attrib /: ownedProperties[{srcClass}]
```

so the final precondition for this chain is, in part,:

```
attrib : ownedProperties[{refClass}] & ...
attrib /: ownedProperties[{srcClass}]
```

which causes no contradiction so the chain is legal within the set iteration. Therefore, on every iteration, the precondition must be true, i.e. must be valid for every attribute owned by class `refClass`. The postcondition of the set iteration body can be computed by concatenating the substitution statements of `deleteAttribute` and `addAttribute`, in part:

```
ownedProperties := ownedProperties \/ { srcClass |-> attrib } ||
ownedProperties := ownedProperties -  { refClass |-> attrib }
...
```

The iteration creates a new `attrib` in class `srcClass` each time and deletes the same `attrib` from class `refClass`.

2. Computing pre and postconditions for `deleteAssociation()` primitive

Here, we need to rewrite the preconditions for the `deleteAssociation()` with the postconditions of the set iteration in step 1. However, since the postcondition of the set iteration updates relevant machine variables for the attribute `attrib`, while the precondition of `deleteAssociation()` checks, among other things, whether the deleted association already exists in the data model, there is no relation between the two and the re-writing required by this step is avoided.

3. Computing pre and postconditions for `deleteClass()` primitive

The precondition of `deleteClass()` is specified as:

```
refClass : className
```

the postcondition of `deleteClass()` is specified, in part, as:

```
 className := className - {refClass}                    ||
 superclass := {refClass} <<| superclass |>> {refClass} ||
 ownedProperties := {refClass} <<| ownedProperties      ||
...
```

4. Computing pre and post for the overall chain

Precondition of `deleteClass` must be rewritten with the postcondition of `deleteAssociation()` and if there are remaining conjuncts, these must be part of the precondition of the whole compound evolution. the precondition of `deleteAssociation` obviously constraints the `association`, `memberEnds` and other association-related functions and, within the context of this example, may not have direct relation to variables updated by `deleteClass`. Therefore, the precondition remains the same and does not need to be re-written. The overall postcondition of the chain can be obtained by combining the postconditions of the set iteration, `deleteAssociation` and `deleteClass` as elaborated above. Accordingly, we conclude that the evolution proposed of `inlineClass` operation is applicable.

In other cases, the evolution proposed by class inlining may not be applicable. For example, if we suppose that the `refClass` (i.e. the class to be inlined) owned a property that is an end of a bi-directional association, we would not expect the following precondition of `deleteClass` operation to be satisfied:

```
ownedProperties[{refClass}] /\ ran(opposite) = {}
```

That is, to delete `refClass`, we need to ensure that none of its owned properties participates in an opposite relation.

Chapter 6

Generating Platform-Specific Data Migration Programs

In previous chapters, we have presented our approach which combines UML with B-method to facilitate the task of data model evolution. The result is a precise abstract description of information system changes. This abstract description can be used to reason about data model evolution and to induce corresponding data migration, moving data instances from one information system version to the other.

To generate an appropriate implementation of our data model evolution specifications, we require a mapping from our abstract model operations to operations upon a specific, concrete platform: some of these operations will update metadata, or features of the representation; others will be data operations implementing the semantics outlined above in AMN. In practice, the kind of data that we might wish to preserve across successive versions of an information system is likely to be stored in a relational database.

The goal of this chapter is to show how the abstract specifications of data model structural properties and data model changes can be transformed into a relational model then into an executable implementation. We have defined a set of formal refinement rules that deal with different concepts such as inheritance and associations. The illustration of these rules will be covered in the first part of the chapter. In the second part of this chapter, we will cover a subsequent refinement to an SQL implementation. In both refinements we cover two sets of refinement rules. The first set deals with the refinement of data structure : introducing more concrete variables or changing the type of existing abstract variables. The second set deals with substitution refinement : rewriting the description of evolution operations in terms of the more concrete variables introduced on data structure refinement.

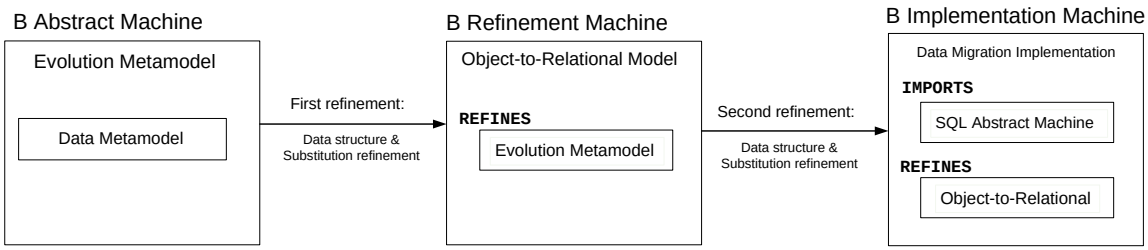


Figure 6.1: From abstract specifications to relational database implementation

Figure 6.1 shows the main elements of our B specifications. The abstract machine for Evolution Metamodel has been presented in Chapter 5. This abstract machine has been refined into more concrete specifications in the form of Object-to-Relational refinement machine, which will be presented at the first part of this chapter. In the second part of this chapter, we introduce the last component : Data Migration Implementation, which imports an abstract machine representing the metamodel of SQL, the language that we are using to implement our data migration programs.

Each of the above refinement steps gives rise to proof obligations. While specification proofs, as we have outlined in Chapter 5, ensure that operations preserve the invariant, refinement proofs ensure the correctness of a refined component with respect to its abstract specifications.

6.1 From an object data model to a relational data model

The increasing popularity of using object models in data modeling [138] and the prevalance of relational model for data persistence created the need for addressing object to relational mapping techniques. Using similar techniques to those reported in [10] and [20], we add two contributions to previous work in this area. First, our mapping rules are formally characterized in B-method, a language with theorem-proving tool, which enables us to verify the correctness of the refinement process. Second, unlike [106] and [110] which have used AMN to describe refinement within the context of information systems development, our main focus here is on describing such rules within the context of information systems evolution and data migration. The main difference of our refinement rules originates from the need to refine model evolution operations into corresponding and faithful relational representation then into SQL executable programs.

Refinement Variable	Related Abstract Variable	Description of Linking Invariant
ownedAttributes, ownedAssoEnds	ownedProperties	ownedProperties abstract variable is refined into two refinement variables : ownedAttributes and ownedAssoEnds, defined as disjoint subsets of ownedProperties.

```
1    ownedAttributes <: ownedProperties &
2    ownedAssoEnds   <: ownedProperties &
3    ownedAttributes \/ ownedAssoEnds = ownedProperties &
4    ran(ownedAttributes) /\ ran (ownedAssoEnds) = {} &
5    propertyType[union(ran(ownedAttributes))] = ran(primitiveType) &
6    propertyType[union(ran(ownedAssoEnds))] = ran(classType)
```

Figure 6.2: Refining data model properties

In this section, we present Object-to-Relational Refinement Machine. The overall machine can be found in appendix C. Here, we describe main refinement rules, according to which this refinement machine was developed.

Our Object-to-Relational refinement follows the rules defined by [20] that derive a relational database schema from an object model whose semantics is similar to ours. The main idea of these rules is to reorganize an object model into a set of independent classes, so that object-oriented concepts such as inheritance and association are transformed into corresponding concepts in relational database domain.

6.1.1 Refining data model properties

In our object relational refinement context, we make a distinction between *attributes* and *association end*s. This distinction is necessary to facilitate the subsequent refinement of data model properties into SQL implementation features where *attribute*s are mapped into columns typed by SQL basic data types (e.g. varchar and integer) and *association end*s are mapped into foreign keys pointing to primary keys of their class types.

In the abstract machine, *both* attributes and association ends were represented by `ownedProperties`: a single variable typed as a relation between `CLASS`: the set that contains all class names to `PROPERTY`: the set that contains all property names (see Section 5.1.3). In refinement, `ownedProperties` relation is refined into two variables: `ownedAttributes` and `ownedAssoEnds`. Figure 6.2 shows how each of these two variables is typed: as a disjoint subset of `ownedProperties`. The linking invariant conjuncts in lines (5-6) ensure the typing of each variable by relating it

Refinement Variable	Related Abstract Variable	Description of Linking Invariant
inheritedAttributes, inheritedAssoEnds,	ownedProperties (of superclass)	ownedProperties of each superclass in the data model is refined into inheritedAttributes and inheritedAssoEnds in all subclasses in the inheritance hierarchy.
propertyClass	owningClass	A property may be owned by multiple classes along the inheritance hierarchy.

```
1     inheritedAttributes  : CLASS    <-> PROPERTY     &
2     inheritedAssoEnds    : CLASS    <-> PROPERTY     &
3     propertyClass        : PROPERTY <-> CLASS        &
4     dom(inheritedAttributes)<:className              &
5     ! cc.(cc  : className &
6           cc /: dom(superclass) =>
7                   inheritedAttributes[{cc}]= {}) &
8     ! cc.(cc : className &
9           cc : dom(superclass) =>
10                  inheritedAttributes [{cc}] = ownedAttributes
11                                                [closure1(superclass)[{cc}]]) &
12    ...
13    propertyClass = (ownedAttributes \/ inheritedAttributes \/
14                     ownedAssoEnds   \/ inheritedAssoEnds ) ~
```

Figure 6.3: Flattening data model inheritance

to `primitiveType` and `classType` : two typing functions defined in the abstract machine (see Section 5.1.1).

6.1.2 Flattening inheritance

Relational databases do not support inheritance[9]. Several solutions exist (see for example [9, 20]). The approach we follow in refinement is that each subclass in an inheritance hierarchy includes both its own attributes and association ends as well as attributes and association ends inherited from its superclasses. Accordingly, in our refinement we introduce two new variables : `inheritedAttributes` and `inheritedAssoEnds`. The typing and linking invariants of these two variables are shown in Figure 6.3.

Each variable is defined as a relation between `CLASS` and `PROPERTY`. We relate `inheritedAttributes` refined variable into abstract variables using three linking invariant predicates, Figure 6.3, lines 4-11. First, the class that has inherited attributes must be one of the data model classes (a subset of `className`). Second, top-level classes (those that do not have superclasses) are not expected to have any inherited attributes. Finally, *inherited* attributes of a class in the data model is defined in terms of *owned* attributes of any of that class superclasses, using `closure1` transitive

closure operator. The `inheritedAssoEnds` relation is similarly defined: as the set of all class-typed properties owned by any superclass in the inheritance hierarchy.

In addition, variable `propertyClass` was introduced as a refinement of abstract variable `owningClass`. In the abstract machine, `owningClass` was typed as a partial function mapping each named property to is owning class. As a result of flattening inheritance, this variable has been refined into `propertyClass` which is a relation from `CLASS` to `PROPERTY`. Now, a property (either an attribute or an association end) may be owned by more than one class in the inheritance hierarchy.

6.1.3 Introduction of keys

Refinement Variable	Related Abstract Variable	Description of Linking Invariant
`classKey`	`className`, `isAbstract`	a unique `classKey` is defined for each non-abstract class in the data model

```
1    classKey       : CLASS    >+> NAT      &
2    dom(classKey) <: className             &
3    dom(classKey) = isAbstract ~ [{FALSE}] &
```

Figure 6.4: Introduction of class keys

In our AMN formalization, a UML class is represented by a name in `CLASS` : the set that contains all *possible* class names. *Existing* class names that are used in the data model are maintained in `className` which is a subset of `CLASS`. In the relational model, each class must have a key. It is a value-based model, which means that each tuple is identified by a key [55].

In our refinement machine, a key is represented by variable `classKey`, which is a partial injective function from `CLASS` to `NAT` : the set of natural numbers, capturing the fact that each class key is unique to one class. The linking invariant (Figure 6.4, lines 2-3) states that each non-abstract class in the data model must have an identifying key.

6.1.4 Refining data model associations

In the abstract machine, an association is represented as a partial function, from `ASSOCIATION` : the set of possible association names to another partial function that goes from `CLASS` to `CLASS` (see Section 5.1.4). In refinement, we map every data model

association to a table. As such, we introduce `associationTable` as a refinement variable, whose typing and linking invariants are shown in Figure 6.5.

An `associationTable` variable is typed as a partial function from `ASSOCIATION` set to another partial function on `PROPERTY`, representing the two ends of the association table. The linking invariant defines each `associationTable` in terms of abstract `association`. This is done first by relating the names of both variables (line 2) and then by relating the two ends to the two classes pointed to by the abstract association variable using `propertyClass` function (lines 3-7).

Refinement Variable	Related Abstract Variable	Description of Linking Invariant
`associationTable`	`association`	Every data model association is mapped to an association table. The association table consists of two ends owned by the classes participating in the association relationship.

```
1    associationTable  : ASSOCIATION +-> (PROPERTY +-> PROPERTY) &
2    dom (association) = dom (associationTable) &
3    ! (aa,ae1,ae2).(aa : dom (associationTable) &
4            ae1 : dom (associationTable(aa)) &
5            ae2 : ran (associationTable(aa)) =>
6            propertyClass [{ae1}] = dom (association(aa))    &
7            propertyClass [{ae2}] = ran (association(aa)))
```

Figure 6.5: Refining data model association

This concludes our object model data structure refinement. With the introduction of the above refinement variables, our data model structure is aligned to relational data model paradigm. This alignment takes us one step closer towards our goal of generating data migration programs in a relational database. However, before discussing how such an implementation may be generated, we need to show how the description of our abstract model edits may be refined to updated the new refinement variables.

6.1.5 Refinement of abstract machine operations

The introduction of refinement variables, as outlined in the previous section, has a direct impact on the description of our abstract model edits. Each model edit needs to be refined. The refinement of each edit is realized by describing its behavior in terms of the more concrete variables introduced in refinement. At the same time, we need to ensure that the refined behavior of each edit is consistent with its behavior

in the abstract machine. This consistency is ensured by the stated linking invariants that relate refinement variables to their counterparts in the abstract machine. In fact, the main purpose of the refinement proof activity is to demonstrate how this consistency is verified. In the following section, we show parts of model edits refined description. The complete description of refinement proof activity can be found in Appendix D.

Class addition and deletion

In refinement, `addClass` operation is described in terms of its effect on the following refinement variables: `classKey`, `inheritedAttributes`, and `inheritedAssoEnds`.

```
1  addClass ( cName , isAbst , super ) =
2      IF isAbst  = FALSE
3      THEN
4          ANY   cKey
5          WHERE cKey : NAT - ran (classKey)
6          THEN   classKey := classKey \/ {cName |-> cKey}
7          END    ||
8      inheritedAttributes := inheritedAttributes \/
9                ({cName}*(ownedAttributes
10                              [closure1(superclass)[{cName}]]))   ||
11     ...
12     END ;
```

Table 6.1: Part of `addClass` operation refinement

Table 6.1 shows part of the refined description of `addClass` operation. If the added class is not abstract, we assign it a fresh class key (lines 2-7) to enable it to persist data. This is going to be an important feature that we will need when refining classes to database table in the subsequent refinement step.

As a result of flattening inheritance in this refinement step, we need to allow newly introduced subclasses to inherit attributes and association ends of its superclasses along the inheritance hierarchy. Lines 8-10 show how this is done for inherited attributes : by mapping the new class to the set of `ownedAttributes` of all its superclasses, using the transitive closure operator on `superclass` relation, for the new class. A similar technique is done for `inheritedAssoEnds`.

The refined description of `deleteClass` operation follows a similar pattern to `addClass`, but with an opposite effect. One difference to note, though, relates to the update of `associationTable` variable. Here, this variable is updated by collecting all

associations where the class named for deletion (or any of its subclasses) participates and removing those associations from `associationTable` variable, as shown in Table 6.2, lines 6-9.

```
1  deleteClass ( cName ) =
2      ANY  classAssos , subclassAssos
3      WHERE
4          classAssos   =  {asso | asso : ASSOCIATION &
5                                  ((dom(associationTable (asso))    \/
6                                    ran (associationTable (asso)))  /\
7                                    ownedAssoEnds [{cName}]) /= {}} &
8          subclassAssos = ...
9      THEN
10         associationTable := (classAssos \/ subclassAssos ) <<|
11                              associationTable
12     END
```

Table 6.2: Part of `deleteClass` operation refinement

Attribute addition and deletion

The refined behavior of `addAttribute` and `deleteAttribute` is described in terms of its effect on the following refined variables: `ownedAttributes`, `inheritedAttributes` and `propertyClass`. While the effect on `ownedAttributes` is straightforwod: adding or subtracting the named attribute to/from its owning class, the effect on `inheritedAttributes` and `propertyClass` may require further explanation.

As shown in Table 6.3, to update `inheritedAttributes`, if the owning class of the new attribute, represented by `cName` parameter, is a superclass, we identify all subclasses of this superclass using the transitive closure operator, applied on the inverse of `superclass` relation, for the owning class. The resulting set of subclasses is mapped to the new attribute using the cartesian product operator.

The update of `propertyClass` specifies the owning classes of the new attribute after refinement. As a result of flattening inheritance, the added attribute is not only owned by its immediate owning class but also by all subclasses of the owning. `deleteAttribute` operation follows a similar pattern, with a reverse effect and can be seen in Appendix C.

Association addition and deletion

The refined behavior of `addAssociation` and `deleteAssociation` is described in terms of its effect on the following refined variables: `associationTable`, `ownedAssoEnds`,

```
addAttribute ( cName , attrName , type , exp ) =
BEGIN
  ownedAttributes := ownedAttributes \/ {cName|->attrName}              ||

  inheritedAttributes := inheritedAttributes \/
                            closure1 (superclass~)[{cName}]*{attrName } ||
  propertyClass := propertyClass  \/
                        {attrName}*closure1(superclass~)[{cName}] \/
                         {attrName |-> cName }
END ;
```

Table 6.3: Part of `addAttribute` operation refinement

`inheritedAssoEnds` and `propertyClass`.

Table 6.4 shows part of the refined description of `addAssociation`. In refinement, creating an association updates `associationTable` variable by adding source and target association ends, represented by `srcProp` and `tgtProp` respectively, as two ends of the table. These two ends are also mapped to their respective owning classes in the set of `ownedAssoEnds`. If any of the source class or target class is a superclass, we require adding the two association ends to the subclasses of these two classes. This is done by applying the transitive closure on the inverse of `superclass` relation for the respective class, similar to the way we did in the refined substitution of `addAttribute` operation above. The refined description of the `deleteAssociation` is similarly defined and can be found in Appendix C.

```
addAssociation (assoName , srcClass , srcProp ,
                tgtClass , tgtProp , isComp , exp ) =
BEGIN
   associationTable := associationTable \/
                {assoName |-> {srcProp|->tgtProp}} ||
    ownedAssoEnds := ownedAssoEnds \/
                {(srcClass|->srcProp) ,
                (tgtClass |->tgtProp) } ||
    inheritedAssoEnds   :=  inheritedAssoEnds \/
              closure1(superclass~)[{srcClass}]*{srcProp} \/
              closure1(superclass~)[{tgtClass}]*{tgtProp} ||
...
END ;
```

Table 6.4: Part of `addAssociation` operation refinement

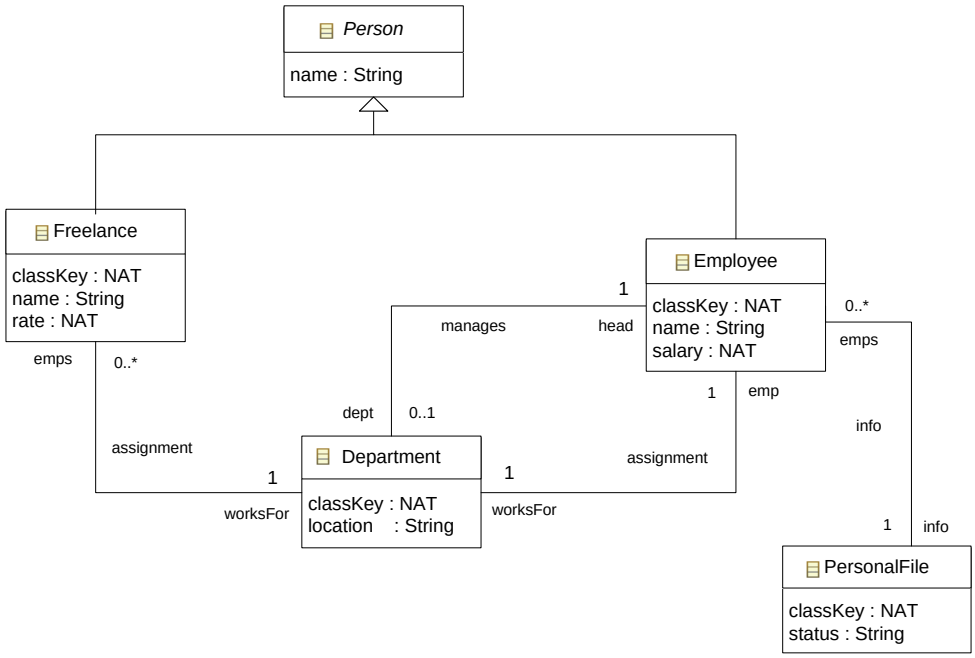


Figure 6.6: Refined Data Model

6.1.6 Example

To demonstrate our Object-to-Relational refinement rules, we use the company data model which we introduced in Chapter 4, Figure 4.5, with its corresponding instance model in Figure 4.6. Applying our refinement rules, we obtain the refined model shown in Figure 6.6. As the figure shows all classes, other than Person class, have got classKey attribute to uniquely identify instances of each class, in preparation for data persistence in the relational model, as we will show in the subsequent refinement step. In addition, Person class attributes (e.g. name) and association ends (e.g. worksFor) have been inherited by Employee and Freelance classes. The instance model of the above refined data model, shown in Figure 6.7, is, to a great extent, similar to the instance model before refinement, with the exception of classKey attribute that is now persisted at the instance level and updated with the object identifier of objects of each class. Using our refined data structure, this Object-Relational model can be represented as follows, in part:

```
...
propertyClass   = {(name |-> Person),(name |-> Employee),
                   (name |-> Freelance),(worksFor|->Freelance)
                   (rate |-> Freelance),(salary  |->Employee),
                   (worksFor|->Person) ,(worksFor|->Employee),...}
```

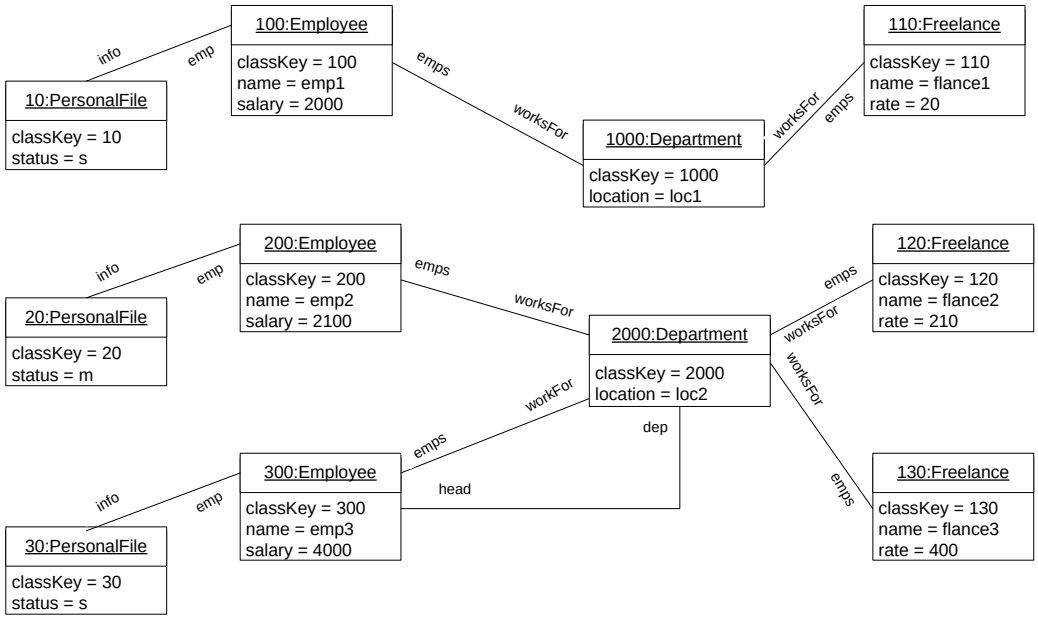


Figure 6.7: Refined Instance Model

```
ownedAttributes      =  {(Person     |-> name),
                         (Department |-> location)
                         (Freelance  |-> rate),
                         (PersonalFile |-> status),...}
ownedAssociationEnds  = {(Person |-> worksFor),
                         (Department |-> employees),
                         (PersonalFile |-> employee},
                         (Employee |->info),...}
inheritedAttributes   = {(Freelance |-> name),(Employee |-> name)}
inheritedAssoEnds     = {(Freelance |-> name),(Employee |-> name)}
...
```

To show how the refined behavior of our model edits can be used to update refinement variables as a result of updates in the abstract machine, assume that our original model has been updated with a new class named BankDetails to hold bank account information of each person working for the company. In addition, a new association named paymentInfo is added with two association ends: bankData of type BankDetails and payee of Person. Further, assume that a new attribute named lastPaymentDate was introduced in Person class to record the last date on which an employee or a freelance received a payment from the company.

These changes in the data model can be described using our model edits : `addClass` to add the new class , followed by `addAssociation` to add the new association together with its two association ends and finally, `addAttribute` to add lastPaymentDate feature. The semantics of this update at the abstract machine level has already been described in Section 5.3. Here, we are interested in demonstrating the updates of the variables introduced as a result of refinement. This can be seen by enumerating the refinement variables that both `addClass` and `addAssociation` update as shown below:

```
addClass(BankDetails) =
...
    classKey := classKey \/ {BankDetails |-> cKey}

addAttribute('Person','lastPaymentDate', ...) =
  ownedAttributes := ownedAttributes \/
                        {Person |-> lastPaymentDate}
  propertyClass   := propertyClass \/
                        {(lastPaymentDate |-> Employee),
                        (lastPaymentDate  |-> Freelance),
                        (lastPaymentDate  |-> Person)}
  inheritedAttributes := inheritedAttributes \/
                           {(lastPaymentDate |-> Employee),
                           (lastPaymentDate |-> Freelance)}
addAssociation('paymentInfo', 'Person', 'account',
                'BankDetails', 'payee', ...) =
  associationTable := associationTable \/
                        {paymentInfo|-> {account |-> payee}}
  ownedAssoEnds := ownedAssoEnds \/
                        {(Person |-> acount) ,
                        (BankDetails |-> paye)}
  inheritedAssoEnds := inheritedAssoEnds \/
                         {(Employee, account), (Freelance, account)}
```

6.2 Generating data migration programs

In this section we describe the second refinement step that we follow to generate SQL implementation programs, corresponding to our data model evolution operations.

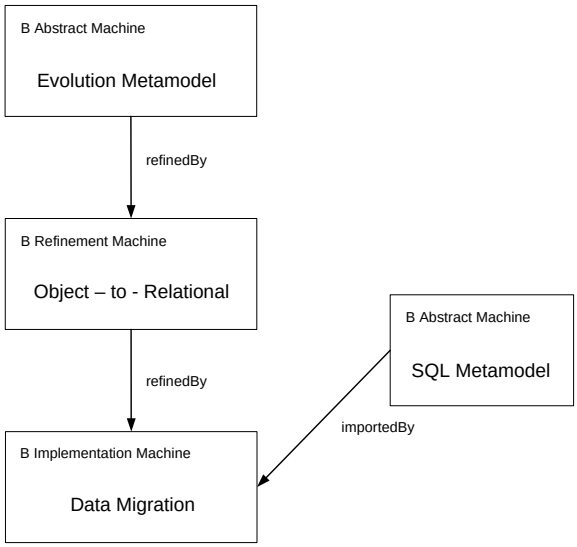


Figure 6.8: Main components of B specifications

Here, we show how the Object-to-Relational Refinement, obtained in the previous refinement step, can be refined further into a data migration program in SQL. To achieve this goal, we perform *two* transformations. First, we transform our data model structural properties into equivalent structural properties in SQL. Second, we transform our model edits into corresponding statements in SQL.

Figure 6.8 shows an overview of B specification elements involved in the refinement step which we perform in this section. For the sake of this refinement step, we require a description of SQL language in B-method. This description can be obtained by translating an SQL metamodel into a corresponding abstract machine in AMN, similar to the way we formalized UML in B (refer to Section 5.1). The main aspects of AMN formalization of SQL metamodel is explained in the following section. For a complete description of the generated SQL machine, please refer to Appendix C.

Given that Object-to-Relational refinement was a refinement of the abstract evolution machine, if we can demonstrate that the generated SQL implementation is a refinement of Object-to-Relational Refinement, then we can conclude that the SQL implementation is, indeed, a refinement of the evolution abstract machine, based on the transitive nature of refinement [6].

6.2.1 Formalizing SQL metamodel in AMN

Structured Query Language (SQL) is currently supported in most relational database management systems and is the focus of an active standardization process. In our formalization, we use a subset of SQL-Foundations part of SQL-2003 syntax [84, 34].

```
1  tableName     <: TABLE                    &
2  tableColumns   : TABLE  <-> COLUMN        &
3  columnName    <: COLUMN                   &
4  columnType     : COLUMN +-> Sql_TYPE      &
5  parentTable    : COLUMN <-> TABLE         &
6  canBeNull      : COLUMN +-> BOOL          &
7  isID           : COLUMN +-> BOOL          &
8  isUnique       : COLUMN +-> BOOL          &
9  primaryKey     : TABLE  <-> COLUMN                &
10 foreignKey     : TABLE  <-> (COLUMN --> TABLE)    &
11 tuple          : TABLE  <-> (COLUMN +-> Sql_VALUE)
```

Table 6.5: Part of the formalization of SQL data structure in AMN

This part of SQL standard defines the data structures and basic operations on SQL data. Below, we focus on conceptualizing the main features of that standard that are relevant to our selected UML data metamodel subset and to the implementation of our evolution primitives.

Table 6.5 shows the formalization of SQL metamodel data structure in AMN. Tables in an SQL model are named elements. They are represented by `tableName`, typed as a subset of `TABLE`: the set of all possible table names. Table columns are represented by a relation from `TABLE` to `COLUMN`: the set of all possible column names.

Each column in an SQL model has a name represented as an element of `columnName` set, and a type represented by `columnType` partial function. We use the given set of `Sql_Type` to represent all SQL basic data types (e.g varchar, integer, decimal, etc.). A column is related to its owning table using `parentTable` relation. Other SQL column characteristics we define include `isID` which specifies whether the named column acts as the primary key of its parent table, `canBeNull` which specifies whether a column must be valued and `isUnique` which specifies whether a column value is unique in the table extension (described below).

A key in SQL can either be a `primaryKey`, defined by one or more table columns to uniquely identify table rows or a `foreignKey`, defined by one or more table columns, each referring to a table in SQL model.

SQL state denotes the extension part (set of instances) of an SQL model. We represent SQL state as a set of named tables, each of which is mapped to pairs of columns and values. We use `Sql_VALUE` to denote the set of possible SQL values of columns: a union over `Sql_TYPE`.

```
1  /***Basic table operations***/
2  addTable (tName)  = ...
3  alterTable (tName, colName) =
4    PRE   tName : tableName & colName : COLUMN
5    THEN
6           IF  (tName |-> colName)  /: tableColumns   THEN
7                tableColumns  := tableColumns \/ {tName |-> colName}
8           ELSE
9                tableColumns  := tableColumns -  {tName |-> colName}
10          END
11               tableColumns  := tableColumns \/ {tName |-> colName}
12     END ;
13 updateTable (tName) = ...
14 removeTable (tName) = ...
15
16 /***Basic column operations***/
17 add_id_Column (colName , tName , type) =
18 addColumn (colName , tName , type)= ...
19 removeColumn (tName, colName ) = ...
20
21 /***Basic key operations***/
22 add_pk_Key (colName, tName) = ...
23 remove_pk_Key (tName) = ...
24 add_fk_Key (colName, tName1, tName2) = ...
25 remove_fk_Key (tName) = ...
26
27 /***Basic value operations***/
28 setValue ( tName , colName , initialValue ) =...
29 removeValue ( tName ) =...
30
```

Table 6.6: Part of the formalization of SQL basic operations

The other aspect of SQL metamodel that needs to be formalized in B is the basic behavior properties which are used to interpret our proposed abstract evolution operations. Table 6.6 shows SQL basic operations formalized in B. These abstract operations consist of substitution statements, acting on variables representing SQL data structure. For example, basic operations on tables include `alterTable` (lines 3-12), which adds a name to the existing set of table names and `removeTable`, which performs the reverse substitution. Similar operations are defined to manipulate columns, primary and foreign keys and to set or remove table values. Other auxiliary operations are defined to facilitate the implementation of data model evolution operations in SQL. For example, `getColumns(tName)` operation returns a sequence of column

```
/***SQL metamodel auxiliary operations***/
allTablecolumns <-- getColumns(tName )  = ...
fkOwngTables <--getForgnKeyTables(tName) =
    PRE   tName : tableName
    THEN
        ANY tables, result WHERE
            tables <: TABLE &
            tables = { ta | ta : TABLE & ta : dom(foreignKey) &
                #co.(co : COLUMN & co |-> tName : foreignKey(ta))} &
            result : iseq(TABLE) &
            ran(result) = tables
        THEN
            fk_owng_Tables := result
        END
    END;
```

Table 6.7: Part of the formalization of SQL auxiliary operations

names specified for a particular table. Operation `getForgnKeyTables(tName)` (lines 33-44) return a sequence of tables names which own foreign keys referring to a specified table name.

6.2.2 Linking data model state to SQL state

To generate SQL implementation that faithfully represents our abstract evolution operations, we need to map our data metamodel concepts into corresponding SQL concepts. Since our main focus is on the interpretation of evolution operations in SQL, we will assume a simple mapping of the data structures in the two domains. In our framework, the mapping between the two domains is performed using linking invariants. The main purpose of these invariants is to precisely describe how a concept in the object model domain (e.g. `inheritedAttributes`) is defined in a relational model. Figure 6.9 provides an overview of how Object-to-Relational model concepts are mapped to corresponding SQL model concepts. The table provides informal description of the mapping rules which are formally characterized, in part, in Table 6.8. To facilitate the specification of linking invariant, we introduce a number of mapping functions (e.g. `propertyToColumn`), each taking an object domain concept and returning a corresponding relational concept. Below we present examples of linking invariants. Complete characterization of these invariants can be found in Data Migration implementation machine in Appendix C.

In our SQL implementation, we require that each class or association in the data

Object Relational Model Concepts	SQL Model Concepts	Brief Description
`className`	Table	Each non-abstract class and association is refined into a relational table.
`associationName`	Table	
`ownedAttributes`	Column	Attributes owned by a class or inherited from class superclasses are refined into columns in the table corresponding to the class.
`inheritedAttributes`	Coumn	
`ownedAssociationEnd`	Column and foreign key	Same as owned and inherited attributes with the addition of foreign key definition pointing to the table corresponding to association end type.
`inheritedAssociationEnd`	Column and foreign key	
`classKey`	Id Column and primary key	
`extent`	tuple	The extent of a class is mapped to a table tuple. Extent object ids are mapped to values representing the table id columns.
`value`	tuple	Each data model value is mapped to a tuple sql value using two mapping functions : one to map property to corresponding columns and one to map actual values to corresponding values.
`link`	tuple	Each data model link is mapped to a tuple where tuple table is mapped to association name and tuple column values representing table foreign keys.

Figure 6.9: Overview of linking invariants relating Object-to-Relational to SQL

model is represented as a separate table in the SQL model. The linking invariant (table 6.8, line 2-5) characterizes this fact. This linking invariant uses `classToTable`: a mapping function that takes a class a returns its corresponding table.

In Object-to-Relation model, `inheritedAttributes` are defined as a relation between `CLASS` set and `PROPERTY` set. In SQL model, the attributes are mapped to columns in all tables corresponding to subclasses of attribute owning class. In characterizing this linking invariant we use three mapping functions : `classToTable`, `propertyToColumn` and `tableHierarchY` which returns a sequence of tables corresponding to subclasses of a particular superclass.

Object-to-Relation association ends are mapped to SQL foreign keys (lines 14-23). This is characterized by stating that each association name is mapped to an association table (using `assoToTable` mapping function) while each association end is mapped to a foreign key owned by a table corresponding to association end owning class.

Finally, the last linking invariant in Table 6.8 relates data model values to corresponding SQL tuple values. This is done by requiring that such a value exists in

```
/* mapping classes to tables */
tableName <: classToTable [className ] &
    ! cl. (cl:className & cl:dom (classKey) => cl:dom(classToTable)) &
dom ( tuple ) =
    classToTable[dom(extent)] \/ assoToTable [dom(link)]             &

/*mapping inherited attributes to columns */
inheritedAttributes = { cc, att | cc : CLASS &
        att : PROPERTY &
        classToTable(cc) : ran(tableHierarchy(cc)) &
        propertyToColumn(att) : tableColumns[{classToTable(cc)}] } &

/* mapping association ends to foreign keys */
! ( assoName , me1 , me2 ) . ( assoName : dom ( memberEnds ) &
        me1 : dom (memberEnds (assoName)) &
        me2 : ran (memberEnds (assoName)) =>
        assoToTable (assoName) : dom ( foreignKey) &
        # ff . (ff : foreignKey [{assoToTable (assoName)}] &
            propertyToColumn (me1) |-> classToTable (owningClass(me1)):ff) &
        # ff . (ff : foreignKey [{assoToTable(assoName)}] &
            propertyToColumn (me2) |-> classToTable (owningClass(me2)):ff) &
        propertyToColumn (me1) |-> classToTable (owningClass (me1)) :
        union (foreignKey [{assoToTable (assoName)}])) &

/* mapping property values to column values */
! (pp , oid ) . (pp : dom (value) &
        oid : dom (value (pp)) =>
        mapSqlValue ((value(pp)(oid))) =
        union (tuple[classToTable [propertyClass[{pp}]]])(propertyToColumn(pp)))
```

Table 6.8: Part of linking invariants relating Object-to-Relational to SQL

tuples of all tables corresponding to class owning the property defining such value.

6.2.3 Example

The data model of our running example, which has been refined into the model shown in Figure 6.6 can be refined into the SQL model shown in Figure 6.10. In particular, each (non-abstract) class or association in the refined data model is refined into a relational table. In our example, the abstract class Person did not have a corresponding table. Other classes such as Employee and Department have corresponding tables, as shown in Figure 6.10.

In addition, data model associations such as *info* relating Employee and PersonalFile classes, have been refined into separate tables, regardless of the multiplicities of

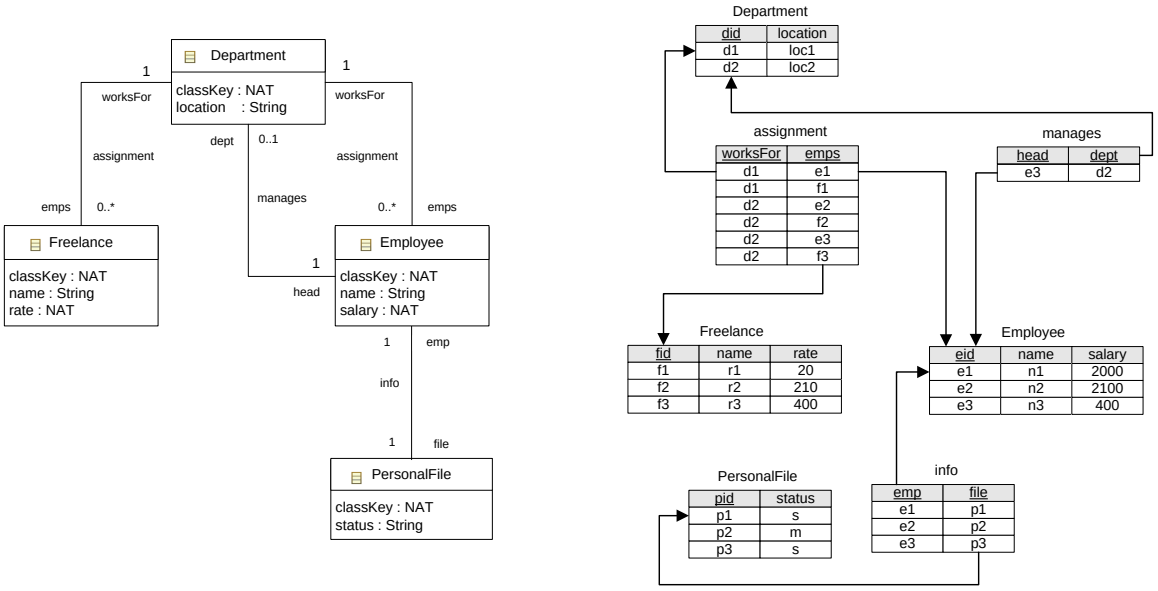


Figure 6.10: Mapping to SQL

their association ends. Each of the association tables has an id column on which a primary key is defined. For example, table Department has departmentID as the id column and primary key and table PersonalInfo has personalInfoID as an id column and a primary key. All data model attributes (owned or inherited) are refined into columns in tables corresponding to their respective owning classes. For example, in table Department, the location column corresponds to a similarly named attribute in class Department in the data model. In table Employee, the name column corresponds to similarly named attribute inherited from Person class.

Data model association ends (owned or inherited) are also refined into columns in tables corresponding to their respective owning classes. The difference here is that on such columns we define foreign keys pointing to the primary key of the tables corresponding to the association end type class. For example, in table assignment, the worksFor column points to the primary key of the Department table. Similarly, in table info, the column file points to the primary key of PersonalFile table.

6.2.4 Implementation of evolution operations in SQL

As described in Chapter 2, an implementation is a particular type of B machine, typically used to refine abstract specifications or other refinement machines into more concrete representations, close to to a specific language of executable code. As one of our main goals in this chapter is to generate executable data migration programs that

correspond to our abstract evolution specifications (and their refined specifications in Object-to-Relational model), using B-method implementation mechanism seems a natural step. Indeed, the formalization of SQL into an abstract machine and the characterization of linking invariants between data model and SQL model states makes us ready to start on this implementation step.

In this implementation, we will perform the second refinement step outlined in Figure 6.1. To perform data structure refinement, we use the linking invariants described in Section 6.2.2. These invariants are essential for verifying the correctness of implementation. To perform substitution refinement, we describe how our abstract evolution operations may be implemented using specific SQL operations. The correctness of this implementation will be determined by discharging implementation proof obligations which are similar to those generated for Refinement. Appendix D reports on our proof activity.

Class addition and deletion

The operation `addClass` can be implemented in the SQL model as shown in Table 6.9. We add a table with a name corresponding to the class name. We then add an id column and designate it as the primary key of the new table. If the added class is a subclass, we need to collect columns of all tables corresponding to its superclasses. This is done by an implementation loop which uses a condition defined based on the size of `inh_Columns` variable. This variable is populated by `columnHierarchy`: a mapping function that takes a class and returns a sequence of columns. Inside the loop, for each column, identified by `curr_column` local variable, we perform a sequence of two steps: we first alter the new table and then add each identified column. To verify loop correctness, the predicate in loop `INVARIANT` clause (lines 23-26) asserts that the column id in which the primary key of the new table was defined, union the the set of columns added by the loop must always be a subset of the columns of the new table.

The implementation of `deleteClass` requires defining two implementation loops. The first loop will de-link foreign keys pointing to the table corresponding to the class being deleted. We use `getForgnKeyTables` query operation to get a sequence of tables owning these foreign keys. We then loop through the returned sequence of tables to remove the columns on which the foreign keys were defined. The second loop of `deleteClass` operation (part of which is shown in Table 6.10) is a loop through the columns of the table corresponding to the class to be deleted. Also here, we introduce an SQL query operation `getColumns` that takes a table name and returns a sequence

```
addClass (cName , isAbst , super) =
BEGIN
      VAR
         tName, colName, colType, counter, curr_column
      IN
         tName := classToTable(cName);
         colName := tName;
         colType := mapClassKey(cName);
         inh_Columns := columnHierarchy (super);
         counter := 1 ;
         curr_column := inh_Columns (counter);

         addTable (tName);
         add_id_Column (colName,tName,colType);
         add_pk_Key (colName,tName);

         WHILE
            counter <= size (inh_Columns)
         DO
            alterTable (tName, curr_column);
            addColumn  (colName,tName, colType);
            counter := counter + 1
         INVARIANT
            primaryKey [{tName}] \/
               propertyToColumn [inh_Columns [1..counter]] <:
                  tableColumns[{tName}]
         VARIANT
            card (inh_Columns) - counter
         END
      END
END;
```

Table 6.9: Implementation of class addition

of columns. We iterate through the returned columns and, in every pass of the loop, we alter the table by removing the column. The loop invariant is self-explanatory. After the second loop is over, we remove the primary key of the table corresponding to the deleted class and delete the entire table. The complete description of the implementation of this operation can be seen in Appendix C.

Attribute addition and deletion

The operation `addAttribute` can be implemented, as shown in table 6.11, by altering the table that corresponds to the owning class of the new attribute. We then add a

```
1     ...
2     VAR
3        allTableColumns, count_column
4     IN
5        allTableColumns := getTableColumns(classToTable(cName));
6        count_column    := 1;
7     WHILE
8        count_column < size(allTableColumns)
9     DO
10       VAR curr_column
11       IN
12          curr_column := allTableColumns (count_column);
13          alterTable (tName);
14          removeColumn (tName, curr_column);
15          count_column := count_column +1
16       END
17    INVARIANT
18       !(ta, col) . (ta : dom(tableColumns) &
19          col : tableColumns[{ta}] =>
20            col |-> ta : parentTable)
21    VARIANT
22       card (allTableColumns) - count_column
23    END
24 END
```

Table 6.10: Part of implementation of class deletion

column with a name and a type corresponding to the name and type of the added attribute. In this implementation, the type mapping is done using `sqlType` function, defined as an injection from `TYPE` (the given set of all types in the data model) to `Sql_TYPE` (the given set of all types in the SQL model).

The `addAttribute` operation may take an expression (denoted by `exp` in `addAttribute` operation parameters). This is used to describe an initial value of the added attribute. To be able to instantiate the added attribute with the value of such expression, we need to translate it into an equivalent SQL value expression. This is done using `translate` function (line 9). Translated value is used to initialize `initialValue` local variable, we then use `setValue` operation to instantiate table tuples accordingly (line 13).

If the attribute being added belongs to a superclass, we need to map the added attribute to all tables in SQL model which correspond to subclasses of attribute owning class in the data model. This can be achieved using a `WHILE` loop construct, which captures the tables in SQL model corresponding to subclasses of a particular

```
addAttribute (cName,attrName,type,exp) =
   BEGIN
      VAR
         tName , colName , colType , initialValue,...
      IN
         tName  := classToTable (cName) ;
         colName := propertyToColumn ( attrName ) ;
         colType := sqlType ( type ) ;
         initialValue := translate(exp);
         alterTable (tName , colName) ;
         addColumn (colName , tName , colType) ;
         updateTable (tName);
         setValue(colName,initialValue)
...
     END
END ;
```

Table 6.11: Part of implementation of attribute addition

superclass in the data model. The loop terminates when the new attribute is mapped to a column in every table corresponding to a subclass of `cName` in the data model. This loop implementation can be seen in Appendix C. The correctness of this loop is demonstrated in Appendix D as part of the Proof Activity.

The implementation of `deleteAttribute` follows the inverse of the same implementation pattern described above for `addAttribute` operation, and can be seen in Appendix C.

Association addition and deletion

The `addAssociation` operation creates a table to store the links between objects of the source and target classes participating in an association relation. An important consideration in our implementation strategy is to store relationship information such as association, seprately from the objects that participate in such a relation. Accordingly, in the implementation of this operation, we create a table (with an id column and a primary key) and a pair of columns corresponding to the two association ends (denoted by `srcProp` and `tgtProp`, in `addAssociation` operation parameters). As with `addAttribute`, this implementation must be followed by an `updateTable` and `setValue` operations that inserts the values of the expression `exp`, passed as a parameter of the operation.

If one of the association ends introduced in `addAssociation` operation belongs

```
1
2  addAssociation ( assoName , srcClass , srcProp ,
3                   tgtClass , tgtProp , isComp , exp ) =
4  ...
5  inh_Tables_src := tableHierarchy (srcClass)
6  ...
7  VAR counter IN
8      counter := 1 ;
9    WHILE
10       counter <= size (inh_Tables_src)
11   DO
12       VAR current_table IN
13           current_table := inh_Tables_src(counter) ;
14
15            alterTable  (current_table, colName) ;
16            addColumn   (firstColumnName,current_table,firstColumnType);
17            add_fk_Key  (firstColumnName,current_table,tgtTable);
18            updateTable (current_table) ;
19            setValue    (current_table,firstColumnName,initialValue)
20            counter     := counter + 1
21        END
22     INVARIANT
23            inh_Tables_src = tableHierarchy(srcClass) &
24            ...
25     VARIANT size (inh_Tables_src)-counter
26     END
27   END
28   ...
29
```

Table 6.12: Part of implementation of association addition

to a superclass, we use a `WHILE` loop construct, to update tables corresponding to subclasses of that superclass. Table 6.12 shows part of the `WHILE` loop used to update inherited tables corresponding to subclasses of the association source class, denoted by `inh_Tables_src`.

Applying `tableHierarchy` mapping function on `srcClass` parameter, we get a sequence of tables corresponding to subclasses of the source class (line 5). We use `WHILE` loop to iterate through this sequence of tables. In every loop pass (linex 12-20), we alter each table in the sequence by adding a column corresponding to the source property, define a foreign key on the added column and set initial value if a default value expression is defined. Loop correctness can be verified base on the invariant predicate, part of which is stated in line 24. For example, the loop should not add or

remove any of the inherited tables (i.e. those tables should remain the same before, while and after the loop executes).

A similar `WHILE` loop is defined to update tables corresponding to subclasses of the association target class. The complete description of `addAssociation` operation and the implementation of `deleteAssociation` operation can be found in Appendix C.

6.3 Generating SQL data migration programs

Having instantiated SQL metamodel with particular values, based on the interpretation realized by our B implementation mechanism, we may define corresponding textual representations that are close to SQL instantiated metamodel. These textual representations may take the form of templates that can be populated from values in the SQL metamodel. Below, we show how such SQL textual representation templates can be defined for each implemented evolution operation.

Here, the basic idea is to to apply a set of pre-defined textual templates over SQL metamodel instantiated in the previous refinement step. Essentially, in this step we follow a template-based generation of the required SQL statements required to build a data migration program. Templates are grouped depending on the kind of model evolution operation (`addClass` , `deleteAttribute`, etc.). The variable parts of templates are defined between angle brackets. These variable parts are instantiated with the concrete information from the given SQL metamodel to obtain the final statements. These statements constitute a data migration program [1].

Given an SQL metamodel properly instantiated with the interpretation of `addClass` evolution operation outlined in the B implementation above, the following template can be used to generate corresponding SQL statements:

```
CREATE TABLE <tName> (
 <tName_id> INT NOT NULL,
 PRIMARY KEY (<tName_id>)
 <columnName 1><columnType 1><columnConstraint 1>
 ,...,
 <columnName N><columnType N><columnConstraint N>
);
```

[1] as SQL could have different dialicate implemented by different RDBMS, the above templates have been instantiated and tested on Apache Derby [11], which is part of Eclipse Data Tool Platform [79].

where columnName 1,...,columnName N are column names obtained from the WHILE loop in B implementation of addClass that collected columns defined by the superclasses of the class; <columnType 1>,...,<columnType N> refer to the SQL type of corresponding columns, properly mapped during B implementation; <columnConstraint1>,..., <columnConstraintn> refer to SQL constraints (e.g. UNIQUE, NOT NULL) obtained from the instantiated SQL abstract machine variables of the respective columns.

Similar to the way we defined SQL textual template for addClass operation, SQL template corresponding to deleteClass operation is defined below. The representation of this template corresponds to B implementation steps used for deleteClass operation. As such, we are able to use the values of SQL variables updated during the implementation to instantiate variable elements of the proposed template:

```
ALTER TABLE <FK_OWNING_TABLE 1>;
DROP CONSTRAINT<FK1>; DROP COLUMN <FK_COLUMN 1>;
...;
ALTER TABLE <FK_OWNING_TABLE N>;
DROP CONSTRAINT <FK N>; DROP COLUMN <FK_COLUMN N>;

ALTER TABLE <tName>;
DROP COLUMN <COLUMN 1>;
...;
ALTER TABLE <tName>;
DROP COLUMN <COLUMN N>;

ALTER TABLE <tName>;
DROP CONSTRAINT <PK>; DROP COLUMN <id_COLUMN 1>;
DROP TABLE < tName>;
```

In the above template <FK_OWNING_TABLE 1> <FK_OWNING_TABLE N> refer to tables owning foreign keys pointing to the table to be deleted. Here, we use the values collected by the WHILE loop in B implementation of deleteClass operation and instantiate this template for all these tables and for the foreign keys that need to be removed, represented by parameters <FK 1>,...,<FK N>. Although SQL syntax allows columns on which these foreign keys are defined to remain in the table after foreign keys deletion, we take the position that such columns are subsequently deleted. Since

our ultimate goal here is to delete the table to which these foreign keys were pointing, values stored in the these columns will loose its semantic validity.

In the subsequent statements of the above template, `ALTER TABLE<tName>` refers to the table corresponding to the class to be deleted. Before this table can be deleted, we first need to delete all of its defined columns, represented by `<COLUMN 1>`,..., `<COLUMN N>` parameters. In addition to dropping all defined columns, we also need to drop the primary key of the table and the id column on which the primary key was defined, before we are, finally, able to drop the table itself.

The SQL template of `addAttribute` operation can be specified as a pair of updates: first to the schema, and then to the rows of the table in question. In addition, here, we must consider not only altering the table where the corresponding column needs to be defined but also all other tables that correspond to the column owning table. The identification of these tables (corresponding to subclasses of attribute owning class) has been done during B implementation. These tables now instantiate proper elements of the SQL metamodel and can be used to instatntaie variable parameters of the template below:

```
ALTER TABLE <tName>;
ADD COLUMN  <columnName><columnType>;
UPDATE <tName>  SET [tuple = exp];

ALTER TABLE <tName 1>;
ADD COLUMN  <columnName 1><columnType 1>;
UPDATE <tName 1>  SET [tuple = exp];
...;
ALTER TABLE <tName N>;
ADD COLUMN  <columnName N ><columnType N>;
UPDATE <tName N>  SET[tuple = exp];
```

In the first part of the above template, we modify the table corresponding to added attribute owning class; by adding column name and type corresponding to the added attribute name and type. We then use an update statement to update specific table rows using a translated default value expression. In the second part of the template, we add the new column and update table values, following a similar pattern, for tables identified as corresponding to subclasses of the new attribute owning class,

as outlined above.

The SQL template of `deleteAttribute` operation is similarly defined. It represents a modification to the table corresponding to the owning class of the attribute to be deleted and a modification to all tables corresponding to subclasses of that class. Here, since the entire column is deleted , we do not need to apply any update on values stored by the deleted column. The template for `deleteAttrribute` is shown below:

```
ALTER TABLE <tName>;
DROP COLUMN  <COLUMN>;

ALTER TABLE <tName 1>; DROP COLUMN  <COLUMN 1 >;
...;
ALTER TABLE <tName N>; DROP COLUMN  <COLUMN N >;
```

As explained in the B implementation of `addAssociation`, an important consideration in our implementation strategy is that we create a table corresponding to every named association in the data model, regardless of the multiplicity of its two association ends. Similar to tables generated for non-abstract classes, this table will have an id column and a primary key. However, given that we only allow binary associations, such table will always consist of two columns, each representing one association end. In addition, a foreign key constraint will be defined on each of these two columns, pointing to the primary key of the table corresponding to the type of the original association end. Accordingly, SQL textual template for representing this operation takes the form specified below:

```
CREATE TABLE <tName> (
<tName_id> INT NOT NULL,
PRIMARY KEY (<tName_id>)
);

ALTER TABLE <tName>;
ADD COLUMN  <firstColumnName><firstColumnType>,
Foreign Key <firstColumnName> REFERENCES <tgtTable>.(<tgtTable_id>);
UPDATE <tName>  SET [tuple = exp];

ADD COLUMN  <secondColumnName><secondColumnType>,
```

```
Foreign Key <secondColumnName> REFERENCES <srcTable>.(<srcTable_id>);
UPDATE <tName>  SET [tuple = exp];

ALTER TABLE <tName 1>;
ADD COLUMN  <columnName 1>< columnType 1>;
Foreign Key <firstColumnName> REFERENCES <tgtTable>.(<tgtTable_id>);
UPDATE < tName 1 >  SET [tuple = exp];
...;
ALTER TABLE <tName N>;
ADD COLUMN  <columnName 1><columnType 1>;
Foreign Key <firstColumnName> REFERENCES <tgtTable>.(<tgtTable_id>);
UPDATE <tName N>  SET [tuple = exp];

ALTER TABLE <tName 1>;
ADD COLUMN  <columnName 2 ><columnType 2>;
Foreign Key <secondColumnName> REFERENCES <srcTable>.(<srcTable_id>);
UPDATE <tName 1>  SET [tuple = exp];
...;
ALTER TABLE <tName N>;
ADD COLUMN  <columnName 2>< columnType 2>;
Foreign Key <secondColumnName> REFERENCES <srcTable>.(<srcTable_id>);
UPDATE < tName N>  SET [tuple = exp]
```

The first part of the template creates an association table together with an id column and a primary key. Subsequently, this table is altered to have the two columns corresponding to the two association ends, each with a foreign key definition. The `UPDATE...SET` statement is used to update table extension with default values corresponding to the translated expression input parameter. In subsequent parts of the template, we alter tables belonging to table hierarchy of the source or target table (i.e. those tables corresponding to classes related to source or target class of the association in the data model through a subclass relationship) following a pattern similar to the pattern we used in altering the association table.

The SQL template for `deleteAssociation` operation takes the form specified below:

```
ALTER TABLE <tName>;
DROP CONSTRAINT <FK1> DROP COLUMN <firstColumnName>;
```

```
DROP CONSTRAINT <FK2> DROP COLUMN <secondColumnName>;
DROP CONSTRAINT <PK> DROP COLUMN <id_COLUMN>;
DROP TABLE <tName>
```

In the above template, we start by altering the table corresponding to the named association. To be able to drop the two columns corresponding to the association ends, we first drop the foreign key constraints defined on these columns. We then drop the primary key constraint defined on the id column, before we drop the association table itself.

Compound evolution operations such as `inlineClass` can be implemented as a sequence of `addAttribute` operations one for each of the attributes of the target class, followed by an update to insert the appropriate values, then a delete to target class. Since the source class and target class tables are not directly linked, we need to perform an inner join to the table corresponding to the association relationship between the two classes to select appropriate records from the target class table columns before it is dropped:

```
ALTER TABLE (<srcClass>),
ADD COLUMN <srcClass_refClassAttribute>
UPDATE <srcClass>
SET <srcClass> . <refClassAttribute> =
  (SELECT <refClass> . <refClassAttribute>
     FROM <refClass>
     INNER JOIN <assoName>
   ON <assoName.srcProp> = <refClass>.<refClass_idColumn>
   INNER JOIN <srcClass>
   ON <assoName.tgtProp = <srcClass>.<srcClass_idColumn>

<deleteClass <refClass> template>>
```

Example

Within the context of our running example and following the SQL template mapping rules outlined above, we could generate SQL executable code to migrate data persisted under data model version 1 and its conformant instance model, represented by Figure 4.5 and Figure 4.6 respectively so that the migrated data can be persisted under model

version 2, represented by Figure 4.10. The corresponding SQL data migration code is shown below:

```
--SQL code generated for inlineClass(Employee, employee, PersonalFile);

alter table Employee add column maritalStatus char (32);
alter table Employee add column location char (32);

update Employee
set maritalStatus =
( SELECT p. maritalStatus
    FROM Personalfile p
    INNER JOIN  info i
    on i.info = p.personalfile_id
    INNER JOIN employee e
          on i.employee = e.employee_id);

--similarly for updating location column
...
alter table  PersonalFile drop column maritalStatus;
alter table PersonalFile  drop column location;
drop table  PersonalFile;


--SQL code generated for addAttribute(Employee, seniority: String
   [if self.age > 50 then 'senior' else 'junior']) ;

alter table Employee add column seniority char (32);
update Employee
   set seniority = CASE
      WHEN age = 50 then 'senior' else 'junior' end

--SQL code generated for modifyAssociation(Department, manager, Employee, 1,1) ;
-- user input is required : provide values for null entries

alter table manages
alter column head not null

--SQL code generated for extractClass(Department, Project, projects)

create table PROJECT (
   project_id int not null GENERATED ALWAYS AS IDENTITY,
   primary key (project_id),
   location varchar(32)
);
insert into PROJECT (LOCATION)
   select LOCATION
   from DEPARTMENT d
```

```
46      where d.PROJECT is not null
47
```

```
1   create table projects (
2      projects_id int not null GENERATED ALWAYS AS IDENTITY,
3      primary key (projects_id),
4      department int,
5      foreign key (department) references department(department_id),
6      project int,
7      foreign key (project) references project(project_id)
8   );
9
10  -- updating the association table
11  insert into PROJECTs(department,project)
12     select department_id,project_id
13     from DEPARTMENT d, COMGTSYSTEM_V1.project p
14     where p.location = d.location
```

The generated code above is an instantiation of SQL templates which, in turn, instantiate our B Data Migration implementation. As a direct result of following B refinement (and implementation) mechanism, we can guarantee that the code above, representing the SQL interpretation of an evolution model preserves the consistency constraints at the object model level. Hence, it ensures that corresponding data will be migrated in a way that preserves the invariants of the new data model .

We may observe that the SQL implementation of `modifyAssociation` (lines 30-34) would generate an error : the column named `head` in table `manages` can not be constrained into a non-nullable value unless all entries in the table meet this new constraint. This requires user input to assign existing null-valued records appropriate values. Such an error should not come as a surprise to the designer, as it was highlighted early while providing the specifications of the evolution at the data model level (please see end of Section 4.4).

In practice, non-trivial preconditions for the migration are more likely to arise out of changes to multiplicities, to value types or ranges, or to constraints associated with model elements, then it is entirely possible that the precondition for the evolution sequence would not hold for the existing data. An important value of our approach is that such errors can be highlighted at an early design stage prior to committing implementation resources.

Chapter 7

Discussion

The potential impact of Model-Driven Engineering (MDE) is considerable: the ability to generate an implementation directly from an abstract model can greatly contribute to system development efficiency. Within the domain of Information Systems (IS), the value of this approach is greatly increased if data held in an existing version of a system can be migrated automatically to a subsequent version.

However, existing work on model-driven approaches has been focused upon the model: addressing the design of domain-specific modeling languages, the development of model transformation techniques, and the generation of implementation code. Little work has been done formalizing and automating the process of data migration. To be able to produce a new version of the system, quickly and easily, simply by changing the model, has significant benefits for developers and customers alike. These benefits are sharply reduced if changes to the model may have unpredictable consequences for the existing data.

In this dissertation we have outlined a possible solution: capturing the changes to data models using a language of model operations, mapping each operation to a formal specification of the corresponding data transformation, checking operations for consistency with respect to the model semantics and the existing data, and-for a specific platform-automating the process of implementation. By decoupling data instances and data models, our approach reduces the complexity of data migration activity whilst also providing an integrated standard metamodel-based development method, which is aligned with the MDA paradigm.

7.1 Research contributions

The contributions of this dissertation can be summarized into the following items:

7.1.1 Modeling evolution

To define a data model evolution language, we need to identify a modeling language and the model elements that can be evolved. Accordingly, our first step towards defining our proposed evolution language was to characterize a data metamodel as a subset of UML metamodel. Inspired by UML classification, we differentiated between two abstraction levels in our data metamodel : one level is used to describe model-level concepts and the other level is used to describe instance concepts. This differentiation allowed us the possibility to define rules of consistency between the two abstraction levels, from syntactic and semantic perspectives. We then showed how a model evolution language (in the form of a metamodel) may be derived for our characterized UML subset. Our main focus next was on the precise definition of the evolution language we have derived.

7.1.2 Precise modeling of data model evolution

In this dissertation, we showed how information systems evolution challenges can be addressed through a formal, model-driven approach. Since our main motivation is to develop a model-driven approach, one challenge we initially faced when we started using B was our inability to integrate B into a model-driven development process. The main reason of this difficulty was the lack of a B metamodel, which can be used to generate B models. We consider the B metamodel we described in Chapter 3 an important contribution that enables the integration of B into a model-driven chain of development.

As reported in [4] and explained in Chapter 5 of this dissertation, we used B-method Abstract Machine Notation (AMN) to assign semantics to data model elements. An essential aspect of this formal semantics was the characterization of consistency conditions at the syntactic and the semantic levels of abstraction. This characterization was important because it gave us a notion of correctness for our model evolution operations. With this formalization, we were able to verify data model consistency utilizing a machine aided method using theorem-provers of B-method. To give precise semantics to our model evolution operations, we used B-method Generalized Substitution Language (GSL). This formalization gave us the ability to precisely specify both our primitive and compound model edits.

7.1.3 Predicting consequences of data model evolutionary changes

Our proposed language of model evolution, properly mapped to a formal semantics in B-method can be used for the analysis of proposed evolutionary changes to data models.

One of the most important consequences to consider is consistency preservation: whether the evolution would violate data model consistency. In this work, we present consistency-preservation arguments in terms of a number of metamodel consistency constraints that may be violated during a data model evolution. Using B-method proof tool, we were able to identify what constraints can possibly be violated by each individual model edit. Subsequently, we modified the preconditions or the substitution of these edits to avoid any consistency violation. As a guarantee of consistency-preservation, all of our model edits have been proved to preserve the stated metamodel consistency constraints.

In addition, as we reported in [5], using our formalization and the B proof tool, we showed that an evolved data model could be a refactoring of a source data model, provided that one essential criteria is met : behavior preservation. As explained in Chapter 5, this criteria means that changes to the source model only affect data structure with out changing any behavior properties in the model. Since such changes account for many kinds of data model evolutionary changes that often claimed to be refactoring without a formal argument [10], we showed how the refinement proof of B-method, applied on two versions of data models can be used to provide a formal argument for such criteria.

Moreover, using the supporting framework of B-method, we showed how the applicability of a sequence of model evolution steps may be determined in advance, and used to check that a proposed evolution will indeed model consistency and data integrity. This applicability information can be fed back to system designers during the modeling activity, allowing them to see in advance how the changes they are proposing would affect the data in the existing system.

7.1.4 Generation of correct data migration programs

Using B-method refinement mechanism, we showed how operating two successive transformations on our abstract evolution specifications can be translated into an application in an executable language such as Structured Query Language (SQL), which is the implementation language we have chosen for describing data migration

implementations. As a product of formal refinement, the generated code is guaranteed to preserve various consistency and integrity notions represented by the data model.

Our refinement approach ensured that the generated SQL implementation preserves two kinds of constraints. First, constraints imposed by the data modeling language (e.g. inheritance constraints). Second, constraints defined by the data model and used to impose business rules. In our framework, both kinds of constraints are defined as invariants in the abstract data model. As the abstract model is formally refined into a relational model and then to an SQL implementation, we are certain that these invariants are preserved.

7.2 Genericity of the approach

In this section, we show that our approach can be *generic* in the sense that it can be used to treat other classical data management problems such as data integration and data warehousing, independent of the modeling environment or the database context. This is possible because, as we explained in Section 2.1, similar modeling concepts are used in most data modeling approaches, such as UML [124], ER [36], ORM [75]. Thus, an implementation that uses a representation of abstract models that includes most of those concepts should be applicable to all such environments. In addition, in this section we also show how our approach is aligned with some recent advances in the database field that depend on conceptual representations such as Ontology-Based Databases (OBD) and ontology evolution.

The basic idea underlying the genericity of our approach is to be able to use abstract models to describe main artifacts involved in the problem domain and manipulate these abstract models towards reaching a solution. This manipulation usually requires an explicit representation of mappings, which describe how two models are related to each other. This mapping representation can then be used to create a model from another model, modify an existing model or generate code, similar to the way we defined abstract evolution model between a source and a target data model to guide data migration.

For the purpose of this section, the exact choice of model representation is not important. However, there are several technical requirements on the representation of abstract models, on which the genericity of our approach depends:

1. We expect the main artifacts of the problem domain to be represented in well-formed conceptual models. If this is not the case, a reverse engineering step needs to be performed as a pre-requisite, using an approach like [30]. We also

require that the expressiveness of the representation of models to be comparable to that of UML or ER models. That is, objects can have attributes (i.e., properties), and can be related by associations (i.e., relationships with no special semantics).

2. We expect objects, properties and relationships to have types. Thus, there are (at least) three meta-levels in the picture. Using conventional metamodel terminology, we have: models; metamodels that consists of the type definitions for the objects of models; and the meta-metamodel, which is the representation language in which models and metamodels are expressed.

3. A model must contain a set of objects, each of which has an identity. By requiring that objects have identity, we can define transformations between models in terms of mappings between objects or combinations of objects.

Applications

Data integration. Data integration is a fundamental task to enable the interoperability of information systems. Data integration involves combining data from different sources and providing users with a view of this data combined together [15]. Data integration is a complex and time-consuming method due to the fact that it involves various distributed data sources and numerous stakeholders (e.g. data proponents). In particular, data sources which are generally designed for different purposes make the development of a unified data repository a challenging task. Several approaches have been proposed to overcome the challenges in the design of data integration, see [135] for a survey.

To perform data integration, it has to be determined how to select, integrate and transform the information stored in *local* data sources into a *global* data store. During the formulation of the integration mapping, possible integration conflicts have to be recognized and eliminated, like schema-level conflicts, e.g. different attribute ranges.

Our approach can be generalized to address schema-level conflicts concerning semantic and structural heterogeneity. If the schema of each data store participating in the integration can be regarded as a model and described in a well-formed abstract representation, we can express the integration mapping between these abstract models on the basis of our evolution metamodel, independent of the modeling language or modeling constructs involved. This abstract integration mapping can then be used as a basis for generating a working implementation of data integration using model-to-text transformation.

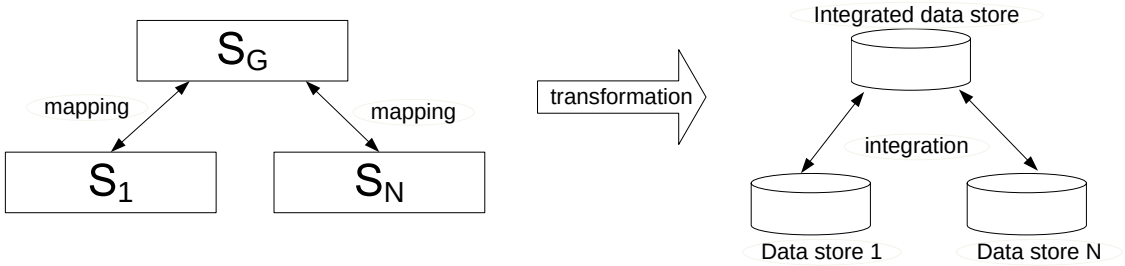


Figure 7.1: Overview of data integration problem from the perspective of our approach

An overview of how our approach may be applied to data integration problem can be seen in Figure 7.1. Assume that some local data schemata (Schema S_1 and Schema S_N) are to be integrated into a common global schema (Schema S_G). First, both local and global schemata need to be represented in a common modeling language. Afterwards, for each representation of a local schema, a mapping onto the global schema is defined. Based on these mappings, the necessary data access and data integration procedures can be generated using transformation techniques from model-driven software domain, following the main principles of the approach we proposed in this thesis.

With our approach, various target platforms can be supported. In contrast to approaches tied to a specific data model such as [96] and [104], any format of data transformation statements like SQL or Java can be generated by our method. Also, various kinds of data sources (e.g. relational databases, semi-structured information sources or flat files) can be integrated (assuming that these artifacts are represented in abstract models and that our approach is interfaced to external schema matching tools that support schema mapping).

Data warehousing. Within data warehousing scenario, ETL (Extraction-Transformation-Loading) processes are responsible for the *extraction* of data from heterogeneous operational data sources, their *transformation* (conversion, cleaning, normalization, etc.) and their *loading* into data warehouses [81]. As such, ETL processes are a key component of data warehousing because incorrect or misleading data will produce wrong business decisions, and therefore, a correct design of these processes at early stages of a data warehousing project is absolutely necessary. Despite the importance of designing the mapping of the data sources to the data warehousing

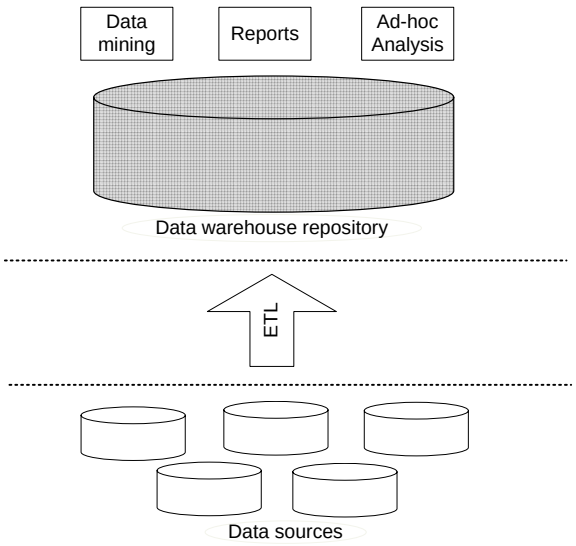


Figure 7.2: Typical data warehouse architecture, based on [91]

repositories along with any necessary constraints and transformations, there are few models that can be used by designers to this end [145, 159].

Figure 7.2 shows a typical data warehouse architecture based on [91]. Data from the operational data stores may be specified in different schemas and have to be extracted, transformed and loaded into a common data warehouse repository. In an ETL process, the data extracted from a source systems passes through a sequence of transformations before they are loaded into a data warehouse. The set of source systems that contribute data to a data warehouse is likely to vary from standalone spreadsheets to mainframe-based systems.

The model-driven approach we proposed in this thesis can be generalized to model different aspects of a data warehouse architecture such as operational data sources, the target data warehouse schema and ETL processes in an integrated manner by using an abstract modeling notation. In particular, static structure concepts of our proposed data modeling can be used to represent various aspects of source data stores and target data warehouse repository at the conceptual level.

In addition, our abstract representation of evolutionary changes from a source data model to a target data model may be used to model the relationship between a schema representing a source operational data store and another schema representing a data warehouse. More specifically, most common ETL processes such as those mentioned in [103], can be mapped into evolutionary steps in our abstract evolution representation, irrespective of the modeling notation employed. This can provide

the necessary mechanisms for an easy and quick specification of common operations required in ETL processes.

During the integration process from data sources into the data warehouse, source data may undergo a series of transformations, which may vary from simple algebraic operations or aggregations to complex procedures. In our approach, the designer can combine a long and complex transformation process into simple and small parts represented by means of an evolution model that is a materialization of evolutionary steps.

Therefore, our approach helps reduce the development time of a data warehouse, facilitates managing data repositories, data warehouse administration, and allows the designer to perform dependency analysis (i.e. to estimate the impact of a change in the data sources on the global data warehouse schema).

Ontology evolution. Gruber [72] characterizes ontology as the explicit specification of a conceptualization of domain. While there are different kinds of ontologies, they typically provide a shared/controlled vocabulary that is used to model a domain of interest using concepts with properties and relationships. In the recent past, such ontologies have been increasingly used in different domains. In the database area, ontologies have been used to facilitate a number of applications such as data exchange and integration[53]. Furthermore, Ontology-Based Database (OBDS) is a database in which an ontology and its instances are stored together [16]. Several OBDBs have been developed that use different approaches on how to store and manage the ontologies and their instances, for example [48].

Ontologies are not static but are frequently evolved to incorporate the newest knowledge of a domain or to adapt to changing application requirements. Ontology providers usually do not know which applications/users utilize their ontology. Supporting different ontology versions is an important approach to provide stability for ontology applications.

In Model-Driven Semantic Web (MDSW) [130], ontologies are represented as models derived from Ontology Definition Metamodel (ODM) [123]. The Ontology Definition Metamodel is an Object Management Group (OMG) specification to make the concepts of Model-Driven Architecture (MDA) applicable to the engineering of ontologies. This allows many problems of ontology engineering to be addressed by model-driven engineering technologies.

In this section, we focus on the problem of ontology evolution and show how elements of the framework we proposed in this thesis can be used to address aspects

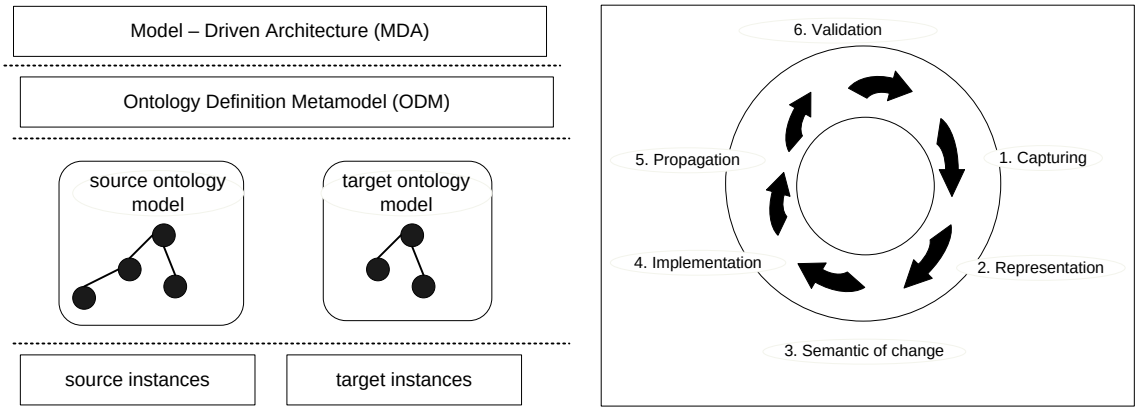


Figure 7.3: Overview of ontology evolution, from the perspective of our approach

of this problem. We analyze ontology evolution process based on our model-driven approach, and present transformation-based conceptual framework for ontology evolution. The framework uses the basic ingredients of our approach : models and transformations.

Figure 7.3 shows an overview of ontology evolution from the perspective of our approach. Based on [150], ontology evolution process is comprised of six phases: (1) change capturing, (2) change representation, (3) semantic of change, (4) change implementation, (5) change propagation, (6) change validation.

The model-driven approach we proposed in this thesis can be generalized to support the main aspects of the above ontology evolution processes. *Capturing changes* can typically be achieved using one of two approaches. In the first approach, ontology changes are explicitly specified and a new ontology version is provided. This is, in essence, the model evolution approach we outlined in this thesis. In the second approach, an ontology difference algorithm is used to determine an evolution mapping between two input ontology versions [118]. The result of this comparison can be mapped to a set of ontology changes. *Change representation* mainly refers to the granularity of changes. Our approach can support a set of simple and complex ontology changes. For example, many of ontology changes classified in [61] can be mapped to our primitive and composite evolution operations. An important consideration in the *semantic of ontology change* is to identify potential problems (inconsistencies) that the intended changes can introduce within the ontology. For example, the deletion of a concept C impacts its children and instances. Such a concern can be addressed on the basis of our consistency preservation mechanism that we outlined in Chapter 4. When the ontology is modified, the *propagation* phase requires ontology instances

to be changed to preserve consistency with the ontology. In line with our proposed induced migration approach, this can be achieved by using the change of the ontology to guide the modification of ontology instances.

Within the framework of [150], our approach does not support *change implementation* or *change validation* phases. In the former phase, the user is presented with a number of change implications and asked to choose one. The latter enables analysis of performed changes and undoing them at user's request. The semantics of our evolution operations are determined a priori and our approach does not include an 'undo' capability.

7.3 Comparison with related work

In model-driven engineering literature there many related and relevant approaches. Model weaving [57, 56] can be used to establish semantic links between models. These semantic links can be collected in a weaving model that is defined based on an extensible weaving metamodel and can be used by a model transformation tool as an input to a model transformation process to translate source models into target models based upon weaving links. Model weaving is of an immediate relevance to our work. A basic language of model operations can be formulated as an extension to the weaving metamodel. At an early stage of our work, we have investigated this idea in more details, as reported in [1]. However, following deeper investigation, the approach proved insufficient for our purposes as it does not support conditional mapping between model elements; nor does it support constraint specifications.

While we might suppose that a data model evolution can be represented as a sequence of model operations obtained automatically from a modeling tool, techniques for the model comparison and difference representation such as [94], [70] and [80] can also be relevant in the context of our work. In general, although model comparison and difference representation approaches address a similar problem : structural evolution of models, they tend to limit their scope to the structural elements of a model with no consideration to integrity constraints or well-formdness rules. In addition, these approaches are generic and do not focus on information system evolution specific requirements such as data migration.

Metamodel evolution and model co-evolution approaches such as [78], [32], [160] and [37] address the problem of metamodel evolution and model co-evolution i.e. adapting (migrating) models that conform to an older version of a metamodel to a newer version of the metamodel. [37], in close relation to our approach, proposed to

represent metamodel changes as difference models conforming to a difference metamodel to identify semi-automated countermeasures in order to co-evolve the corresponding models. While metamodel evolution approaches share the same aim of our work: reduce manual migration activities, they operate on different abstraction level (migration of M1 model elements upon M2 model changes) and focus mainly on structural evolution of generic metamodels where changes related to integrity constraints of data models are neither classified nor explicitly considered.

The generation of B formal specifications from UML diagrams has been investigated by [98, 157, 99, 33]. In particular, [33] approach is supported by U2B where only the insert operation is generated for associations. Operations on attributes are not considered. Moreover, because the considered domain being reactive systems, the semantics attached to UML diagrams is rather different from ours. [99] approach is supported by ArgoUML + B tool. None of these two approaches includes generation of code. [98] has developed an approach, supported by the UML-RSDS tool, to generate B, JAVA and SMV specifications from a subset of UML diagrams comprising class diagrams involving inheritance, state diagrams and constraints expressed in OCL. The rules used to translate class diagrams are similar to ours. The generation of SMV specifications permits to detect some intra- and inter-diagram inconsistencies. However no details are given about the formalism and the correctness of the generated code. Although these approaches and others provided solid foundations for showing how a formal method can be used to assign semantics to a graphical notation or modeling language, contrary to our approach, the main focus in these approaches was on the initial system development rather than system maintainance or evolution and the essential requirement of moving software artifacts (such as data) from one system version to the other.

7.4 Limitations and future work

7.4.1 Feedback generation

Following a proof-based approach during software design has been debated in literature . While some authors following this approach report positive experience in the domain of database design [146], model-drive engineering [109] and in other domains [8]; others find such technique, neither feasible nor justifiable [66],[161] and [63], considering the efforts and time involved. In our experience, using a proof-based approach as part of the design process proved useful, for two main reasons. First, the proof-based technique helped us complete specifications for our model edits. Due to

this activity, we can guarantee that a conformant source data model can be evolved, using our model edits, into a conformant target data model. Second, given that our approach is specified at metamodel level of design, the time and efforts involved can be justified since we can generically use these specifications to address the evolution of any model at a lower abstraction level that conforms to our data metamodel, i.e. proofs are performed once and utilized many times.

However, the approach we presented in this dissertation is a one-way approach : our model transformation rules translate from our data and evolution metamodels to B not the other way around. When the type checker or the prover of Atelier B finds an error in the specifications, the designer must be able to understand the B specifications and then has to search in evolution model to find the error. Here, we assumes that designers are familiar with such formal languages and are able to interpret messages generated by a proof tool. This assumption may not be valid in all cases as designers familiar with mathematical domains could be a minority . For the proof tool to be of value to a designer, as a future work, we plan to complete the feedback loop and complement our approach with another model transformation step to interpret messages returned by the proof tool in terms of the UML. A similar idea was reported [93] but was realized in a different context.

7.4.2 From data migration to model migration

In this dissertation, we have looked into the data migration problem from a data model evolution perspective : how can we capture evolutionary changes on data models and how to map those into corresponding migration rules to transform data instances according to data model changes. However, not only may models evolve, but so also may the languages in which the models are expressed. Like data models, modeling languages are subject to evolution due to changing requirements, fixing errors and keeping up with technological progress [59].

An important application of model-driven development techniques where metamodels play a central role is Domain-Specific Modeling (DSM) [92]. DSM requires metamodels that allow a domain expert to capture key concepts of a domain of interest. However, these metamodels rarely defines the domain completely, and must often be evolved. Metamodel evolution can affect model integrity. For example, when a metamodel concept is removed, any models that use the removed concept no longer conform to the metamodel. Due to metamodel evolution, existing models may no longer conform to the evolved metamodel and thus need to be migrated to reestablish conformance to the evolved metamodel.

An important future work item would be to show how our data model evolution approach, set out in this dissertation, can be extended to include updates to meta-models and domain-specific languages used for data modeling language as well as to updates to the data model (language evolution versus model evolution).

7.4.3 Predicting consequences of evolution on behavior properties

In this dissertation, we are concerned with changes that affect the form or validity of the data in the system: we did not consider changes to behavioral properties. Following an evolution, we may be interested in the question of whether particular operations or workflows existed in the source model can still be triggered in the target model and deliver the same goal. To be able to address this question, we need to be able to compare the source model and the target model with respect to their behavior.

Information about the availability and effect of operations and worklows can be drawn from UML state diagrams. An operation may be considered available when suitably labeled transition from the current state is complete. i.e. the guard of the operation is satisfied.

Our approach may be extended to address not only the consistency of data, but also the applicability or availability of particular operations or workflows. If an operation or workflow is associated with a particular precondition, then we may map this precondition to an additional constraint in AMN, and check to see whether its truth or falsity would be affected by the proposed data model evolution.

7.5 Conclusion

As development in most information and data-intensive systems is not a one-off activity, these systems will need to be updated, repeatedly, in response to changes in context and requirements. At each update, the data held within the old system must be transferred to the new implementation. If changes have been made to the data model, then careful consideration must be given to the transformation and representation of existing data—which may be of considerable value and importance to the organization.

In this dissertation, we showed how to address the challenges of systems evolution through a formal, model-driven approach. Using the Unified Modeling Language (UML) as an example, we showed how a sequence of proposed changes to a system can themselves be represented as a model. We show how such a model may be

used as the basis for the automatic generation of a corresponding data migration function, in the standard Structured Query Language (SQL). We showed also how a formal representation of the model in the Abstract Machine Notation (AMN) of the B-Method allows us to check that this function is Applicable—that the migrated data would fit within the constraints of the new system.

Our approach offers an opportunity to consider the question of information system evolution and data migration, in detail, at the design stage. It demonstrated how this may be achieved through the definition of formal semantics for an evolution modeling language and allows us to verify that a proposed change is consistent with representational and semantic constraints, in advance of implementation.

Bibliography

[1] Mohammed Aboulsamh and Jim Davies. Towards a model-driven approach to information systems evolution. In William Song, Shenghua Xu, and Changxuan Wan, editors, *Information Systems Development*, pages 269–280. Springer New York, 2011.

[2] Mohammed A. Aboulsamh, Edward Crichton, Jim Davies, and James Welch. Model-driven data migration. In Juan Trujillo, Gillian Dobbie, Hannu Kangassalo, Sven Hartmann, Markus Kirchberg, Matti Rossi, Iris Reinhartz-Berger, Esteban Zimányi, and Flavius Frasincar, editors, *ER Workshops*, volume 6413 of *Lecture Notes in Computer Science*, pages 285–294. Springer, 2010.

[3] Mohammed A. Aboulsamh and Jim Davies. A metamodel-based approach to information systems evolution and data migration. In Jon Hall, Hermann Kaindl, Luigi Lavazza, Georg Buchgeher, and Osamu Takaki, editors, *ICSEA*, pages 155–161. IEEE Computer Society, 2010.

[4] Mohammed A. Aboulsamh and Jim Davies. A formal modeling approach to information systems evolution and data migration. In Terry A. Halpin, Selmin Nurcan, John Krogstie, Pnina Soffer, Erik Proper, Rainer Schmidt, and Ilia Bider, editors, *BMMDS/EMMSAD*, volume 81 of *Lecture Notes in Business Information Processing*, pages 383–397. Springer, 2011.

[5] Mohammed A. Aboulsamh and Jim Davies. Specification and verification of model-driven data migration. In Ladjel Bellatreche and Filipe Mota Pinto, editors, *MEDI*, volume 6918 of *Lecture Notes in Computer Science*, pages 214–225. Springer, 2011.

[6] Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[7] Aditya Agrawal. Graph rewriting and transformation (GReAT): A solution for the model integrated computing (MIC) bottleneck. In *Automated Software Engineering (ASE)*, pages 364–368. IEEE Computer Society, 2003.

[8] Idir Aït-Sadoune and Yamine Aït Ameur. A proof based approach for modelling and verifyingweb services compositions. In *International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 1–10. IEEE Computer Society, 2009.

[9] Scott Ambler. *Agile Database Techniques*. John Wiley and Sons , October 2003.

[10] Scott W. Ambler and Pramodkumar J. Sadalage. *Refactoring Databases : Evolutionary Database Design (Addison Wesley Signature Series)*. Addison-Wesley Professional, March 2006.

[11] Apache Derby. http://db.apache.org/derby/, Accessed February 2012.

[12] Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A model-driven process for engineering a toolset for a formal method. *Software Practical Experience*, 41(2):155–166, 2011.

[13] Atelier B. Atelier proof tool, 2012.

[14] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In Umeshwar Dayal and Irving L. Traiger, editors, *SIGMOD Conference*, pages 311–322. ACM Press, 1987.

[15] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv.*, 18(4):323–364, December 1986.

[16] Ladjel Bellatreche, Yamine Aït Ameur, and Chedlia Chakroun. A design methodology of ontology based database applications. *Logic Journal of the IGPL*, 19(5):648–665, 2011.

[17] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML class diagrams. *Artif. Intell.*, 168(1):70–118, October 2005.

[18] Jean Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[19] Michael Blaha. *Patterns of Data Modeling*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2010.

[20] Michael Blaha and William Premerlani. *Object-oriented modeling and design for database applications*. Prentice-Hall, 1997.

[21] Marko Boger, Thorsten Sturm, and Per Fragemann. Refactoring browser for UML. In *Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 366–377, London, UK, UK, 2003. Springer-Verlag.

[22] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide(2nd Edition)*. Addison-Wesley Professional, 2005.

[23] Behzad Bordbar, Dirk Draheim, Matthias Horn, Ina Schulz, and Gerald Weber. Integrated model-based software development, data access, and data migration. In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 382–396. Springer, 2005.

[24] Rafael Magalhães Borges and Alexandre Cabral Mota. Integrating UML and formal methods. *Electron. Notes Theor. Comput. Sci.*, 184:97–112, July 2007.

[25] Artur Boronat and José Meseguer. An algebraic semantics for mof. In *Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering*, FASE'08/ETAPS'08, pages 377–391, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] Philippe Brèche. Advanced principles for changing schemas of object databases. In *Proceedings of the 8th International Conference on Advances Information System Engineering*, pages 476–495. Springer-Verlag, 1996.

[27] G. H. W. M. Bronts, S. J. Brouwer, C. L. J. Martens, and Henderik Alex Proper. A unifying object role modeling theory. *Inf. Syst.*, 20(3):213–235, 1995.

[28] Stephen D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[29] Manfred Broy and María Victoria Cengarle. UML formal semantics: lessons learned. *Software and System Modeling*, 10(4):441–446, 2011.

[30] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. MoDisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 173–174, New York, NY, USA, 2010. ACM.

[31] Barrett R. Bryant, Jeff Gray, Marjan Mernik, Peter J. Clarke, Robert B. France, and Gabor Karsai. Challenges and directions in formalizing the semantics of modeling languages. *Comput. Sci. Inf. Syst.*, 8(2):225–253, 2011.

[32] Erik Burger and Boris Gruschko. A change metamodel for the evolution of mof-based metamodels. In Gregor Engels, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung*, volume 161 of *LNI*, pages 285–300. GI, 2010.

[33] Michael Butler and Colin Snook. Uml-b: Formal modeling and design aided by uml. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, January 2006.

[34] Coral Calero, Francisco Ruiz, Aline Lúcia Baroni, Fernando Brito Abreu, and Mario Piattini. An ontological approach to describe the sql: 2003 object-relational features. *Computer Standards & Interfaces*, 28(6):695–713, 2006.

[35] Stefano Ceri, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. Automatic generation of production rules for integrity maintenance. *ACM Transactions on Database Systems*, 19:367–422, September 1994.

[36] Peter P. Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[37] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *Enterprise Distributed Object Computing Conference (EDOC)*, pages 222–231. IEEE Computer Society, 2008.

[38] Tony Clark and Andy Evans. Foundations of the unified modeling language. In *Proceedings of the 2nd BCS-FACS conference on Northern Formal Methods*, (2FACS'97), pages 6–6, Swinton,UK, 1997. British Computer Society.

[39] Kajal T. Claypool, Jing Jin, and Elke A. Rundensteiner. Serf: Schema evalution through an extensible, re-usable and flexible framework. In Georges Gardarin, James C. French, Niki Pissinou, Kia Makki, and Luc Bouganim, editors, *International Conference on Information and Knowledge Management(CIKM)*, pages 314–321. ACM, 1998.

[40] Kajal T. Claypool, Elke A. Rundensteiner, and George T. Heineman. Rover: flexible yet consistent evolution of relationships. *Data Knowl. Eng.*, 39:27–50, October 2001.

[41] Anthony Cleve, Anne-France Brogneaux, and Jean-Luc Hainaut. A conceptual approach to database applications evolution. In *Proceedings of the 29th international conference on Conceptual modeling*, ER'10, pages 132–145. Springer-Verlag, 2010.

[42] Anthony Cleve, Jean Henrard, Didier Roland, and Jean-Luc Hainaut. Wrapper-based system evolution application to codasyl to relational migration. In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pages 13–22, Washington, DC, USA, 2008. IEEE Computer Society.

[43] Anthony Cleve, Tom Mens, and Jean-Luc Hainaut. Data-intensive system evolution. *IEEE Computer*, 43(8):110–112, 2010.

[44] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. Viatra - visual automated transformations for formal verification and validation of UML models. In *Automated Software Engineering (ASE)*, pages 267–270. IEEE Computer Society, 2002.

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

[46] Jim Davies, Charles Crichton, Edward Crichton, David Neilson, and Ib Holm Sørensen. Formality, evolution, and model-driven software engineering. *Electr. Notes Theor. Comput. Sci.*, 130:39–55, 2005.

[47] Jim Davies, James Welch, Alessandra Cavarra, and Edward Crichton. On the generation of object databases using booster. In *International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 249–258. IEEE Computer Society, 2006.

[48] Hondjack Dehainsala, Guy Pierra, and Ladjel Bellatreche. OntoDB: an ontology-based database for data intensive applications. In *Proceedings of the 12th international conference on Database systems for advanced applications*, DASFAA'07, pages 497–508, Berlin, Heidelberg, 2007. Springer-Verlag.

[49] Christine Delcourt and Roberto Zicari. The design of an integrity consistency checker (ICC) for an object-oriented database system. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 97–117, London, UK, 1991. Springer-Verlag.

[50] Cecilia Delgado, José Samos, and Manuel Torres. Primitive operations for schema evolution in odmg databases. In Dimitri Konstantas, Michel Léonard, Yves Pigneur, and Shushma Patel, editors, *Object-Oriented Information Systems (OOIS)*, volume 2817 of *Lecture Notes in Computer Science*, pages 226–237. Springer, 2003.

[51] Birgit Demuth and Heinrich Hußmann. Using UML/OCL constraints for relational database design. In Robert B. France and Bernhard Rumpe, editors, *UML*, volume 1723 of *Lecture Notes in Computer Science*, pages 598–613. Springer, 1999.

[52] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., 1976.

[53] Li Dong and Huang Linpeng. A framework for ontology-based data integration. In *Proceedings of the 2008 International Conference on Internet Computing in Science and Engineering*, ICICSE '08, pages 207–214, Washington, DC, USA, 2008. IEEE Computer Society.

[54] Marina Egea and Vlad Rusu. Formal executable semantics for conformance in the mde framework. *Innovations in Systems and Software Engineering (ISSE)*, 6(1-2):73–81, 2010.

[55] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems (4th Edition)*. Benjamin Cummings, Redwood City, Calif., USA, 2003.

[56] Marcos Didonet Del Fabro and Patrick Valduriez. Semi-automatic model integration using matching transformations and weaving models. In Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo, editors, *ACM Symposium on Applied Computing (SAC)*, pages 963–970. ACM, 2007.

[57] Marcos Didonet Del Fabro and Patrick Valduriez. Towards the efficient development of model transformations using model weaving and matching transformations. *Software and System Modeling*, 8(3):305–324, 2009.

[58] David Faitelson, James Welch, and Jim Davies. From predicates to programs: The semantics of a method language. *Electron. Notes Theor. Comput. Sci.*, 184:171–187, 2007.

[59] Jean-Marie Favre. Languages evolve too! changing the software time scale. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 33–44, Washington, DC, USA, 2005. IEEE Computer Society.

[60] Fabrizio Ferrandina, Thorsten Meyer, Roberto Zicari, Guy Ferran, and Joëlle Madec. Schema and database evolution in the o2 object database system. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB*, pages 170–181. Morgan Kaufmann, 1995.

[61] Giorgos Flouris, Dimitris Manakanatas, Haridimos Kondylakis, Dimitris Plexousakis, and Grigoris Antoniou. Ontology change: Classification and survey. *Knowl. Eng. Rev.*, 23(2):117–152, June 2008.

[62] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[63] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[64] Robert B. France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, 1998.

[65] Stephen J. Garland, John V. Guttag, and James J. Horning. An overview of larch. In Peter E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 329–348. Springer, 1993.

[66] Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and System Modeling*, 4(4):386–398, 2005.

[67] Martin Gogolla and Arne Lindow. Transforming Data Models with UML. In *Knowledge Transformation for the Semantic Web*, pages 18–33. 2003.

[68] Joseph A. Goguen, Claude Kirchner, Hélène Kirchner, Aristide Mégrelis, José Meseguer, and Timothy C. Winkler. An introduction to obj 3. In Stéphane Kaplan and Jean-Pierre Jouannaud, editors, *CTRS*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer, 1987.

[69] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent UML refactorings. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *The Unified Modeling Language (UML) Modeling Languages and Applications*, volume 2863 of *Lecture Notes in Computer Science*, pages 144–158. Springer, 2003.

[70] Jeff Gray, Yuehua Lin, and Jing Zhang. Automating change evolution in model-driven engineering. *IEEE Computer*, 39(2):51–58, 2006.

[71] D. Gries. *The Science of Programming*. Springer Verlag, New York, USA, 1981.

[72] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, June 1993.

[73] Jean-Luc Hainaut, Anthony Cleve, Jean Henrard, and Jean-Marc Hick. *Migration of Legacy Information Systems*. 2008.

[74] Terry Halpin. Object-role modeling (ORM/NIAM). *Handbook on Architectures of Information Systems, Springer Berlin Heidelberg*, pages 81–103, 2006.

[75] Terry Halpin and Tony Morgan. *Information Modeling and Relational Databases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[76] David Harel and Bernhard Rumpe. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer*, 37(10):64–72, 2004.

[77] Michael Hartung, James F. Terwilliger, and Erhard Rahm. Recent advances in schema and ontology evolution. In Zohra Bellahsene, Angela Bonifati, and Erhard Rahm, editors, *Schema Matching and Mapping*, pages 149–190. Springer, 2011.

[78] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Jürgens. Automatability of coupled evolution of metamodels and models in practice. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 645–659. Springer, 2008.

[79] http://www.eclipse.org/datatools/. Eclipse Data Tools Project, 2012.

[80] http://www.eclipse.org/modeling/emft/?project=compare. Eclipse Compare Project, 2009.

[81] W. H. Inmon. *Building the data warehouse*. QED Information Sciences, Inc., Wellesley, MA, USA, 1992.

[82] International Standards Organization (ISO). Vienna development method (VDM) — specification language. *ISO/Information Technology/IEC 13817-1:1996*, 1996.

[83] International Standards Organization (ISO). Z formal specification notation — syntax, type system and semantics. *ISO/Information Technology/IEC 13568:2002*, 2002.

[84] International Standards Organization (ISO). Database languages SQL: Parts 1 to 4 and 9 to 14. *ISO/Information Technology/IEC 9075-1:2003 to 9075-14:2003*, 2003.

[85] Jonathan Jacky. *The Way of Z*. Cambridge University Press, Cambridge, UK, 1997.

[86] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-oriented software engineering - a use case driven approach*. Addison-Wesley, 1992.

[87] Yanbing Jiang, Weizhong Shao, Lu Zhang, Zhiyi Ma, Xiangwen Meng, and Haohai Ma. On the classification of UML's metamodel extension mechanism. In Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors, *UML*, volume 3273 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2004.

[88] Edward Crichton Jim Davies, James Welch. Model-driven engineering of information systems: 10 years and 1000 versions. *Science of Computer Programming, Special Issue on Success Stories in Model Driven Engineering*, (Submitted for publication).

[89] Cliff Jones. *Systematic Software Development Using VDM*. Prentice Hall, 2nd edition, 1991.

[90] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

[91] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2002.

[92] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1st edition, 2008.

[93] Leonid Kof and Birgit Penzenstadler. From requirements to models: Feedback generation as a result of formalization. In Haralambos Mouratidis and Colette Rolland, editors, *Conference on Advanced Information Systems Engineering (CAiSE)*, volume 6741 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2011.

[94] Dimitrios S. Kolovos. Establishing correspondences between models with the epsilon comparison language. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *ECMDA-FA*, volume 5562 of *Lecture Notes in Computer Science*, pages 146–157. Springer, 2009.

[95] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The epsilon object language (EOL). In Arend Rensink and Jos Warmer, editors, *ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2006.

[96] Stefan Kurz, Michael Guppenberger, and Burkhard Freitag. A UML profile for modeling schema mappings. In John F. Roddick, V. Richard Benjamins, Samira Si-Said Cherfi, Roger H. L. Chiang, Christophe Claramunt, Ramez Elmasri, Fabio Grandi, Hyoil Han, Martin Hepp, Miltiadis D. Lytras, Vojislav B. Misic, Geert Poels, Il-Yeol Song, Juan Trujillo, and Christelle Vangenot, editors, *ER (Workshops)*, volume 4231 of *Lecture Notes in Computer Science*, pages 53–62. Springer, 2006.

[97] Ralf Lämmel. Coupled Software Transformations (Extended Abstract). In *First International Workshop on Software Evolution Transformations*, November 2004.

[98] Kevin Lano, David Clark, and Kelly Androutsopoulos. UML to B: Formal verification of object-oriented models. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods (IFM)*, volume 2999 of *Lecture Notes in Computer Science*, pages 187–206. Springer, 2004.

[99] Hung Ledang and Jeanine Souquières. Modeling class operations in B: Application to UML behavioral diagrams. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 289–. IEEE Computer Society, 2001.

[100] Barbara Staudt Lerner. A model for compound type changes encountered in schema evolution. *ACM Trans. Database Syst.*, 25(1):83–127, 2000.

[101] Barbara Staudt Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 67–76, New York, NY, USA, 1990. ACM.

[102] B-Core UK Limited. The B-toolkit., 2009.

[103] Sergio Luján-Mora, Panos Vassiliadis, and Juan Trujillo. Data mapping diagrams for data warehouse design with UML. In Paolo Atzeni, Wesley W. Chu, Hongjun Lu, Shuigeng Zhou, and Tok Wang Ling, editors, *ER*, volume 3288 of *Lecture Notes in Computer Science*, pages 191–204. Springer, 2004.

[104] Sanjay Madria, Kalpdrum Passi, and Sourav Bhowmick. An XML schema integration and query mechanism system. *Data Knowl. Eng.*, 65(2):266–303, May 2008.

[105] Amel Mammar and Régine Laleau. A formal approach based on UML and B for the specification and development of database applications. *Autom. Softw. Eng.*, 13(4):497–528, 2006.

[106] Amel Mammar and Régine Laleau. From a B formal specification to an executable code: application to the relational database domain. *Information & Software Technology*, 48(4), 2006.

[107] Rafael Marcano-Kamenoff and Nicole Lévy. Transformation rules of OCL constraints into B formal expressions. In *CSDUML'2002, Workshop on critical*

systems development with UML. 5th International Conference on the Unified Modeling Language, Dresden, Germany, September 2002.

[108] Slavisa Markovic and Thomas Baar. Refactoring OCL annotated UML class diagrams. *Software and Systems Modeling*, 7(1), 2008.

[109] Tiago Massoni, Rohit Gheyi, and Paulo Borba. A framework for establishing formal conformance between object models and object-oriented programs. *Electr. Notes Theor. Comput. Sci.*, 195:189–209, 2008.

[110] Brian Matthews and Elvira Locuratolo. Formal development of databases in asso and b. In *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I - Volume I*, FM '99, pages 388–410, London, UK, UK, 1999. Springer-Verlag.

[111] Tom Mens and Tom Tourwe. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.

[112] Thomas O. Meservy and Kurt D. Fenstermacher. Transforming software development: An MDA road map. *IEEE Computer*, 38(9):52–58, September 2005.

[113] Hyun Jin Moon, Carlo Curino, MyungWon Ham, and Carlo Zaniolo. Prima: archiving and querying historical data with evolving schemas. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *SIGMOD Conference*, pages 1019–1022. ACM, 2009.

[114] Carroll Morgan. *Programming from specifications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[115] Robert J. Muller. *Database Design for Smarties: Using UML for Data Modeling*. Morgan Kaufmann; 1st edition, 1999.

[116] Eric J. Naiburg and Robert A. Maksimchuk. *UML for Database Design*. Addison-Wesley Professional, 2001.

[117] Shamkant B. Navathe. Evolution of data modeling for databases. *Commun. ACM*, 35:112–123, September 1992.

[118] Natalya F. Noy, Abhita Chugh, William Liu, and Mark A. Musen. A framework for ontology evolution in collaborative environments. In *Proceedings of the 5th international conference on The Semantic Web*, ISWC'06, pages 544–558, Berlin, Heidelberg, 2006. Springer-Verlag.

[119] Object Management Group. UML profile for CORBA, 2001.

[120] Object Management Group. UML 2.0 superstructure specification, 2005.

[121] Object Management Group. OCL specifications, version 2, 2006.

[122] Object Management Group. OMG's MetaObject Facility (MOF) version 2.0, retrieved february 09, 2011, 2006. http://www.omg.org/spec/MOF/2.0.

[123] Object Management Group. Ontology definition metamodel (ODM), 2009.

[124] Object Management Group. Unified Modeling Language (UML), Infrastructure, Version 2.2, 2009. http://www.omg.org/docs/formal/09-02-04.pdf.

[125] Object Management Group. MDA guide version 1.0.1, June 2003.

[126] Antoni Olive. *Conceptual Modeling of Information Systems.* Springer-Verlag Berlin Heidelberg, 2007.

[127] William F. Opdyke. Refactoring: A program restructuring aid in designing object-oriented application frameworks, phd thesis, 1992.

[128] Richard F. Paige, Phillip J. Brooke, and Jonathan S. Ostroff. Metamodel-based model conformance and multiview consistency checking. *ACM Transactions on Software Engineering and Methodologies*, 16(3):11, 2007.

[129] George Papastefanatos, Fotini Anagnostou, Yannis Vassiliou, and Panos Vassiliadis. Hecataeus: A what-if analysis tool for database schema evolution. In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pages 326–328, Washington, DC, USA, 2008. IEEE Computer Society.

[130] Fernando Silva Parreiras, Steffen Staab, and Andreas Winter. On marrying ontological and metamodeling technical spaces. In Ivica Crnkovic and Antonia Bertolino, editors, *6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/SIGSOFT FSE*, pages 439–448. ACM, 2007.

[131] D. Jason Penney and Jacob Stein. Class modification in the GemStone object-oriented DBMS. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 111–117, New York, NY, USA, 1987. ACM.

[132] Stephan Philippi. Model driven generation and testing of object-relational mappings. *Journal of Systems and Software*, 77(2):193 – 207, 2005.

[133] Iman Poernomo. A type theoretic framework for formal metamodeling. In *Architecting Systems with Trustworthy Components*, pages 262–298, 2004.

[134] Anne Pons and Rudolf K. Keller. Schema evolution in object databases by catalogs. In *Proceedings of the 1997 international conference on International database engineering and applications symposium*, IDEAS'97, pages 368–376. IEEE Computer Society, 1997.

[135] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, December 2001.

[136] Awais Rashid and Peter Sawyer. A database evolution taxonomy for object-oriented databases: Research articles. *Journal of Software Maintainance and Evolution*, 17(2):93–141, March 2005.

[137] Awais Rashid, Peter Sawyer, and Elke Pulvermüller. A flexible approach for instance adaptation during class versioning. In Klaus R. Dittrich, Giovanna Guerrini, Isabella Merlo, Marta Oliva, and Elena Rodríguez, editors, *Objects and Databases*, volume 1944 of *Lecture Notes in Computer Science*, pages 101–113. Springer, 2000.

[138] Roberto V. Zicari. How good is UML for Database Design? Interview with Michael Blaha, retrieved august 01, 2011, 2011. http://www.odbms.org/blog/2011/07.

[139] Donald Bradley Roberts. *Practical Analysis for Refactoring, PhD thesis*. University of Illinois at Urbana-Champaign, 1999.

[140] James E. Rumbaugh, Michael R. Blaha, William J. Premerlani, Frederick Eddy, and William E. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[141] Steve Schneider. *The B-Method: An Introduction*. Palgrave Macmillan, 2001.

[142] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[143] Graeme Paul Smith. *The Object-Z Specification Language. Advances in Formal Methods*. Kluwer Academic Publishers, 2000.

[144] Colin Snook and Michael Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15:92–122, January 2006.

[145] Xudong Song, Xiaolan Yan, and Liguo Yang. Design ETL metamodel based on UML profile. In *Proceedings of the 2009 Second International Symposium on Knowledge Acquisition and Modeling - Volume 03*, KAM '09, pages 69–72, Washington, DC, USA, 2009. IEEE Computer Society.

[146] David Spelt and Susan Even. A theorem prover-based analysis tool for object-oriented databases. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1579 of *Lecture Notes in Computer Science*, pages 375–389. Springer, 1999.

[147] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.

[148] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional, 2008.

[149] Susan Stepney, Rosalind Barden, and David Cooper. *Object orientation in Z, Workshops in Computing*. Springer, 1992.

[150] Ljiljana Stojanovic, Alexander Maedche, Boris Motik, and Nenad Stojanovic. User-driven ontology evolution management. In *Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*, EKAW '02, pages 285–300, London, UK, UK, 2002. Springer-Verlag.

[151] Gerson Sunye, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 134–148, London, UK, 2001. Springer-Verlag.

[152] Antero Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28:438–479, September 1996.

[153] B. Thalheim. *Fundamentals of Entity-relationship Modeling*. Springer-Verlag Berlin and Heidelberg GmbH Co. KG, 1999.

[154] The Object Management Group. Common Warehouse Metamodel (CWM) Specification, 2003.

[155] The Object Management Group. Software Process Engineering Metamodel Specification (SPEM), 2005.

[156] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Automated Software Engg.*, 8(1):89–120, 2001.

[157] Helen Treharne. Supplementing a UML Development Process with B. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *International Symposium of Formal Methods Europe FME*, volume 2391 of *Lecture Notes in Computer Science*, pages 568–586. Springer, 2002.

[158] Can Türker and Michael Gertz. Semantic integrity support in SQL:1999 and commercial (object-)relational database management systems. *The VLDB Journal*, 10:241–269, December 2001.

[159] Panos Vassiliadis, Alkis Simitsis, and Spiros Skiadopoulos. Conceptual modeling for ETL processes. In Il-Yeol Song and Dimitri Theodoratos, editors, *DOLAP 2002, ACM Fifth International Workshop on Data Warehousing and OLAP, November 8, 2002, McLean, VA, Proceedings*, pages 14–21. ACM, 2002.

[160] Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, pages 600–624, 2007.

[161] Chris Wallace. Using Alloy in process modeling. *Information & Software Technology*, 45(15):1031–1043, 2003.

[162] James Welch, David Faitelson, and Jim Davies. Automatic maintenance of association invariants. *Software and System Modeling*, 7(3):287–301, 2008.

[163] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, 1990.

[164] J. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Series in Computer Science. Prentice Hall International, 1996.

[165] Alexandre V. Zamulin. An object algebra for the ODMG standard. In *Proceedings of the 6th East European Conference on Advances in Databases and Information Systems*, ADBIS '02, pages 291–304, London, UK, UK, 2002. Springer-Verlag.

[166] Jing Zhang, Yuehua Lin, and Jeff Gray. Generic and domain-specific model refactoring using a model transformation engine. In *Volume II of Research and Practice in Software Engineering*, pages 199–218. Springer, 2005.

Appendix A

Implementation and case study

Our focus in this chapter is to demonstrate how, using the open source Eclipse development platform [1] , we can realize a solution for data model evolution. Such a solution can be used by information system designers in performing data model updates, analyzing some of the consequences of their updates and subsequently migrating the data.

Overview

Our Eclipse-based solution is developed according to the principles of the approach we presented in previous chapters. Figure A.1 shows an overview of our proposed model-driven approach that is detailed in subsequent sections. In our approach, changes to a UML data model are specified based on a language of model operations defined in the form of an evolution metamodel. Using model transformation, these model operations are given semantics in B-method to precisely specify their applicability and intended effect on data instances. The B-method specifications of model operations represent an abstract program that can be formally checked prior to transformation into a concrete programming notation such as Standard Query Language (SQL).

Our approach essentially consists of series of model transformation [142] across a number of MOF-based metamodels. Each metamodel is used to define a structure and well-formedness rules that must be satisfied (conformed to) by models at the lower abstraction level. By using Eclipse as a development environment, we have defined these metamodels using the Eclipse Modeling Framework (EMF) [2][148]. Below we explain the main components of our prototypical implementation.

[1] http://www.eclipse.org

[2] http://www.eclipse.org/modeling/emf

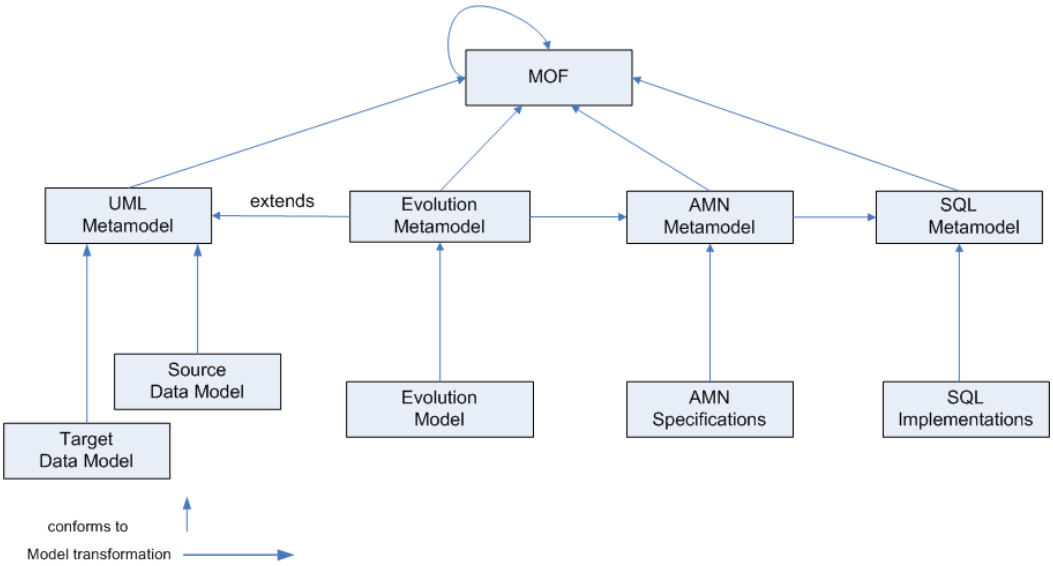


Figure A.1: Overview of the implementation approach

Evolving data models

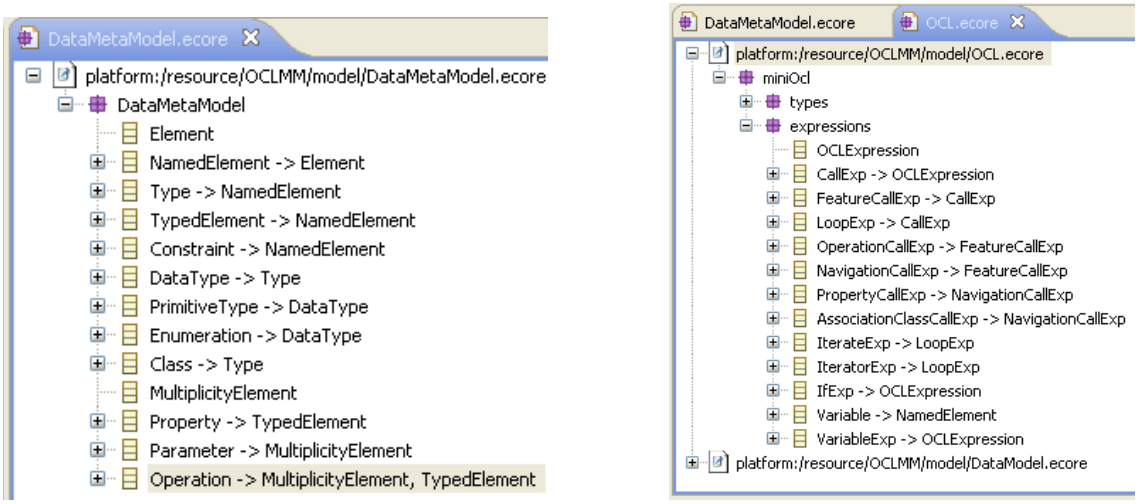
Data modeling with UML and OCL

Following the main principles outlined in Section 4.2, in our current implementation using EMF, we defined two metamodels: one representing the subset of UML and one representing the Object Constraint Language (OCL), as can be seen in Figure A.2 below .

These metamodels can be instantiated into data models similar to the Employee Information System data model. Figure A.3 below shows a fragment of how these metamodels can be instantiated. The instance model consists of classes, attributes, associations and constraints. Of particular interest in this figure is the way an OCL constraint instantiates an OCL metamodel.

When expressing a constraint as an instance of the OCL metamodel, the body of the constraint can be regarded as a binary tree where each node represents an instance of a metaclass of the OCL metamodel.

We show in Fig. A.3 the constraint C2 as an instance of the OCL metamodel. In this example, the `forAll` iterator (represented as an instance of the metaclass *IteratorExp*) is the root of the tree. The left child of the root is the source of the iterator. In its turn, this left child is represented as an instance of the *PropertyCallExpression*-metaclass corresponding to the navigation through the association end *employees*. Its source is the access to the *self* variable. The right child of the root is the body of



(a) UML Metamodel, equivalent to Figure 4.2

(b) OCL Metamodel

Figure A.2: A fragment of UML and OCL Metamodels defined in Eclipse Modeling Framework (EMF)

the iterator expression. The root of this right subtree is the operation $\geq$ represented as an instance of the *OperationCallExp* metaclass. This node has two children. The first one is its source, an access to the attribute *salary* (with a last child representing the access to the `e` variable). The second one has the same structure, an access to the *minSalary* attribute followed by an access to the *self* variable.

Editing data models

In our approach, according to our evolution metamodel presented and explained in Section 4.3, each change to a data model is an operation on a model instance—adding, modifying, or deleting a model element—and thus our evolutionary steps correspond to operations at the metamodel level. Accordingly we have extended our UML subset that we use for data modeling with classes of model operations, introducing add, modify, and delete operations for each class of model element and explicitly defining how a target data model is produced from a source data model.

To be able to include operations corresponding to more complex model evolution steps, such as the inlining of an existing class, and the introduction of an association class, we have introduced combinator for our evolution metamodel, allowing us to compose changes both sequentially (;) and in iteration. In addition, we use an *OclExpression* to describe the initial values of a new or modified attribute or association.

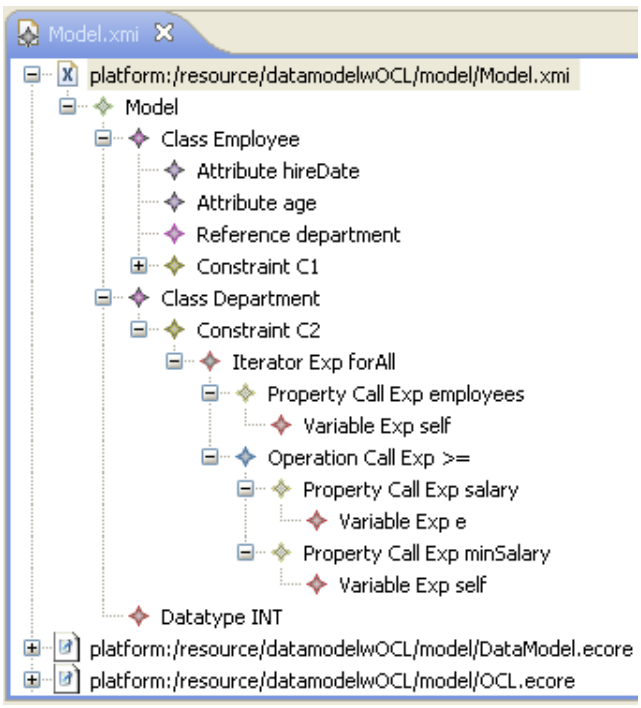


Figure A.3: data model instantiated, equivalent to Figure 4.5

This expression may refer to other attributes or associations in the current model. For more details on the above metamodel, please refer to Section 4.3.

```
import 'platform:/resource/org.example.dataModel/model/DataModel.ecore' as dmodel

Model       ::= EvolutionModel ID editing SourceModel Operation ";"
SourceModel ::= model UriID ";"
Operation   ::= AddClass | DeleteClass | AddPropertyWithValue |…|InlineClass|…";"
AddClass    ::= addClass  "(" ID ")" ";"
DeleteClass ::= deleteClass "(" dmodel::Class ")" ";"
AddPropertyWithValue::= addPropertyWithValue "(" dmodel::Class "," ID ":" Type
                        "," Multiplicity "," "<" OCLExpression ">" ")" ";"
InlineClass::= inlineClass "(" dmodel::Class "," dmodel::Class ","
               dmodel::Property ")" ";"
…
Type::= String | Integer | Boolean | dmodel::Class
ID  ::= ("a".."z"|"A".."Z"|"_") ("a".."z"|"A".."Z"|"_"|"0".."9")*
```

Figure A.4: Data Model Evolution Grammar (excerpt)

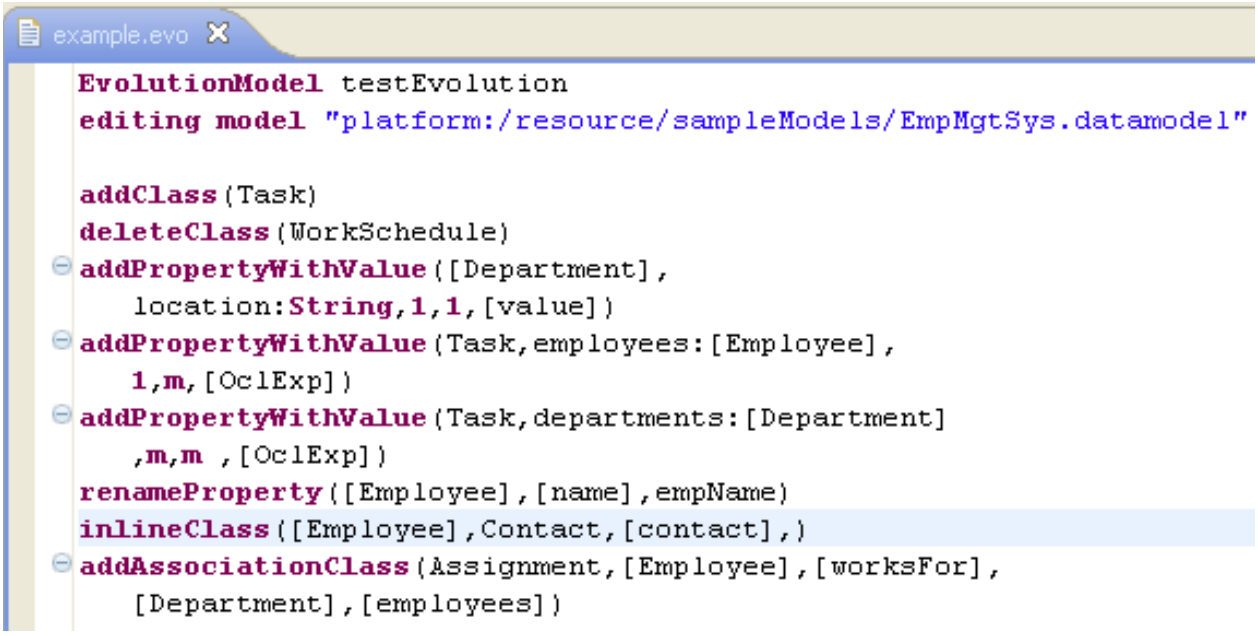


Figure A.5: Data Model Editor

Concrete syntax

To derive a concrete syntax from our evolution metamodel, we have followed the rules defined by [12] on how to derive a context-free EBNF grammar from a metamodel. Figure A.4 shows an excerpt of the EBNF version of the grammar we have written using Xtext by openArchitectureWare [3]. Xtext is an open-source language development framework that enables the generation of various language artifacts based on concrete syntax. In the figure, the [dmodel::Class] represents a cross-reference to a specific element of the source data model being edited (e.g. Class). Based on the grammar rules above, using Xtext, we have generated the editor shown in Figure A.5.

Referencing a specific data model, written in our UML subset, an information system designer can use this editor to update data model elements, adding , deleting or modifying model components. The editor provides standard features such as keyword highlighting, auto-completion and content assistance. Models written in the editor can be parsed as instances of our evolution metamodel as can be seen in Figure A.6 below.

Using model transformation, this evolution model can be transformed into an SQL model, suitable for migrating data instances to the target (evolved) model. However,

[3]http://www.eclipse.org/Xtext

to ensure that such data migration is successful, we first transform the evolution model into a a model in B-method.

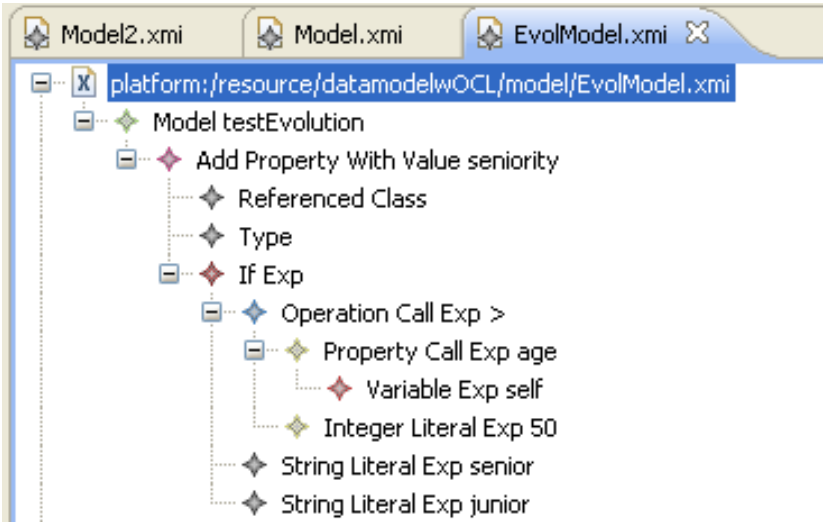


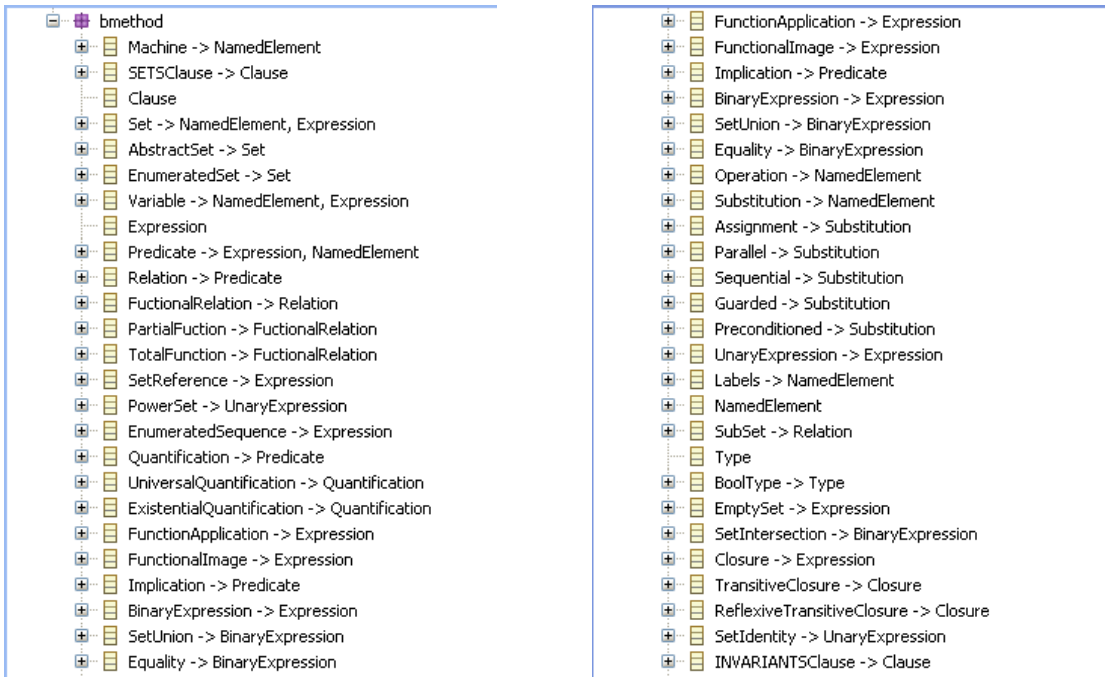
Figure A.6: Part of parsed evolution model

Assigning B formal semantics

Introducing a MOF-based metamodel of B-method enables us to bring B-method into a model-driven engineering setting and define B abstract machines, refinement and implementation as models of B-method metamodel.

Figure A.7 shows parts of the B-method metamodel which we defined in Eclipse using Eclipse Modeling Framework (EMF). This metamodel is the EMF implementation of the B-method metamodel as outlined in Chapter 3. Similar to the way we defined our UML metamodel subset, this definition of B-method metamodel in EMF enables us to use this metamodel as an input in a model transformation process to assign formal semantics to our data metamodel and evolution metamodel.

More specifically, assigning formal semantics to data model evolution with B-method consists of the specification of an Abstract Machine (defined in B Abstract Machine Notation), able to capture the semantics of data models defined in UML and a collection of B abstract operations (defined in B Generalized Substitution Language), able to capture the semantics of our model evolution operations. In particular, our B semantic domain consists of the following two parts:



(a) (b)

Figure A.7: A fragment of B Metamodel

1. *Generic Abstract Machine.* This Abstract Machine describes corresponding constructs of UML modeling language subset, that are necessary for modeling data models.

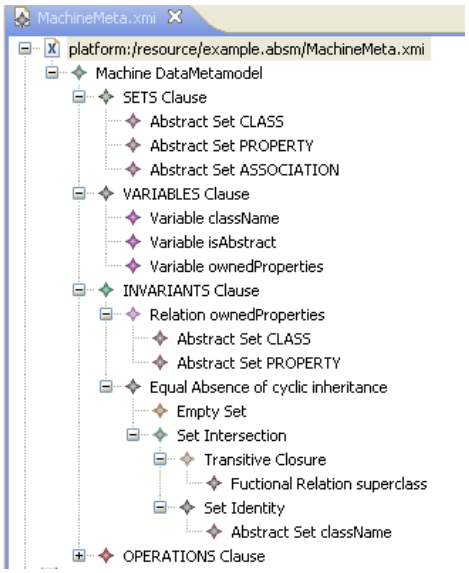


Figure A.8: Part of the abstract machine used to assign semantics to UML data metamodel

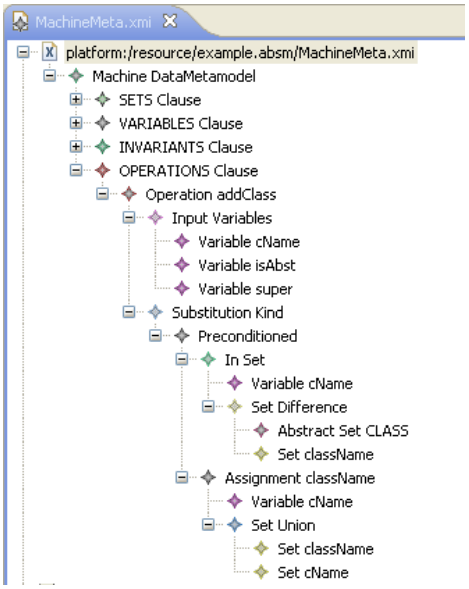
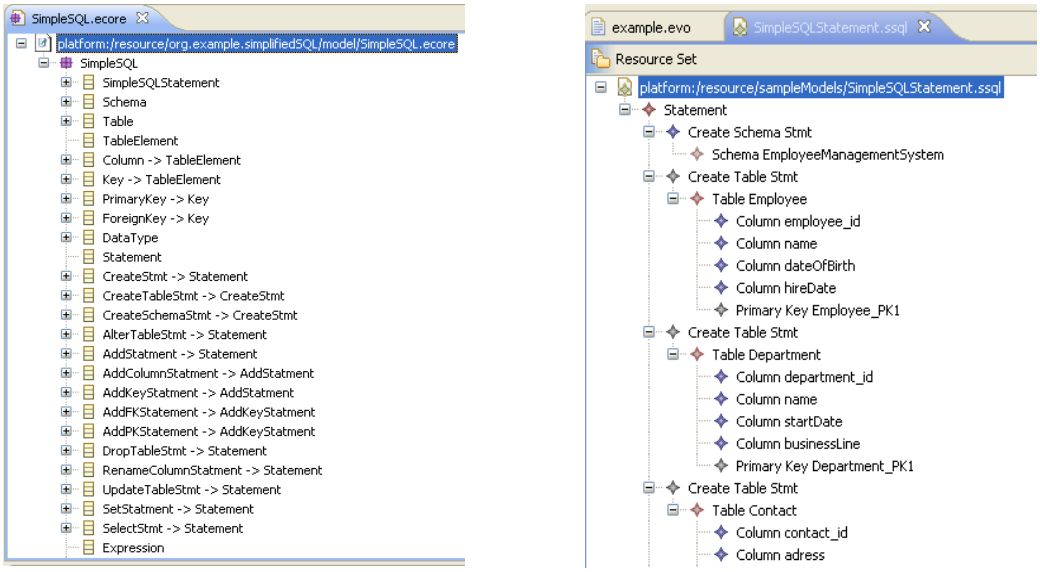


Figure A.9: Part of preconditioned substitution template used to assign semantics to evolution operations

Figure A.8 shows the main clauses of the B abstract machine which is built using concepts of the B-method metamodel, shown in Figure A.7. This generic abstract machine is used to assign formal semantics to the various concepts in the data metamodel. For example, main elements of the data metamodel are represented using abstract set concepts, main characteristics of the data metamodel class concept are represented as machine variables with appropriate type. Machine invariants such as absence of cyclic inheritance are also represented, instantiating relevant elements of the B-method metamodel.

2. *B Abstract Operations.* The meaning of the evolution model defined with the Evolution metamodel is specified by means of B abstract operations expressed in the form of Generalized Substitution Language (GSL). Starting from the given instance of the abstract machine, they evolve both model elements and data instances.

   Figure A.9 shows how we instantiate appropriate elements of the B-method metamodel to assign formal semantics to the evolution operations. For example, operation `addClass` has three input variables. In a preconditioned substitution, these input variables are typed in the precondition clause and assigned to appropriate state variables (`className` in the above example).



(a) SQL Metamodel

(b) SQL Model

Figure A.10: A fragment of SQL Metamodel and a model defined in Eclipse Modeling Framework (EMF)

The above two components of our B-method semantic domain are obtained automatically by means of model transformation from a data model and its corresponding evolution model.

Generating SQL data migration programs

Given the prevalence of relational databases, we assume that the data we want to evolve will have relational implementation. In our current implementation, as a relational database management system (DBMS), we use Apache Derby [4] as implemented in Eclipse data tool platform project [5].

Similar to the way we defined UML and OCL metamodels using EMF framework, we first define SQL metamodel, as a .ecore file. Using our object-to-relational transformation, the data model above may be translated into an initial SQL model conforming to SQL metamodel, as can be seen in Figure A.12

Using model-to-text transformation technique, we can transform SQL model into a corresponding collection of SQL statements. In the prototypical implementation of the approach, we have chosen to do this using the Xpand notation [6]. This notation

[4]http://db.apache.org/derby
[5]http://www.eclipse.org/datatools
[6]http://www.eclipse.org/modeling/m2t/?project=xpand

SQLModel2Text.xpt

```
1 «IMPORT SQLMetamodel»
2
3 «DEFINE main FOR SQLStatement»
4     «FILE name + ".sql"»
5     «EXPAND SQLStmt FOREACH Statment»
6     «ENDFILE»
7 «ENDDEFINE»
8
9 «DEFINE SQLStmt::CreateTable FOR Statment::CreateTableStmt »
10            CREAT TABLE «this.prms.name»
11            primaryKey INTEGER NOT NULL PRIMARY KEY
12            CREAT column «this.prms.name»
13 «ENDDEFINE»
14
15 «DEFINE SQLStmt::DROP FOR Statment::DropTableStmt »
16            DROP TABLE «this.prms.name»
17 «ENDDEFINE»
```

Figure A.11: SQL script Generation (excerpt)

is intended for transforming model descriptions into text files, but performs equally well as a tool to navigate the structure of a model and generate an appropriate text.

EmpInfoSystem_v1.sql

Connection profile

Type: Derby_10.x

```
1  create schema EEInfoSystem;
2
3  create table Employee(
4  employee_id int not null,
5  primary key(employee_id),
6  name char(12),
7  dateOfBirth date,
8  hireDate date);
9
10 create table Department(
11 department_id int not null,
12 primary key(department_id),
13 startDate date,
14 businessLine char(12));
15
16 create table EEMgtSystem.Contact(
17 contact_id int not null,
18 primary key(contact_id),
19 address char(12),
20 phone char(12));
```

(a) SQL Statements

EMPLOYEE

EMPLOYEE_ID...	NAME [CH...	DATEOFBIRTH...	HIREDATE...	WORKSFOR
1000	Employee1	1970-11-12	2010-01-01	10
2000	Employee2	1960-11-12	2010-05-01	20
3000	Employee3	1985-11-12	2010-07-01	30
4000	Employee3	1985-11-14	2010-07-01	40
5000	Employee5	1985-11-15	2010-07-05	50
6000	Employee6	1985-11-16	2010-07-06	60
7000	Employee7	1985-11-17	2010-07-07	70
8000	Employee8	1985-11-18	2010-07-08	80
9000	Employee9	1985-11-19	2010-07-09	90

(b) SQL Table

Figure A.12: A fragment of SQL Statements and a table

Figure A.11 shows part of an Xpand template used to generate SQL code. Typically, the first statement in such a template will be an `IMPORT` statement that imports the required metamodels: once a metamodel has been imported, we can refer to its elements in the programming context in which code fragments are to be produced. The `DEFINE` clause represents a fragment that can be expanded within the context of template execution; the `EXPAND` clause directs the execution to another `DEFINE` template. The output SQL file and (a fragment of) corresponding physical database table are shown in Figure A.12.

Case study [7]

In this case study, we will report on a practical realization of our data model evolution approach. In particular, we will introduce and explain our experience applying the approach which we outlined in the previous chapters to a real life case study. This explanation will focus upon one information system, developed using Booster, a domain-specific data modeling framework created at Oxford. We start by giving a short overview of Booster and our motivation to apply our approach within a Booster context. We then give a brief description of True Colours, the information system on which our evolution approach was demonstrated. We also explain how data model changes were captured, analyzed and used to generate data migration code. We end our case study presentation by outlining some lessons learned.

A brief overview of Booster

Booster is a data modeling language, described in [46, 47] and [162], based on elements of the Abstract Machine Notation (AMN) of the B-method and the Refinement Calculus [114]. The approach aims at automatically generating complete and working object database implementations, sequentially accessed through queries and transactional updates.

A Booster model is a collection of classes, each defining the structure and behavior of a particular kind of object. A `Class` has a list of `attributes`, `associations` and `methods`. `associations` in Booster are normally bidirectional [162]: an explicit association in one direction creates an implicit association in the other. `methods` are defined in terms of preconditions, postconditions, and change-lists.

A compiler for the Booster language maps completed method predicates into statements in a programming notation [58]. Development with the Booster language involves the use of the Booster toolset, which is an extended version of B-toolkit that automates the translation from abstract specification to executable code. This translation takes place as a series of steps [162, 58].

Currently limited data migration support is provided by Booster existing compiler, any significant evolution must be decomposed into a series of smaller changes. There is no notion of 'evolution operations', nor is there any formal 'language of changes'. Although Booster generates complete information systems, there is as yet no facility

[7] Work on this case study was done within the context of [88]. I would like to acknowledge the support provided by James Welch during the analysis of Booster data model changes while working on this case study.

for specifying evolution within the model, and hence generated systems often require a certain amount of manual adaptation following a model revision.

We find it tempting to apply our approach into a Booster information system for two main reasons. First, the verification of our proposed generic approach is mainly based on using AMN of the B-method. Development of Booster grew out of the experience in using the B-toolkit. Second, A translation from Booster object model to B-method is appealing because we can reuse proof theories and tools that were developed for B-method.

Research on Booster has been driven by practical application, through the development and evolution of a range of systems in use within and outside the University of Oxford. One of these systems has been available for inspection at various stages of its development is True Colours system.

Overview of True Colours

True Colours is a patient — or mood-monitoring system designed to support clinical care, health research, and self-management for patients with long-term mental health conditions. The system is used to monitor patients with bipolar disorder (a psychiatric diagnosis) in two counties south of England. In addition, the system is used to computerize many of the reporting and management activities of clinical users and support a range of research projects and clinical studies.

The primary function of the system is to support patient monitoring and self-management of bipolar disorder. Participants are assigned monitoring schedules in which they are prompted for a response at specified times: these could be at particular times of the day, on particular days of the week, at certain times of the month, or at random times within specified intervals. Schedules may be prescribed for days, weeks, months, or indefinite periods. Prompts are typically sent as text messages, inviting or reminding the participant to complete a questionnaire. The questionnaires are carefully designed and validated, and become familiar to the participants, who are given a credit card aide memoire of the questions. Responses may be provided by logging in to a website, by sending an email, or by texting a reply. Each of the questionnaires has a corresponding summary calculation, yielding a single score that serves as a quick, positive or negative, indication of health. The scores are presented in a graph, providing a useful visualization of how health has improved, or not, over time, alongside relevant clinical information: in particular, details of medication.

All of this information is automatically exported to the main patient records system used in the relevant part of the UK National Health Service (in this case,

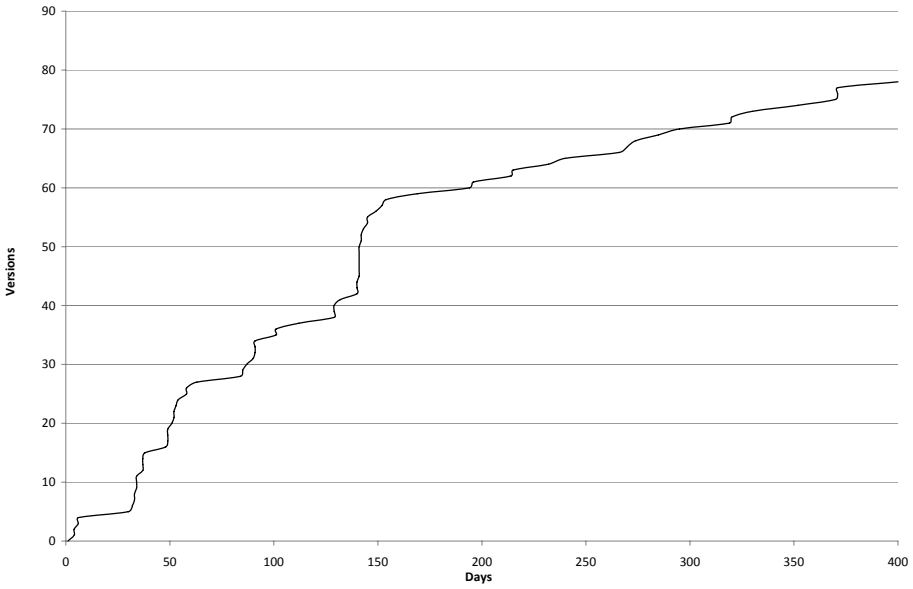


Figure A.13: True colours versions over time

Oxford Health NHS Foundation Trust), and sent also to clinicians in encrypted PDF format. Negotiations are underway regarding wider adoption: for the monitoring of other conditions in Oxfordshire, and for the monitoring of bipolar patients across Scotland

Categorization of data model changes

The first version of the model was created on 4th December 2009, with an initial collection of 7 classes, 46 attributes, and 50 methods, including all the basic functionality needed for user management. Initially, the only users of the system were the clinicians and nurses involved in the monitoring and research programs.

Figure A.13 shows the rate of progress in terms of major versions developed, from December 2009 to June 2011. As the rate varied over the first six months, we will use the version number as the horizontal axis in a subsequent graph. As might be expected, the rate of change in the model remained fairly constant after the first ten or fifteen versions, until the point where the system was about to be exposed to a new user group: that of the patients.

At this point, clinical and nursing staff became more focused upon the potential patient experience, and produced more requests for minor modifications to the design: for the most part, these would entail the addition of attributes to the model, and the modification of existing methods; the classes and enumerations would remain unchanged.

Booster primitive edit	**Total number of changes**
Add Class	42
Delete Class	5
Add Attribute	536
Change Attribute Type	5
Change Attribute Multiplicity	50
Rename Attribute	22
Delete Attribute	59
Add Method	955
Update Method	427
Delete Method	235
Add Enumeration	29
Modify Enumeration	20
Change Class Invariant	42

Table A.1: Recurrence of Booster model edits

Using Booster version files as input to an Eclipse Compare tool, we were able to identify, Booster data model changes from one version to the other. Table A.1 shows the frequency with which each primitive model edit has occurred. We may observe that the highest two numbers refer to the number of times a method was added and the number of times an attribute was also added. This shows a pattern of evolution in which an application being more tightly scoped. Here, designers were not interested in modifying existing elements, rather, they are mainly adding new elements (e.g. attributes, methods or classes). This is different from other evolution patterns were data models are updated to capture some structure or process at a greater level of detail.

Figure A.14 shows the number of changes made, for each kind of model component, against version number. FigureA.15 shows an example of true colours data model changes from one version to the other.

Instantiation and analysis

These version changes can directly be mapped into primitive edits against our evolution metamodel and subsequently given semantics in AMN enabling reasoning and analysis of evolution within our proposed B-method framework. However, since our approach is defined at the metamodel level of design, and our consistency constraints are stated in terms of sets and relations representing metamodel concepts, we need

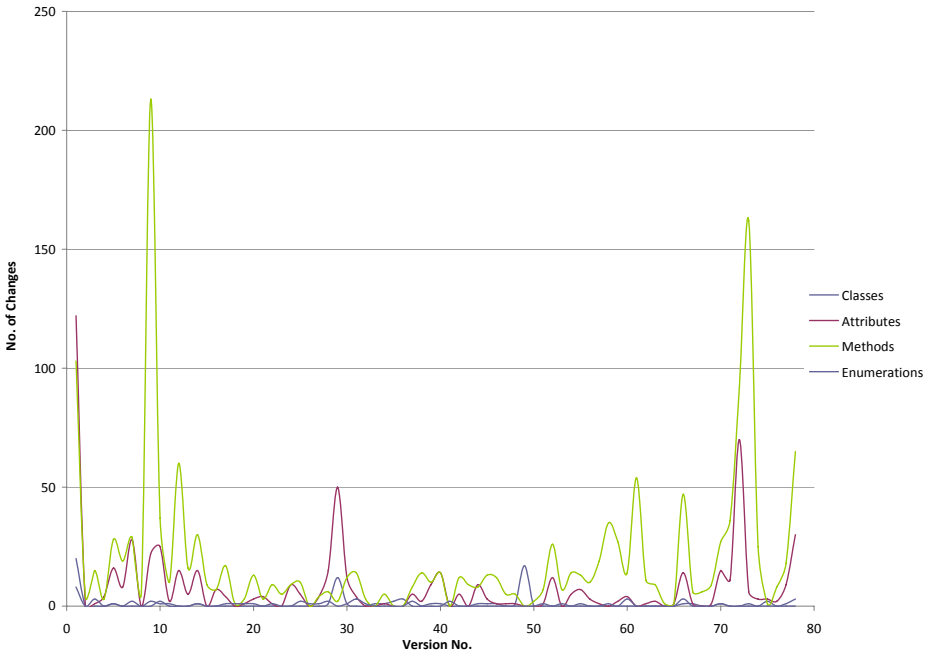


Figure A.14: True colours changes

to introduce an instantiation mechanism by which we can systematically instantiate these metamodel concepts with values from Booster data models.

Booster data model version 9	Booster data model version 10
Enumeration addition	

`CLASS ResponseStatus [20] CONTROL`
`  {NewResponse,`
`  ManuallyDecoded,`
`  AutomaticallyDecoded }`
	`CLASS ResponseStatus [20] CONTROL`
	`  {NewResponse,`
	`  ManuallyDecoded,`
	`  AutomaticallyDecoded,`
	`  FromWeb, FromEmail,`
	`  AutoFromText,`
	`  ManualFromText }`

Attribute renaming	
`objectHistoryForResponse :` `  [Response . responseHistory];`	`objectHistoryForScheduleEvent :` `  [ScheduleEvent . scheduleEventHistory];`
Attribute addition	

`CLASS Response[1000]`
`  ATTRIBUTES`
`  responseTemp (ID) : NAT`
`  END;`
	`CLASS Response`
	`  responseDate (ID) : DATE;`
	`  responseTime (ID) : TIMEOFDAY;`
	`  responseStatus (ID) : ResponseStatus;`
	`  ...`

Attribute deletion	**Attribute addition**
`scheduleResponses :` `  SET ( Response . responseForSchedule )`	`scheduleScheduleEvents :` `  SET ( ScheduleEvent . scheduleEventForSchedule )`
	Class addition

	`CLASS ScheduleEvent[10000]`
	`ATTRIBUTES`
	`  scheduleEventDate (ID) : DATE;`
	`  scheduleEventTime (ID) : TIMEOFDAY;`
	`  ...`

Figure A.15: An example of version-level changes

In other words, to be able to verify the conformance of a Booster data model and its corresponding data instances with respect to the formal semantics, we need to

allow the variables of our data metamodel to have values and encode data model-specific constraints so that we are able to check the conformance of the model and the data with respect to these constraints. The instantiation mechanism which we propose next helps us perform these tasks.

The main idea of our instantiation mechanism is based on the observation that each element in a data model is an instance (object) of a class in the metamodel (for example: all the associations in a UML data model are instances of the Association class in the UML metamodel). Hence, in our instantiation mechanism, we follow the following four steps:

Step 1: each instantiable concept in Booster data metamodel (corresponding to UML metamodel subset) is transformed into a B abstract machine. For example, we have created a separate abstract machine for each Booster `Class`, Booster `Property` and Booster `Association` concept. The properties of each concept are transformed to variables in the corresponding abstract machine. We have then used an abstract machine operation to *instantiate* the variables of the machine with corresponding values from Booster data model.

This way, if we have a Booster class called *Questionnair*, we would create an abstract machine with the same name and populate its metamodel variables such as `className`, `isAbstract` , `ownedProperties`, etc from the *Questionnair* class. Similaly, if we have a property called *QuestionnairePrompt*, we would create an abstract machine and, in this case, populate our property variables such as `propertyName`, `propertyType`, `owningClass` from the values of this property.

Step 2: data representing each metamodel concept (e.g Class, Property, etc) which was collected in individual abstract machines in the previous step is merged, using set union, into variables of one abstract machine representing an aggregated data metamodel. As a result, in this data metamodel machine, we have variables whose values are sets of data of all objects of Booster data model elements.

Step 3: with this instantiation mechanism, we can now express Booster data model-specific constraints as invariants of the data metamodel machine. As a result Proof obligations generated in the support tool can then inspect all objects to verify the consistency of data instances.

Step 4: all individual machines of the data model see `SEES` a context machine which defines all the sets of the data model, such as `CLASS`, `PROPERTY`, `ASSOCIATION` etc. with possible values.

Using the above instantiation mechanism we were able to instantiate our abstract data model and evolution machines with Booster data model values. We have performed this instantiation to analyze three arbitrary subsequent versions (from version 9 to version 12). Once version 9 instantiates the abstract machine corresponding to our metamodel, we would use our evolution operations to evolve the model to the subsequent version (guided by the changes identified in the change files).

Although none of these three versions was a refactoring of its predecessor, the evolution from one version to the other was applicable as the preconditions of operations involved were all satisfiable.

Appendix B

B-method notation ASCII symbols

The table below provides a mapping of main B-method ASCII symbols used in our B specifications.

Symbol	Operator	Symbol	Operator
:	$\in$	/:	$\notin$
<:	$\subseteq$	=>	$\Rightarrow$
;	$\fatsemi$	\/	$\cup$
/\	$\cap$	&	$\wedge$
or	$\vee$	!	$\forall$
#	$\exists$	-->	$\rightarrow$
+->	$\pfun$	>-->	$\rightarrowtail$
<-->	$\leftrightarrow$	|->	$\mapsto$
|>	$\rhd$	<|	$\lhd$
||>	$\rhd\!\!\!\rhd$	>+->	
+->>	$\pfun\!\!\!\rightarrow$	>+->	$\rightarrowtail\!\!\!\mapsto$
-->>	$\rightarrow\!\!\!\rightarrow$	>-->	$\rightarrowtail$
==	$=\!=$	||	$\parallel$
/=	$\neq$	<=	$\leq$
{}	$\varnothing$	~	$\sim$
*	$\otimes$	<=>	$\Leftrightarrow$
NAT	$\mathbb{N}$		

Table B.1: Mapping B ASCII charcters to math operators

Appendix C

B Specifications

Semantics of UML data metamodel

```
MACHINE
    DataMetamodel

SETS
    CLASS ; OBJECTID ; PROPERTY ; ASSOCIATION ;
    VALUE ; TYPE ; PrimitiveTYPE

CONSTANTS
    typeOf , primitiveType , classType

PROPERTIES
    primitiveType : PrimitiveTYPE --> TYPE       &
    classType     : CLASS --> TYPE               &
    ran (primitiveType) /\ ran (classType) = {}  &
    typeOf        : VALUE >-> TYPE

VARIABLES
    /* Class variables */
      className , isAbstract , superclass , ownedProperties ,

    /* Property variables */
      propertyName , propertyType , owningClass ,
      propertyLowerMultip , propertyUpperMultip ,
      isComposite , opposite ,

    /* Association variables */
      association , associationName , memberEnds ,

    /* Association variables */
      extent , value , link

```

```
INVARIANT
    /* CLASS formalization */
      className      <: CLASS                    &
      isAbstract      : CLASS +-> BOOL           &
      superclass      : CLASS +-> CLASS          &
      ownedProperties : CLASS <-> PROPERTY       &

    /* PROPERTY formalization*/
      propertyName       <: PROPERTY             &
      isComposite         : PROPERTY +-> BOOL    &
      propertyLowerMultip : PROPERTY +-> NAT     &
      propertyUpperMultip : PROPERTY +-> NAT1    &
      owningClass         : PROPERTY +-> CLASS   &
      propertyType        : PROPERTY +-> TYPE    &

    /* ASSOCIATION formalization*/
      associationName <: ASSOCIATION                                &
      association      : ASSOCIATION +-> (CLASS +-> CLASS )        &
      memberEnds       : ASSOCIATION +-> (PROPERTY +-> PROPERTY ) &
      opposite         : PROPERTY    +->  PROPERTY                &

    /* Instance-level formalization*/
      extent : CLASS       +-> POW (OBJECTID)                     &
      value  : PROPERTY    <-> (OBJECTID +-> VALUE)               &
      link   : ASSOCIATION <-> (OBJECTID +-> POW ( OBJECTID))     &


    /*************************Name uniquness*********************/
    !(c1,c2).(c1:className & c2:className &
        c1 /= c2 & (c1|->c2) : closure (superclass) =>
        ownedProperties[{c1}] /\ ownedProperties[{c2}] = {}) &

    /*********************Association bidirectionality***********/

    /* A member end property of an association must be owned
    by a class participating in the same association*/

    !(aa,p1,p2).(aa:dom(association) &
        p1:dom(memberEnds(aa)) &
        p2=memberEnds(aa)(p1) =>
        owningClass(p1) : dom(association(aa)) &
        owningClass(p2) : ran(association(aa))) &

    /* If a property is an opposite to another property in an association,
    the other property should be the opposite of this property*/

    !(aa1,p1,p2).(aa1:dom(memberEnds) &
```

```
79          p1:dom(memberEnds(aa1))&
80          p2:ran(memberEnds(aa1))&
81          (p1|->p2) : opposite =>
82          #(aa2).(aa2: dom(memberEnds)&
83              memberEnds(aa2) = {(p2|->p1)})) &
84
85      /****************Absence of circular inheritance******************/
86      /*In an inheritance hierarchy, no class can generalize itself*/
87      closure1(superclass) /\ id(className) = {} &
88
89      /***************Instance conformance*******************************/
90      /* An object can only take values for the attributes of its class
91      (including its superclass) */
92
93      !(cc,oo,pp).(cc:className &
94          oo:extent(cc)         &
95          pp:PROPERTY           &
96          oo : dom (value (pp)) &
97          owningClass(pp) = cc
98          =>
99          pp : ownedProperties [(closure1 (superclass) \/ id(className))[{cc}]]) &
100
101     /*****************Value conformance*****************************/
102     /*Each attribute is instantiated into a value and assigned to an object.
103     This value needs to be consistent with the type of the attribute*/
104
105     !(cc,aa,oo).(cc:dom(extent) &
106         aa:ownedProperties[{cc}] &
107         oo:extent(cc) =>
108         typeOf(value(aa)(oo)) = propertyType(aa) ) &
109
110     /******************Link conformance*****************************/
111     /* The number of objects participating in an association link
112     must be permitted by the multiplicity of corresponding member ends */
113
114     !(aa,pp).(aa:dom(memberEnds) & pp : dom(memberEnds(aa)) =>
115         (!(cc,oo).(cc:dom(association(aa)) & oo:extent(cc) =>
116                 (card(link(aa)(oo))) >= propertyLowerMultip(pp) &
117                 (card(link(aa)(oo))) <= propertyUpperMultip(pp) )))
118
119
120      /*******************other consistency conditions*********************/
121      /*invariant properties on owningClass */
122
123       ran(owningClass) <: className &
124       dom(owningClass) <: propertyName &
125
```

```
INITIALISATION
    ...

OPERATIONS
    ...
END
```

Semantics of model evolution primitives

```
1
2  OPERATIONS
3
4   /*************************class addition**********************/
5      addClass(cName,isAbst,super) =
6      PRE
7          cName  : CLASS - className &
8          cName /: dom(extent)       &
9          cName /: dom(superclass)   &
10         cName /: ran(superclass)   &
11         cName /: dom(ownedProperties) &
12         cName /: dom(isAbstract)   &
13         isAbst : BOOL              &
14         super  : className
15
16     THEN
17         className  := className   \/ {cName}             ||
18         isAbstract := isAbstract  \/ {cName |-> isAbst}  ||
19         superclass := superclass  \/ {cName |-> super }  ||
20         extent     := extent      \/ {cName |->{}}
21
22     END;
23
24     /*************************class deletion**************************/
25  deleteClass (cName) =
26     PRE
27         cName  : className &
28         cName /: ran(superclass) &
29         ownedProperties[{cName}] /\ ran(opposite) = {}
30
31     THEN
32         ANY
33             cObjects, cProperties,cAssociations
34         WHERE
35             cObjects = extent(cName) &
36             cProperties = ownedProperties[{cName}] &
37             cAssociations = {asso|asso : ASSOCIATION &
38                 (dom(association(asso))\/
39                     ran(association(asso))) = {cName}}
40         THEN
41             className       := className - {cName}                ||
42             isAbstract      := {cName} <<| isAbstract             ||
43             superclass      := {cName} <<| superclass |>> {cName} ||
44             ownedProperties := {cName} <<| ownedProperties     ||
45             propertyName    := propertyName - cProperties      ||
```

```
46              propertyType    := cProperties   <<| propertyType  ||
47              owningClass     := owningClass   |>> {cName}       ||
48              association     := cAssociations <<| association   ||
49              memberEnds      := cAssociations <<| memberEnds    ||
50              extent          := {cName}       <<| extent        ||
51              value           := cP roperties  <<| value         ||
52              link            := cAssociations <<| link
53          END
54      END;
55
56  /**********************attribute addition*******************/
57   addAttribute(cName,attrName,type,exp) =
58      PRE
59          cName    : CLASS                     &
60          attrName : PROPERTY - propertyName   &
61          cName /= owningClass(attrName)       &
62          attrName /:ownedProperties[{cName}] &
63          attrName /: dom(propertyType)        &
64          attrName /: dom(value)               &
65          type      : ran(primitiveType)       &
66          exp       : VALUE                    &
67          typeOf(exp) = type                   &
68          cName    /: ran(superclass)
69      THEN
70          ANY objectId
71          WHERE objectId = extent(cName)
72          THEN
73              propertyName          := propertyName    \/ {attrName}         ||
74              ownedProperties       := ownedProperties \/ {cName|->attrName} ||
75              owningClass(attrName) := cName                                 ||
76              propertyType          := propertyType    \/ {attrName|-> type} ||
77              value                 := value           \/
78                                           {attrName|->(objectId * {exp})}
79          END
80      END;
81
82  /************************attribute deletion******************/
83  deleteAttribute(cName,attrName) =
84      PRE
85          cName : CLASS &
86          attrName : PROPERTY
87      THEN
88          propertyName    := propertyName - {attrName}                ||
89          ownedProperties := ownedProperties - { cName |-> attrName}  ||
90          owningClass     := {attrName} <<| owningClass               ||
91          propertyType    := {attrName} <<| propertyType              ||
92          value           := {attrName} <<| value
```

```
93      END;
94
95    /**********************association addition******************/
96    addAssociation(assoName,srcClass,srcProp,tgtClass,tgtProp,isComp,exp) =
97      PRE
98          assoName: ASSOCIATION - associationName &
99          srcClass : CLASS & srcClass /: dom(association(assoName)) &
100         tgtClass : CLASS & tgtClass /: ran(association(assoName)) &
101         srcProp  : PROPERTY - propertyName &
102         srcProp /:ownedProperties[{srcClass}]&
103         srcClass /= owningClass(srcProp) &
104         srcProp /: dom(isComposite) &
105         srcProp /: dom(propertyType) &
106         srcProp /: dom(opposite) &
107         tgtProp  : PROPERTY - propertyName &
108         tgtProp /:ownedProperties[{tgtClass}]&
109         tgtClass /= owningClass(tgtProp) &
110         tgtProp /: dom(isComposite) &
111         tgtProp /: dom(propertyType) &
112         tgtProp /: dom(opposite) &
113         isComp   : BOOL &
114         exp      : OBJECTID
115     THEN
116         ANY srcOID
117         WHERE srcOID = extent(owningClass(srcProp))
118         THEN
119             associationName := associationName \/ {assoName}            ||
120             propertyName    := propertyName    \/  {srcProp,tgtProp}    ||
121             association     := association     \/
122                                 {assoName|->{srcClass|->tgtClass}}      ||
123             memberEnds      := memberEnds      \/
124                                 {assoName|->{srcProp|-> tgtProp}}       ||
125             ownedProperties := ownedProperties \/
126                                {srcClass|->srcProp, tgtClass|->tgtProp} ||
127             owningClass     := owningClass     \/
128                                 {(srcProp|-> srcClass),
129                                  (tgtProp|-> tgtClass)}                 ||
130             propertyType    := propertyType    \/
131                                   {(srcProp|-> classType(tgtClass)),
132                                    (tgtProp|-> classType(srcClass))}    ||
133             isComposite     := isComposite     \/ {srcProp|-> isComp}   ||
134             link            := link            \/
135                                    { assoName |-> (srcOID * {{exp}} )}
136         END
137     END;
138
139   /**************************association deletion*************************************/
```

```
140  deleteAssociation(assoName) =
141     PRE
142         assoName : ASSOCIATION
143     THEN
144         ANY
145             srcClass,tgtClass,
146             srcProp,tgtProp
147         WHERE
148             srcClass : dom(association(assoName))&
149             tgtClass : ran(association(assoName))&
150             srcProp  : dom(memberEnds(assoName)) &
151             tgtProp  : ran(memberEnds(assoName))
152         THEN
153             associationName := associationName - {assoName}      ||
154             propertyName    := propertyName - {srcProp,tgtProp}  ||
155             association     := {assoName} <<| association        ||
156             memberEnds      := {assoName} <<| memberEnds         ||
157             propertyType    := {srcProp,tgtProp}<<| propertyType ||
158             ownedProperties := ownedProperties  -
159                                  {srcClass|->srcProp, tgtClass|->tgtProp} ||
160             owningClass     := owningClass      -
161                                  {srcProp|->srcClass}-{tgtProp|->tgtClass} ||
162             link            := {assoName} <<| link
163         END
164     END
165  END
166
```

Appendix :
Object to Relational Refinement

```
REFINEMENT
   Object_to_Relational
REFINES
   DataMetamodel3

VARIABLES
    classKey ,  ownedAttributes , ownedAssoEnds ,
    inheritedAttributes , inheritedAssoEnds ,
    propertyClass , associationTable

INVARIANT
    classKey                 : CLASS    >+> NAT              &
    propertyClass            : PROPERTY <-> CLASS            &
    ownedAttributes         <: ownedProperties               &
    ownedAssoEnds           <: ownedProperties               &
    inheritedAttributes      : CLASS    <-> PROPERTY         &
    inheritedAssoEnds        : CLASS    <-> PROPERTY         &
    associationTable         : ASSOCIATION +-> ( PROPERTY +-> PROPERTY ) &

/*************linking invariants ******************/

    dom(classKey) <: className &
    dom(classKey) = isAbstract ~ [{FALSE}] &

    propertyClass =
     (ownedAttributes \/ inheritedAttributes \/
      ownedAssoEnds   \/ inheritedAssoEnds ) ~        &

    owningClass = (ownedAttributes \/ ownedAssoEnds)~ &

    dom ( association ) = dom ( associationTable ) &

            ! (aa,ae1,ae2).(aa : dom (associationTable) &
                ae1 : dom (associationTable(aa)) &
                ae2 : ran (associationTable(aa)) =>
                propertyClass [{ae1}] = dom (association(aa))    &
                propertyClass [{ae2}] = ran (association(aa)))


    ownedAttributes      \/ ownedAssoEnds       = ownedProperties    &
    ran ( ownedAttributes ) /\ ran ( ownedAssoEnds ) = {}            &

    propertyType [ ran ( ownedAttributes ) ] = ran ( primitiveType ) &
```

```
44      propertyType [ ran ( ownedAssoEnds ) ] = ran ( classType )       &
45
46     ownedAttributes =
47         ownedProperties |> propertyType ~ [ ran ( primitiveType ) ]   &
48     ownedAssoEnds   =
49           ownedProperties |> propertyType ~ [ ran ( classType ) ]     &
50
51    dom ( inheritedAttributes ) <: className &
52       ! cc . ( cc : className &
53           cc /: dom ( superclass ) /* toplevel class */
54           =>
55           inheritedAttributes [ { cc } ] = {} ) &
56       ! cc . ( cc : className &
57            cc : dom ( superclass ) /* inherited class */
58            =>
59            inheritedAttributes [{cc}] = ownedAttributes
60                                         [closure1(superclass)[{cc}]]) &
61
62    dom ( inheritedAssoEnds ) <: className &
63      ! cc . ( cc : className &
64          cc /: dom ( superclass )
65          =>
66          inheritedAssoEnds [ { cc } ] = {} ) &
67      ! cc . ( cc : className &
68          cc : dom ( superclass )
69            =>
70            inheritedAssoEnds[{cc}] = ownedAssoEnds
71                                       [closure1(superclass)[{cc}]])
72   /************************************************************/
73  INITIALISATION
74
75      classKey                := {} ||
76      inheritedAttributes     := {} ||
77      inheritedAssoEnds       := {} ||
78      ownedAttributes         := {} ||
79      ownedAssoEnds           := {} ||
80      associationTable        := {} ||
81      propertyClass           := {}
82      ...
83
84  OPERATIONS
85
86  /**********************Refinement of class addition******************/
87  addClass ( cName , isAbst , super ) =
88      IF isAbst  = FALSE
89      THEN
90          ANY   cKey
```

```
91          WHERE cKey : NAT - ran ( classKey )
92          THEN
93                classKey := classKey \/ { cName |-> cKey } END ||
94      inheritedAttributes := inheritedAttributes \/
95           ({cName}*(ownedAttributes
96                    [closure1 (superclass \/{cName|->super})[{cName}]])) ||
97      inheritedAssoEnds := inheritedAssoEnds \/
98            ({cName }*(ownedAssoEnds
99                     [closure1(superclass \/ {cName|->super})[{cName}]]))  ||
100     propertyClass := propertyClass \/
101                        ((inheritedAttributes[{super}])*{cName}) \/
102                        (ownedAttributes [{super}]*{cName} )               ||
103     END ;
104
105 /**********************Refinement of class deletion**********************/
106 deleteClass ( cName ) =
107   BEGIN
108     ownedAttributes     := { cName }     <<| ownedAttributes     ||
109     ownedAssoEnds       := { cName }     <<| ownedAssoEnds       ||
110     propertyClass       := propertyClass |>> { cName }           ||
111     inheritedAttributes := { cName }   <<| inheritedAttributes   ||
112     inheritedAssoEnds   := { cName }   <<| inheritedAssoEnds     ||
113     classKey            := { cName }   <<| classKey             ||
114     ANY
115         classAssos , subclassAssos
116     WHERE
117         classAssos = { asso | asso : ASSOCIATION &
118                     ( ( dom ( associationTable ( asso ) ) \/
119                         ran ( associationTable ( asso ) ) ) /\
120                         ownedAssoEnds [ { cName } ] ) /= {} } &
121         subclassAssos = { asso | asso : ASSOCIATION &
122                         ( ( dom ( associationTable ( asso ) ) \/
123                             ran ( associationTable ( asso ) ) ) /\
124                             inheritedAssoEnds [ { cName } ] ) /= {} }
125     THEN
126         associationTable := ( classAssos \/ subclassAssos ) <<|
127                                 associationTable
128     END
129   END ;
130 /********************Refinement of attribute addition**********************/
131 addAttribute ( cName , attrName , type , exp ) =
132 BEGIN
133   ownedAttributes := ownedAttributes \/ { cName |-> attrName } ||
134
135   inheritedAttributes := inheritedAttributes \/
136                            closure1 (superclass~)[{cName}]*{attrName } ||
137   propertyClass := propertyClass  \/
```

```
                        {attrName}*closure1(superclass~)[{cName}] \/
                         {attrName |-> cName }
END ;
/******************Refinement of attribute deletion***********************/
deleteAttribute ( cName , attrName ) =
BEGIN
   ownedAttributes     := ownedAttributes - { cName |-> attrName }    ||
   ownedAssoEnds       := ownedAssoEnds   - { cName |-> attrName }    ||
   inheritedAttributes := inheritedAttributes -
                             closure1(superclass~)[{cName}]*
                                {attrName} ||
   propertyClass       := propertyClass  -
                           {attrName} * closure1(superclass~)[{cName}]
                            - { attrName |-> cName }
END ;
/**********************Refinement of association addition**********************/
addAssociation ( assoName , srcClass , srcProp ,
                 tgtClass , tgtProp , isComp , exp ) =
BEGIN
   associationTable := associationTable \/
                { assoName |-> { srcProp |-> tgtProp } } ||
    ownedAssoEnds := ownedAssoEnds \/
               { ( srcClass |-> srcProp ) ,
               ( tgtClass |-> tgtProp ) } ||
    inheritedAssoEnds   :=  inheritedAssoEnds \/
              closure1 ( superclass ~ ) [ { srcClass } ] * { srcProp } \/
              closure1 ( superclass ~ ) [ { tgtClass } ] * { tgtProp } ||
    propertyClass       := propertyClass  \/
              { srcProp } * closure1 ( superclass ~ ) [ { srcClass } ] \/
              { srcProp |-> srcClass }                      \/
              { tgtProp } * closure1 ( superclass ~ ) [ { tgtClass } ] \/
              { tgtProp |-> tgtClass }

END ;
/**********************Refinement of association deletion********************/

deleteAssociation ( assoName ) =
  ANY
      srcProp , tgtProp
  WHERE
       srcProp = { srcP | srcP : PROPERTY &
                   srcP : dom ( associationTable ( assoName ) ) } &
       tgtProp = { tgtP | tgtP : PROPERTY &
                   tgtP : ran ( associationTable ( assoName ) ) }
  THEN
       associationTable    := { assoName } <<| associationTable ||
       ownedAssoEnds       := ownedAssoEnds     |>> ( srcProp \/ tgtProp ) ||
```

```
185         inheritedAssoEnds   := inheritedAssoEnds |>> ( srcProp \/ tgtProp ) ||
186         propertyClass       := ( srcProp \/ tgtProp ) <<| propertyClass
187   END
188 END
189
```

Appendix :
SQL Abstract Machine

```
MACHINE
 SQL

SETS
 TABLE ; COLUMN ; TUPLE ; Sql_TYPE ; Sql_VALUE

CONSTANTS
    idColType ,
    Sql_Default

PROPERTIES
    idColType   : NAT >-> Sql_TYPE &
    Sql_Default : Sql_VALUE

VARIABLES
 tableName , tableColumns ,
 columnName , columnType , parentTable , canBeNull , isID , isUnique ,
 primaryKey , foreignKey ,
 tuple

INVARIANT
 tableName       <: TABLE                   &
 tableColumns     : TABLE  <-> COLUMN       &
 columnName      <: COLUMN                  &
 columnType       : COLUMN +-> Sql_TYPE     &
 parentTable      : COLUMN <-> TABLE        &
 canBeNull        : COLUMN +-> BOOL         &
 isID             : COLUMN +-> BOOL         &
 isUnique         : COLUMN +-> BOOL         &
 primaryKey       : TABLE  <-> COLUMN             &
 foreignKey       : TABLE  <-> (COLUMN --> TABLE) &
 tuple            : TABLE  <-> (COLUMN +-> Sql_VALUE)


INITIALISATION
 tableName        := {} ||
 tableColumns     := {} ||
 columnName       := {} ||
 columnType       := {} ||
 parentTable      := {} ||
 canBeNull        := {} ||
 isID             := {} ||
 isUnique         := {} ||
```

```
 primaryKey       := {} ||
 foreignKey       := {} ||
 tuple            := {}

OPERATIONS

/************************Basic table operations**********************/
addTable (nn) =
    PRE
      nn : TABLE - tableName
    THEN
      tableName := tableName \/ {nn}
    END ;
/*-------------------------*/
alterTable (nn , col) =
    PRE
      nn : tableName & col : COLUMN
  THEN
         IF  (nn |-> col)  /: tableColumns THEN
              tableColumns  :=  tableColumns \/ {nn |-> col}
         ELSE
             tableColumns   := tableColumns - {nn |-> col}
         END
             tableColumns   := tableColumns \/ {nn |-> col}
  END ;
/*-------------------------*/
updateTable (nn) =
   PRE
       nn : dom (tuple)
   THEN
       skip
   END ;
/*-------------------------*/
removeTable (nn) =
    PRE
      nn : tableName
    THEN
      tableName := tableName - {nn }
    END ;

/************************Basic column operations**********************/
add_id_Column (nn , tn , type) =
  PRE
     nn     : COLUMN - columnName &
     tn     : tableName &
     type   : NAT
  THEN
```

```
91      columnName  := columnName  \/ { nn }                    ||
92      columnType  := columnType  \/ { nn |-> idColType (type)} ||
93      parentTable := parentTable \/ { nn |-> tn }              ||
94      canBeNull   := canBeNull   \/ { nn |-> FALSE }          ||
95      isID        := isID        \/ { nn |-> TRUE }           ||
96      isUnique    := isUnique    \/ { nn |-> TRUE }
97    END ;
98  /*----------------------*/
99  addColumn (nn , tn , type) =
100   PRE
101     nn     : COLUMN - columnName &
102     tn     : tableName           &
103     type   : Sql_TYPE
104   THEN
105     columnName := columnName   \/ { nn }          ||
106     columnType := columnType   \/ { nn |-> type}  ||
107     parentTable := parentTable \/ { nn |-> tn }   ||
108     canBeNull  := canBeNull    \/ { nn |-> TRUE}  ||
109     isID       := isID         \/ { nn |-> FALSE} ||
110     isUnique   := isUnique     \/ { nn |-> FALSE}
111   END ;
112 /*----------------------*/
113 removeColumn (colName , tName) =
114   PRE
115     colName : columnName &
116     tName   : tableName &
117     colName : tableColumns [ { tName } ]
118    THEN
119     columnName  := columnName - {colName}      ||
120     columnType  := { colName } <<| columnType  ||
121     parentTable := { colName } <<| parentTable ||
122     canBeNull   := { colName } <<| canBeNull   ||
123     isID        := { colName } <<| isID        ||
124     isUnique    := { colName } <<| isUnique
125  END ;
126
127  /*--------------------------------*/
128 allTablecolumns <-- getColumns ( tName ) =
129  PRE
130     tName : tableName
131  THEN
132     allTableColumns := [tableColumns [{ tName }]]
133  END ;
134 /************************Basic key operations***********************/
135 add_pk_Key ( cn , tName ) =
136    PRE
137       cn    : columnName &
```

```
138        tName : tableName  &
139        tName : parentTable [ { cn } ]
140     THEN
141        primaryKey  := primaryKey \/ { tName |-> cn }
142     END ;
143  /*-----------------------*/
144  remove_pk_Key ( tKey ) =
145    PRE
146        tKey : dom (primaryKey)
147    THEN
148        primaryKey  := { tKey } <<| primaryKey
149    END;
150
151  /*-------------------------*/
152  add_fk_Key ( cn , t1 , t2 ) =
153     PRE
154        cn : columnName    &
155        t1 : tableName     &
156        t2 : tableName     &
157        t1 /= t2 & t1 : parentTable [ { cn } ]
158     THEN
159        foreignKey  := foreignKey \/ { t1 |-> { cn |-> t2 } }
160    END ;
161
162  /*-----------------------------------*/
163  fkOwngTables <--getForgnKeyTables (tName) =
164      PRE
165          tName : tableName
166      THEN
167          ANY tables, result WHERE
168              tables <: TABLE &
169              tables = { ta | ta : TABLE & ta : dom(foreignKey) &
170                  #co.(co : COLUMN & co |-> tName : foreignKey(ta))} &
171              result : iseq(TABLE) &
172              ran(result) = tables
173          THEN
174              fk_owng_Tables := result
175          END
176      END;
177  /*-------------------------*/
178  remove_fk_Key ( tKey ) =
179     PRE
180        tKey : dom ( foreignKey )
181     THEN
182        foreignKey  := { tKey } <<| foreignKey
183     END ;
184  /************************Basic value operations**********************/
```

```
185 setValue ( tName , colName , initialValue ) =
186  PRE
187      tName        : dom (tuple) &
188      colName      : tableColumns [{ tName }] &
189      initialValue : Sql_VALUE
190  THEN
191     tuple ( tName ) := { colName |-> initialValue }
192  END ;
193 
194 /*--------------------------*/
195 removeValue ( tName ) =
196  PRE
197      tName   : dom (tuple)
198  THEN
199      tuple := {tName} <<| tuple
200  END
201 
202 END
203 
```

Appendix :
Data Migration Implementation

```
IMPLEMENTATION
    DataMigration

REFINES
    Object_to_Relational

IMPORTS
    SQL

CONSTANTS
    sqlType , propertyToColumn , tableHierarchy , mapObjectID , mapSqlValue ,
    classToTable , mapClassKey ,
    sqlTypeOf , assoToTable , columnHierarchy

PROPERTIES
    mapSqlType       : TYPE        >-> Sql_TYPE    &
    mapSqlValue      : VALUE       >-> Sql_VALUE   &
    mapObjectID      : OBJECTID    --> Sql_VALUE   &
    propertyToColumn : PROPERTY    --> COLUMN      &
    classToTable     : CLASS       --> TABLE       &
    classToTable~    : TABLE       +-> CLASS       &
    assoToTable      : ASSOCIATION --> TABLE       &
    tableHierarchy   : CLASS +-> seq ( TABLE )     &
    columnHierarchy  : CLASS +-> seq ( COLUMN )

INVARIANT

/*mapping data model subclasses into corresponding SQL Tables */
    !cl.(cl : className
        =>
        ran(tableHierarchy(cl)) = classToTable[closure1(superclass~)[{cl}]]) &

/*mapping data owned attributes into corresponding SQL columns */
    ownedAttributes = { cc, att |
        cc : CLASS &
        att : PROPERTY &
        cc |-> att : ownedAttributes &
        classToTable(cc) |-> propertyToColumn(att) : tableColumns } &

/*mapping data inherited attributes into corresponding SQL columns */
        inheritedAttributes = { cc, att | cc : CLASS &
        att : PROPERTY &
        classToTable(cc) : ran(tableHierarchy(cc)) &
```

```
44          propertyToColumn(att) : tableColumns[{classToTable(cc)}] } &
45
46      propertyClass = { pp, cc | pp : PROPERTY &
47          cc : CLASS &
48          pp |-> cc : propertyClass &
49          propertyToColumn(pp) |-> classToTable(cc) : parentTable } &
50
51
52  /*Each non-abstract class in the data model should be
53      represented as a table in the SQL model*/
54      tableName <: classToTable [ className ] &
55      ! cl . (cl:className & cl : dom (classKey) => cl : dom(classToTable ) ) &
56
57  /*column names, execluding id column names, are a subset of property names
58      in the abstract data model*/
59      columnName <: propertyToColumn [ propertyName ] &
60      ! colName . ( colName : columnName &
61          isID ( colName ) = FALSE
62          =>
63          colName : propertyToColumn [ propertyName ] ) &
64
65
66  /* value of the id column in SQL must match the value of the class key
67      in the object model*/
68      ! (cc , col) . (cc : dom ( classKey) &
69          col : tableColumns [ classToTable [{cc}]] &
70          isID (col) = TRUE
71          =>
72          sqlTypeOf ( idColType (classKey ( cc))) =
73            tuple (classToTable (cc)) (col)) &
74
75  /* association member ends in the abstract data model become
76      foreign keys in an association table in the SQL model*/
77      ! ( assoName , me1 , me2 ) . ( assoName : dom ( memberEnds ) &
78          me1 : dom (memberEnds (assoName)) &
79          me2 : ran (memberEnds (assoName))
80        =>
81          assoToTable (assoName ) : dom ( foreignKey) &
82          # ff . ( ff : foreignKey [ { assoToTable ( assoName)}] &
83              propertyToColumn ( me1 ) |-> classToTable ( owningClass ( me1)) : ff) &
84          # ff . ( ff : foreignKey [ { assoToTable ( assoName ) } ] &
85              propertyToColumn ( me2) |-> classToTable ( owningClass ( me2)) : ff) &
86          propertyToColumn ( me1 ) |-> classToTable ( owningClass ( me1)) :
87          union ( foreignKey [ { assoToTable ( assoName)}])) &
88
89  /*all table names in an SQL model are mapped either from
90      data model classes, represented by (dom(extent)) or
```

```
91      from data model associations*/
92      dom ( tuple ) =
93      classToTable [ dom ( extent ) ] \/ assoToTable [ dom ( link ) ] &
94
95  /*a value in the abstract data model is mapped to a corresponding
96      tuple value in SQL model*/
97      ! ( pp , oid ) . ( pp : dom ( value) &
98          oid : dom ( value ( pp ) ) =>
99          mapSqlValue ((value (pp) (oid))) =
100         union (tuple[classToTable [propertyClass[{pp}]]])(propertyToColumn(pp )))
101
102 /* a link in the abstract data model is mapped to a tuple in an association table
103     where object ids that make up the link in the data model are mapped
104     to corresponding foreign keys */
105
106 !(nn,o1,o2).(nn : dom(link) &
107     o1 : dom(link(nn)) &
108     o2 : ran(link(nn))
109     =>
110     assoToTable(nn) : dom(tuple) &
111     mapObjectID(o1)) =
112        tuple( assoToTable(nn) )[ foreignKey(assoToTable(nn)) ] &
113     mapObjectID[o2])) =
114        tuple(assoToTable(nn)[foreignKey[assoToTable(nn)]]) )
115
116
117 OPERATIONS
118
119 /*****************Implementation of class addition***********/
120
121 addClass (cName , isAbst , super) =
122 BEGIN
123       VAR
124          tName, colName, colType, counter, curr_column
125       IN
126          tName := classToTable(cName);
127          colName := tName;
128          colType := mapClassKey(cName);
129          inh_Columns := columnHierarchy (super);
130          counter := 1 ;
131          curr_column := inh_Columns (counter);
132
133          addTable (tName);
134          add_id_Column (colName,tName,colType);
135          add_pk_Key (colName,tName);
136
137          WHILE
```

```
138             counter <= size (inh_Columns)
139          DO
140             alterTable (tName, curr_column);
141             addColumn  (colName,tName, colType);
142             counter := counter + 1
143          INVARIANT
144             primaryKey [{tName}] \/
145                propertyToColumn [inh_Columns [1..counter]] <:
146                   tableColumns[{tName}]
147          VARIANT
148             card (inh_Columns) - counter
149          END
150       END
151 END;
152
153 /*****************Implementation of class deletion***********/
154 deleteClass ( cName ) =
155
156    BEGIN
157       VAR
158          tName,fk_owng_Tables, fk_column, count_fkTable
159       IN
160          fk_owng_Tables := getFKTables (tName);
161          tName          := classToTable(cName);
162          fk_column      := foreignKey(curr_fkTable)
163          count_fkTable  := 1;
164
165 /* loop to de-link foreign keys */
166       WHILE
167          count_fkTable < = size (fk_owng_Tables)
168       DO
169          VAR
170             curr_fkTable
171          IN
172             curr_fkTable := fk_owng_Tables (count_fkTable) ;
173             remove_fk_Key (curr_fkTable );
174             alterTable (curr_fkTable, fk_column) ;
175             removeColumn (fk_column , curr_fkTable);
176             count_fkTable := count_fkTable + 1
177          END
178       INVARIANT
179          !(ta1,ta2,col) . (ta1:dom(foreignKey) &
180             (col|->ta2):foreignKey[{ta1}] =>
181               (ta1|->col) : tableColumns
182       VARIANT size (fk_owng_Tables) - counter
183       END
184     END;
```

```
185
186 /* loop to remove table columns */
187
188    VAR
189       allTableColumns, count_column
190    IN
191       allTableColumns := getTableColumns(classToTable(cName));
192       count_column    := 1;
193    WHILE
194       count_column < size(allTableColumns)
195    DO
196       VAR curr_column
197       IN
198          curr_column := allTableColumns (count_column);
199          alterTable (tName);
200          removeColumn (tName, curr_column);
201          count_column := count_column +1
202      END
203    INVARIANT
204       !(ta, col) . (ta : dom(tableColumns) &
205          col : tableColumns[{ta}] =>
206            col |-> ta : parentTable)
207    VARIANT
208       card (allTableColumns) - count_column
209    END
210    END
211
212    remove_pk_Key (tName);
213    remove_id_Column (tName, colName);
214    removeTable(tName)
215
216    END;
217
218 /*****************Implementation of attribute addition***********/
219
220 addAttribute (cName,attrName,type,exp) =
221
222    BEGIN
223       VAR
224          inh_Tables , tName , colName , colType , initialValue
225       IN
226          inh_Tables := tableHierarchy (cName) ;
227          tName  := classToTable (cName) ;
228          colName := propertyToColumn ( attrName ) ;
229          colType := sqlType ( type ) ;
230          initialValue := translate(exp);
231          alterTable (tName , colName) ;
```

```
232          addColumn (colName , tName , colType) ;
233          updateTable (tName);
234          setValue(colName,initialValue)
235
236       VAR
237          counter
238       IN
239          counter := 1 ;
240       WHILE
241          counter <= size (inh_Tables)
242       DO
243         VAR
244            current_table IN
245            current_table := inh_Tables (counter) ;
246
247            alterTable (current_table,colName) ;
248            updateTable (current_table) ;
249            setValue(colName,initialValue);
250            counter := counter + 1
251         END
252       INVARIANT
253          counter : 1..size(inh_Tables)+1 &
254          inh_Tables = tableHierarchy(cName) &
255          ran(inh_Tables) =
256             classToTable [closure1(superclass~) [{cName}]] &
257          {cc,att | cc: CLASS & att: PROPERTY &
258                att: ownedAttributes[closure1(superclass)[{cc}]] &
259                propertyToColumn(att): (tableColumns \/
260                {classToTable(cName) |-> propertyToColumn (attrName)})
261                   [{classToTable(cc)}]}
262          =
263          inheritedAttributes \/ (inh_Tables;classToTable~) [1..counter-1]*{attrName}
264       VARIANT size (inh_Tables)-counter
265       END
266    END
267   END
268  END ;
269
270  /*****************Implementation of attribute deletion***********/
271  deleteAttribute (cName , attrName) =
272
273     BEGIN
274        VAR inh_Tables , tName , colName,
275            counter, current_table
276        IN
277           inh_Tables := tableHierarchy (cName) ;
278           tName      := classToTable (cName) ;
```

```
279          colName := propertyToColumn (attrName) ;
280          counter := 1;
281          current_table := inh_Tables (counter);
282          alterTable (tName , colName) ;
283          removeColumn (colName , tName) ;
284      WHILE
285          counter < = size(inh_Tables)
286      DO
287          alterTable (current_table , colName) ;
288          removeColumn (colName , current_table) ;
289          updateTable (current_table) ;
290          counter := counter + 1
291      INVARIANT
292          counter : 1..size(inh_Tables)+1 &
293          inh_Tables = tableHierarchy(cName) &
294          ran(inh_Tables) =
295             classToTable [closure1(superclass~) [{cName}]] &
296          propertyToColumn (attrName)})
297             [{classToTable(cc)}]}
298          =
299          inheritedAttributes \/
300             (inh_Tables;classToTable~) [1..counter-1]*{attrName}
301       VARIANT
302          card ( inh_Tables ) - counter
303       END
304       END
305    END ;
306
```

Appendix D

Proof activities

Abstract machine proofs

The machine corresponding to the evolution metamodel has been proved with the AtelierB [13]. As we explained in Chapter 5, Section 5.4.1, from a theoretical point of view, one proof obligation is generated for each substitution statement of abstract machine operations. This proof obligation verifies that the execution of the operation maintains abstract machine invariant. In order to facilitate the establishment of this proof obligation, the prover operates simplification rules that lead to a great number of proofs which are easier to achieve than one large proof for each operation.

To carry out these proofs, AtelierB [13] includes two complementary proof tools. The first one is automatic, implementing a decision procedure that uses a set of deduction and rewriting rules. The second prover allows the users to enter into a dialogue with the automatic prover by defining their own deduction and/or rewriting rules that guide the prover to find the right way to discharge the proofs. To ensure the correctness of the interactive proofs, each added rule must be proved by the prover of the AtelierB before it can be used.

To ensure the correctness of abstract specifications, we have established the correctness of each abstract evolution operation using AtelierB prover. About 78 proofs were raised. The proof obligations generated concern the consistency constraints (formalized as abstract machine invariants) and model edits (formalized as abstract machine operations) as we described in Sections 5.2 and 5.3 respectively. With AtelierB (version 4.0), about 76% of these proofs have been automatically discharged. Proof obligations involving transitive closure operator (which we use to represent the inheritance hierarchy in the data model) were not discharged due to lack of inference rules within the prover to discharge proof obligations involving this operator.

Below, we present a representative sample of proof obligation examples that we had to prove manually. In each example, we first present the proof obligation generated by the prover, followed by a brief explanation of the basis on which the proof obligation was discharged. At the end of this appendix, we present a detailed example, showing in a step-by-step, how a particular proof obligation was discharged using AtelierB prover.

addClass. Proof Obligation No.5

```
1  Local hypotheses           &
2  c1: className \/ {cName}   &  c2: className \/ {cName}   &
3  not(c1 = c2)               &  c1|->c2: closure1(superclass\/{cName|->super}) &
4  "Check that the invariant
5  (!(c1,c2).(c1: className & c2: className &
6             c1/=c2 & c1|->c2: closure(superclass) =>
7              ownedProperties(c1) /\ ownedProperties(c2) = {}))
8  is preserved by addClass operation
9  =>
10 (ownedProperties \/ {cName |->{}})(c1) /\ (ownedProperties \/ {cName|->{}})(c2) = {}
```

Table D.1: PO of addClass edit vs. property name uniqueness

The goal of this proof obligation is to show that invariant property name uniqueness, which enforces property name uniqueness within data model class inheritance hierarchy is preserved by `addClass` model edit. Figure D.1 shows an example clarifying the goal of this proof obligation. In the diagram, figure D.1(a) shows an invalid state space as property `ax` is already defined in class `C3` which is a superclass of class `C1`, i.e. property `ax` is already inherited by `C1` and does not need to be defined in `C1` again. Figure D.1(b), on the other hand, shows a valid state space. Although property `ax` is defined in classes `C4` and `C5`, these two classes are not in the inheritance hierarchy of `C1`, so property name uniqueness is not violated.

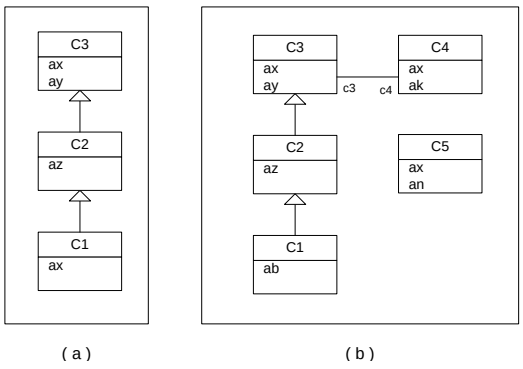


Figure D.1: Illustration of property name uniqueness

Proof. We discharge this proof obligation using case analysis. If `c1 = cName`, this means that `c1` is a fresh class name being added by `addClass` operaion and as such, `c1` did not have any properties before `addClass` operation executed. This gives us an empty set from the function application. If `c1 /= cName`, this means that `c1` used to exist in the data model before the operation executed. Since the invariant was preserved before the operation executed, we can safely assume that `cName /: dom(ownedProperties)`. Same case analysis is done for variable `c2`. In summary, if neither `c1` nor `c2` is `cName`, then their old values used to hold. If one of them is `cName`, then the function application gives the empty set, which shows that the invariant holds □

addClass. Proof Obligation No.6

```
1  Check that the invariant
2  (closure1(superclass)/\id(className) = {})
3  is preserved by addClass operation
4  =>
5  closure1(superclass \/ {cName |->super}) /\
6  id(className  \/ {cName})         = {}
```

Table D.2: addClass edit vs. absence of circular inheritance invariant

The goal of this proof obligation is to demonstrate that the absence of circular inheritance invariant is not violated by `addClass` edit. Note that in the abstract machine, we use variable `superclass`, defined as a partial function from `CLASS` to `CLASS` to identify the immediate superclass of a data model class. We use the transitive closure, denoted by `closure1` , applied on superclass function to identify other superclasses of the inheritance hierarchy. Assuming that the absence of circular inheritance invariant and the precondition of `addClass` operation hold, we would like to prove that the invariant still holds after the operation is executed.

Proof. We first apply deduction. The new goal becomes:

```
closure1(superclass\/{cName|->super}) /\
id(className \/ {cName}) = {}
```

Since AtleierB prover has not got enough inference rules to support discharging proof obligations involving transitive closure, a new rule is added to the prover to assert that `closure1` of `superclass` is a tree that has no cycles and that the newly added class is a new leaf (i.e. not a superclass itself). The added rule is stated as follows:

```
1  closure1(f) /\ id(E) = {} &
2  not(x:dom(f)) &
```

```
3 not(x=y)  =>
4 closure1(f \/ {x|->y}) /\ id (E \/{x}) = {}
```

This states that if the goal to be proved has the form specified in line 4 in the listing above, then this goal could be proved on the basis of hypotheses stated in lines 2-3. With the help of the interactive prover, the hypothesis in line 2 could be mapped to the precondition : `cName /: dom(superclass)`, and the hypothesis in line 3 to `cName : CLASS - className & super: className`. Accordingly, the goal is proved □

addClass. Proof Obligation No.8

```
1 Local hypotheses  &
2 cc :  dom(extent      \/ {cName|->{}})     &
3 aa : (ownedProperties \/ {cName|->{}})(cc) &
4 oo : (extent\/{cName|->{}})(cc) &
5 "Check that the invariant
6 (!(cc,aa,oo).(cc : dom(extent) & aa : ownedProperties(cc) &
7               oo : extent(cc) =>  typeOf(value(aa)(oo)) = propertyType(aa)))
8 is preserved by addClass operation  =>
9 typeOf(value(aa)(oo)) = propertyType(aa)
```

Table D.3: addClass edit vs. value conformance

The purpose of this proof obligation is to show that the type of each value assigned to an object in the data model should match the type given to the attribute, which the value instantiates. For example, we could have an attribute named age : NAT in Class Person. When the class is instantiated into objects (for example, Person is instantiated into p1 object), these objects can be assigned values corresponding to their class attributes (e.g., p1 can have value of 27 for the age attribute).

Proof. By case analysis. If `cc=cName`, we would get a false hypothesis for variable `aa` since, in this case, the function application would type `aa` by the the empty set. The same can be said about variable `oo`. Based on these two false hypotheses, we discharge this case. If `cc/=cName`, this implies the situation prior to the execution of the operation, and, hence, we could instantiate all the uiversally quantified variables of the invariant (i.e. `cc,aa,oo`) based on hypotheses already existing in the machine □

deleteAttribute. Proof Obligation No.5

This proof obligation ensures that typing invariant of machine values is not violated by attribute deletion.

```
"Check that the invariant
  (value: PROPERTY +-> (OBJECTID +-> VALUE))
    is preserved by deleteAttribute operation =>
       {attrName}<<|value: PROPERTY +-> (OBJECTID +-> VALUE)
```

Table D.4: deleteAttribute edit vs. value conformance

Proof. After deduction, the goal becomes :
`{attrName}<<|value: PROPERTY <-> (OBJECTID +-> VALUE)`
Since `dom(a): POW(s) & ran(a): POW(t) => a : s <-> t`, this goal can be re-written into two sub-goals:
Subgoal-1 : `dom({attrName}<<|value) <: PROPERTY` and
Subgoal-2 : `ran({attrName}<<|value) <: OBJECTID +-> VALUE`
Subgoal-1 can be simplified into `dom(value)-{attrName} <: PROPERTY`, which can be further simplified to:
`dom(value) <: PROPERTY\/{attrName}`, since `a: POW(c\/b) => a-b: POW(c)`.
This Subgoal can now be re-written as `dom(value) <: PROPERTY`, and discharged since it exists in machine hypothesis. Subgoal-2 can be simplified into :
`ran({attrName}<<|value) <: OBJECTID +-> dom(typeOf)`,
using equality of `VALUE = dom(typeOf)` from the machine. This Subgoal can be discharged since `(ran(r): POW(b)) => ran(a<<|r):POW(b))` □

addAssociation. Proof Obligation No.16

```
Local hypotheses&
srcOID = extent(owningClass(srcProp)) &
aa: dom(memberEnds\/{assoName|->{srcProp|->tgtProp}}) &
pp: dom((memberEnds\/{assoName|->{srcProp|->tgtProp}})(aa)) &
cc$0: dom((association\/{assoName|->{srcClass|->tgtClass}})(aa)) &
oo$0: extent(cc$0) &
Check that the invariant
(!(aa,pp).(aa: dom(memberEnds) &
           pp: dom(memberEnds(aa)) =>
             !(cc,oo).(cc: dom(association(aa)) &
               oo: extent(cc) => card(link(aa)(oo))>=propertyLowerMultip(pp) &
                                 card(link(aa)(oo))<=propertyUpperMultip(pp))))
is preserved by addAssociation operation
=>
card((link\/{assoName|->srcOID*{{exp}}})(aa)(oo$0))<=propertyUpperMultip(pp)
```

Table D.5: addAssociation edit vs. link conformance (cardinality)

The aim of this invariant is to ensure association link consistency. In the data model, each association is instantiated into a link. For example, the association named *weworksFor* between *Employee* Class and *Department* Class, is instantiated

into a link, we may call it *wf*. This link will be linking objects of the *Employee* Class (e.g. e1, e2, e3) to objects of the *Department* Class (e.g. d1, d2, d3). The cardinality of *wf* will be determined by the multiplicity of the association member end of the association which *wf* is an instance of (in this case, *worksFor*). In this example, *worksFor* which is an association name, has two member ends: *employees* property of *Employee* Class and *department* property of the *Department* Class. Each member end will have a lower and an upper multiplicity. Assume that the lower and upper multiplicity of the *department* end is 1 and 3 respectively, this means that an employee object can be linked to a minimum of 1 department object and a maximum of 3 department objetcs. In this proof obligation, we need to verify that `addAssociation` operation does not cause the number of linked objects to exceed the upper multiplicity of the association ends.

Proof. We discharge this proof using case analysis. We first assume that `aa` is a new association introduced by `addAssociation` operation, i.e. `aa = assoName`. In this case, the operation does not affect (increase or decrease) the cardinality of objects of the owning class of the association end, here `pp` , i.e. the number of these objects is determined outside the operation and read into the operation using `srcOID = extent(owningClass(srcProp))`. In addition, under the same assumption (i.e. `aa = assoName`), the operation does not affect the upper multiplicity of the association end (`pp` in this case). Since the operation affects neither the cardinality of linked objects nor the upper multiplicity of the property, we discharge the proof under this case. If `aa/=assoName`, this means that `aa` corresponds to an existing association name already existing in the data model. Since we assume that the invariant holds prior to the execution of the operation, we can discharge the proof under this case too □

Refinement proofs

As outlined in Section 2.4 in Chapter 2 (and demonstrated in Ch 5, Section 5.4.2), the invariants of the Refinement machines does not only define new variables introduced in refnement but also relates those variables to variables defined in the abstract machine (i.e. glue the two state spaces). Accordingly, for a refinement step to be valid, we need to ensure that every possible execution in the refinement machine must correspond to some execution of the abstract machine. In the following proof activity, we demonstrate that each of our refined evolution operations is a valid refinement of its corresponding operation at the abstract machine level. In other words, our main

task is to discharge the following proof obligation:

$\mathrm{I} \wedge \mathrm{J} \wedge \mathrm{P} \Rightarrow [\mathrm{S1}] \neg [S] \neg J$

Where I denotes the invariant of the abstract machine; J denotes the invariant of the refinement machine; P denotes the preconditions of the abstract machine; [S1] and [S] denote the execution of refinement machine and that of the abstract machine respectively.

Since we assume that the abstract machine substitution [S] is called within its precondition and given that $\neg S \ \neg J$ means that there exists at least one S that establishes J, our main focus will be to prove that each refinement operation satisfies the relevant linking invariants in the refinement machine which defines the properties of new variables in relation to abstract variables. Once such proof is established, we can conclude that the refined substitution is developed in accordance with the abstract specification.

To ensure the correctness of the refinement process we performed, we have established the correctness of each refinement rule using Atelier B prover [13]. About 120 refinement proofs were raised. With AtelierB (version 4.0), about 70% of these proofs have been automatically discharged, but this concerns only the easier proofs. The remaining proofs are rather hard and time-conuming. Fortunately, the generic feature of our refinement rules made it possible to define proof tactics that enable to automate the refinement proofs. This means that, once the proof of a generic refinement rule has been obtained, it is possible to reuse it in other instantiations of the rule. Below, we describe some of the proofs we had to perform manually using Atelier B prover.

addAssociation.Proof Obligation No.2

This proof obligation is raised to verify if `addAssociation` operation preserves the following linking invariant:

`owningClass = (ownedAttributes \/ ownedAssoEnds)~`

which relates `owningClass` variable, defined in the abstract machine to the union of the inverse of two sets: `ownedAttribute` and `ownedAssoEnds`. In other words, we need to ensure that any owned attribute or association end in the refined data model is still owned by a data model class. We need to ensure that this is the case after performing the refined substitution of `addAssociation` operation. The goal we need to prove is:

```
"‘addAssociation preconditions in this component’" &
assoName: ASSOCIATION &
not(assoName: associationName) &
srcClass: CLASS &
not(srcClass: dom(association(assoName))) &
tgtClass: CLASS &
not(tgtClass: ran(association(assoName))) &
srcProp: PROPERTY &

"‘Check that the invariant
(owningClass = (ownedAttributes \/ownedAssoEnds )~)
is preserved by addAssociaiton operation
=>
(ownedAttributes \/ (ownedAssoEnds \/
      {srcClass|->srcProp,tgtClass|->tgtProp}))~ =
owningClass\/{srcProp|->srcClass,tgtProp|->tgtClass}
```

Table D.6: PO No.2 of addAssociation refined operation

```
(ownedAttributes \/(ownedAssoEnds \/
   {srcClass|->srcProp,tgtClass|->tgtProp}))~ =
owningClass\/
   {srcProp|->srcClass,tgtProp|->tgtClass}
```

$\Leftrightarrow$

```
ownedAttributes~ \/ (ownedAssoEnds~ \/
   {srcProp|->srcClass,tgtProp|->tgtClass}) =
owningClass\/
   {srcProp|->srcClass,tgtProp|->tgtClass}
```

By replacing `(ownedAttributes \/ ownedAssoEnds)~` by `ownedAttributes~ \/ ownedAssoEnds~` in the LHS of equality, *(inverse of union is union of inverses)*

$\Leftrightarrow$

```
ownedAttributes~ \/ (ownedAssoEnds~ \/
   {srcProp|->srcClass,tgtProp|->tgtClass}) =
(ownedAttributes \/ ownedAssoEnds )~ \/
   {srcProp|->srcClass,tgtProp|->tgtClass}
```

By replacing the definition of `owningClass` *from the linking invariant*

$\Leftrightarrow$

```
ownedAttributes~ \/(ownedAssoEnds~ \/
   {srcProp|-> srcClass,tgtProp|->tgtClass}) =
ownedAttributes~ \/ ownedAssoEnds ~\/
   {srcProp|-> srcClass,tgtProp|->tgtClass}
```

```
1  "‘deleteClass preconditions in this component’" &
2  cName: className &
3  not(cName: ran(superclass)) &
4  ownedProperties[{cName}]/\ran(opposite) = {} &
5  "‘Local hypotheses’" &
6  classAssos = {asso | asso: ASSOCIATION
7          not(dom (associationTable(asso)) \/
8          ran (associationTable(asso)) /\
9          ownedAssoEnds[{cName}] = {})} &
10 subclassAssos = {asso | asso: ASSOCIATION &
11           not (dom (associationTable (asso)) \/
12           ran (associationTable(asso)) /\
13           inheritedAssoEnds[{cName}] = {})} &
14 "‘Check that the invariant
15 (propertyClass = (
16    ownedAttributes \/ inheritedAttributes \/
17    ownedAssoEnds   \/ inheritedAssoEnds)~)
18 is preserved by deleteClass operation "
19 =>
20 propertyClass |>>{cName} =
21    ({cName}<<|ownedAttributes \/
22    ({cName}<<|inheritedAttributes) \/
23    ({cName}<<|ownedAssoEnds)   \/
24    ({cName}<<|inheritedAssoEnds))~
```

Table D.7: PO No.13 of `deleteClass` refined operation

By replacing `(ownedAttributes \/ ownedAssoEnds)~` by `ownedAttributes~ \/ ownedAssoEnds~` in the RHS of equality *(inverse of union is union of inverses)*

⇔

```
ownedAttributes~ \/(ownedAssoEnds~ \/
   {srcProp|->srcClass,tgtProp|->tgtClass})
=
ownedAttributes~ \/ ownedAssoEnds~ \/
   {srcProp|->srcClass,tgtProp|->tgtClass}
```

associativity of union □

deleteClass.Proof Obligation No.13

This proof obligation is raised to verify if `deleteClass` refinement operation preserves the following linking invariant on `propertyClass` variable:

```
propertyClass = (
   ownedAttributes \/ inheritedAttributes \/
   ownedAssoEnds   \/ inheritedAssoEnds)~
```

where `propertyClass` is a refinement variable defined as a relation from `PROPERTY` to `CLASS` and used to refine abstract variable `owningClass` which was defined as a partial function from `PROPERTY` to `CLASS`. The reason for this refinement step was the flattening of inheritance where in Object-to-Relational refinement, the same property could be owned by multiple classes in the data model(see Section 6.1.2). The goal we have is:

```
propertyClass |>>{cName} =
   ({cName}<<|ownedAttributes      \/
   ({cName}<<|inheritedAttributes) \/
   ({cName}<<|ownedAssoEnds)       \/
   ({cName}<<|inheritedAssoEnds))~
```

$\Leftrightarrow$

```
 (ownedAttributes \/ inheritedAttributes \/
  ownedAssoEnds   \/ inheritedAssoEnds)~ |>>{cName} =
  ownedAttributes~     |>>{cName}  \/
  (inheritedAttributes~|>>{cName}) \/
  (ownedAssoEnds~      |>>{cName}) \/
  (inheritedAssoEnds~  |>>{cName})
```

By using the definition of `propertyClass` *in the linking invariant*
$\Leftrightarrow$

```
(ownedAttributes~  \/ inheritedAttributes~ \/
 ownedAssoEnds~    \/ inheritedAssoEnds~) |>>{cName} =
(ownedAttributes~ |>>{cName}  \/ (inheritedAttributes~|>>{cName}) \/
(ownedAssoEnds~   |>>{cName}) \/ (inheritedAssoEnds~  |>>{cName})
```

By replacing `(a\/b)~` by `(a~\/b~)`
$\Leftrightarrow$

```
(ownedAttributes~  \/ inheritedAttributes~ \/
 ownedAssoEnds~    \/ inheritedAssoEnds~) |>>{cName} =
(ownedAttributes~ |>>{cName}  \/ (inheritedAttributes~|>>{cName}) \/
(ownedAssoEnds~   |>>{cName}) \/ (inheritedAssoEnds~  |>>{cName})
```

By replacing `(a \/ b)|>> c` by `(a |>> c \/ b |>>c)`
$\Leftrightarrow$

```
(ownedAttributes~|>>{cName} \/ inheritedAttributes~|>>{cName} \/
ownedAssoEnds~ |>> {cName}  \/ inheritedAssoEnds~  |>>{cName}) =
ownedAttributes~|>>{cName}  \/ (inheritedAttributes~|>>{cName}) \/
(ownedAssoEnds~|>>{cName})  \/ (inheritedAssoEnds~|>>{cName})
```

□

addAtribute.Proof Obligation No.2

```
1  "‘addAttribute preconditions in this component’" &
2  cName: CLASS &
3  cName: className &
4  attrName: PROPERTY &
5  not(attrName: propertyName) &
6  not(cName = owningClass(attrName)) &
7  not(attrName: ownedProperties[{cName}]) &
8  not(attrName: ownedProperties[closure1(superclass)[{cName}]]) &
9  not(attrName: dom(propertyType)) &
10 not(attrName: dom(value)) &
11 type: ran(primitiveType) &
12 exp: VALUE &
13 typeOf(exp) = type &
14 not(cName: ran(superclass )) &
15 "‘Check that the invariant
16 (owningClass = (ownedAttributes \/ ownedAssoEnds)~)
17 is preserved by addAttribute " &
18 =>
19 (ownedAttributes \/ {cName|->attrName} \/
20   ownedAssoEnds )~ =
21     owningClass <+ {attrName|->cName}
```

Table D.8: PO No.2 of `addAttribute` refined operation

This proof obligation is raised to verify if `addAttribute` refinement operation preserves the linking invariant on `owningClass`, as stated in Table D.9. The initial goal is:

```
(ownedAttributes \/ {cName|->attrName} \/ ownedAssoEnds)~ =
   owningClass <+ {attrName|->cName}
```

⇔

```
(ownedAttributes \/ {cName|->attrName}\/ownedAssoEnds)~ =
(ownedAttributes \/ ownedAssoEnds)~<+{attrName|->cName}
```

By using equality of `owningClass`

This equality is true because `attrName` is not an existing attribute in the data model (otherwise the union operator `\/` and overriding operator `<+` would not be equivalent) This can be proved using the hypothesises:

```
dom(owningClass) <: propertyName    (from invariant)
not(attrName: propertyName)         (from preconditions)
```

addClass.Proof Obligation No.7

```
1  "‘addClass preconditions in this component’" &
2    cName: CLASS &
3    not(cName: className) &
4    not(cName: dom(extent)) &
5    not(cName: dom(superclass)) &
6  ...
7  "‘Local hypotheses’" &
8    isAbst = FALSE &
9    cKey: INTEGER &
10   not(cKey: ran(classKey)) &
11 "‘Check that the invariant
12 (inheritedAssoEnds: CLASS <-> PROPERTY)
13 is preserved by addClass operation" &
14 =>inheritedAssoEnds \/
15        {cName}*ownedAssoEnds [{closure1(superclass) (cName)}]:
16                 CLASS <-> PROPERTY
```

Table D.9: PO 7 of `addClass` refinement operation

The main purpose of this proof obligation is to ensure that `addClass` refinement operation does not violate the typing invariant of `inheritedAssoEnds` variable after performing substitution. After deduction, the goal becomes:

```
inheritedAssoEnds \/
   {cName}*ownedAssoEnds [{closure1(superclass) (cName)}]:
            CLASS <-> PROPERTY
```

$\Leftrightarrow$

```
inheritedAssoEnds : CLASS <-> PROPERTY &
    {cName}* ownedAssoEnds [{closure1(superclass)(cName)}] :
            CLASS <-> PROPERTY
```

By applying `a: s <-> t & b: s <-> t => a\/b: s <-> t`

The first part of the goal: `inheritedAssoEnds: CLASS <-> PROPERTY` is discharged because it exists in hypothesis. The second part of the goal can be rewritten as :

```
not({cName} = {}) &
not(ownedAssoEnds[{closure1(superclass)(cName)}] = {}) =>
   {cName} <: CLASS &
    ownedAssoEnds[{closure1(superclass)(cName)}] <: PROPERTY
```

By applying `(not(a = {}) & not(b = {}) =>`
`a: POW(s) & b: POW(t)) => a*b: s <-> t`

After deduction, the goal becomes:

```
{cName} <: CLASS &
    ownedAssoEnds[{closure1(superclass)(cName)}] <: PROPERTY
```

The first part of this goal: `{cName} <: CLASS` can be simplified to `cName:CLASS` and discharged because it exsits in hypothesis. The second part of the goal:
`ownedAssoEnds[{closure1(superclass)(cName)}] <: PROPERTY`
Can be rewritten as:

```
(ownedProperties|>propertyType~[ran(classType)])
  [{closure1(superclass)(cName)}] <: PROPERTY
```

By using definition `ownedAssoEnds = ownedProperties |>propertyType~ [ran(classType)]` from the linking invariants of the Refinement machine.
⇔

```
ownedProperties[{closure1(superclass)(cName)}] /\
propertyType~[ran(classType)] <: PROPERTY
```

By applying `binhyp(a: POW(c))  => a/\b: POW(c)`, this goal can be discharged because both parts of the set intersection exist in the hypothesis □

addClass.Proof Obligation No.9

```
1  "‘Check that the invariant
2  (dom(classKey) = isAbstract~[{FALSE}])
3  is preserved by addClass operation " &
4   =>
5   dom(classKey\/{cName|->cKey}) =
6          (isAbstract\/{cName|->isAbst})~[{FALSE}]
```

Table D.10: PO 9 of `addClass` refinement operation

The main purpose of this proof obligation is to ensure that `addClass` refinement operation does not violate the linking invariant on `classKey` variable. After deduction, goal is :

```
dom(classKey    \/ {cName|->cKey}) =
    (isAbstract \/{cName|->isAbst})~[{FALSE}]
```

⇔

```
dom(classKey)     \/ {cName} =
     (isAbstract~ \/ {isAbst|->cName})[{FALSE}]
```

By distributing the inverse operator over both parts of set union

⇔

```
isAbstract~ [{FALSE}]   \/ {cName} =
            (isAbstract~ \/ {FALSE|->cName})[{FALSE}]
```

By using `dom(classKey) = isAbstract~[{FALSE}]` *from its definition in the linking invariants of the Refinement machine*

⇔

This goal can be rewritten as:

```
FALSE: {FALSE} =>
    isAbstract~[{FALSE}] \/ {cName} =
          isAbstract~[{FALSE}]\/{cName}
&
not(FALSE: {FALSE}) =>
    isAbstract~[{FALSE}] \/ {cName} =
           isAbstract~[{FALSE}]
```

The first part of the goal can obviously be discharged. Applying deduction, the second goal becomes:

```
isAbstract~[{FALSE}] \/ {cName} = isAbstract~[{FALSE}]
```

which can also be discharged because there are contradictory hypothesis.

deleteAssociation.Proof Obligation No.13

This proof obligation is raised to verify `deleteAssociation` refinement operation does not violate the linking invariant on `inheritedAssoEnds` refinement variable. In particular, we would like to ensure that, after deleting an association where a named superclass participates (either as a source class or a target class of the association), the `inheritedAssoEnds` variable of that class will be empty after deleting the association.

After deduction, goal can be stated as:

```
"deleteAssociation preconditions in this component" &
assoName: ASSOCIATION &
"‘Local hypotheses’" &
srcProp = {srcP | srcP: PROPERTY &
           srcP:  dom(associationTable (assoName))} &
tgtProp = {tgtP | tgtP: PROPERTY &
           tgtP: ran (associationTable(assoName))} &
cc: className &
not(cc: dom(superclass)) &
"‘Check that the invariant
(!cc.(cc: className & cc/:dom(superclass) => inheritedAssoEnds[{cc}] = {}))
is preserved by deleteAssociation operation "
=>
(inheritedAssoEnds|>>(srcProp\/tgtProp))[{cc}] = {}
```

Table D.11: PO 13 of `deleteAssociation` refinement operation

```
(inheritedAssoEnds |>>(srcProp \/ tgtProp))[{cc}] = {}
```

This goal can be simplified as:

```
inheritedAssoEnds[{cc}]-srcProp <: tgtProp
```

where `tgtProp` is a set defined as:

```
tgtProp = {tgtP | tgtP: PROPERTY & tgtP: ran (associationTable(assoName))}
```

⇔

```
 inheritedAssoEnds[{cc}] <: tgtProp\/srcProp
```

By applying `a: POW(c\/b)=> a-b: POW(c)`.

⇔

```
inheritedAssoEnds[{cc}] <: {tgtP | tgtP: PROPERTY &
    tgtP: ran(associationTable(assoName))} \/ srcProp
```

By applying the equality of `tgtProp`.

⇔

```
inheritedAssoEnds[{cc}] <:
    {tgtP | tgtP: PROPERTY &
            tgtP: ran (associationTable(assoName))} \/
    {srcP | srcP: PROPERTY &
            srcP: dom(associationTable(assoName))}
```

By applying the equality of `srcProp`.

⇔

```
{} <: {tgtP | tgtP: PROPERTY &
          tgtP: ran(associationTable(assoName))} \/
       {srcP | srcP: PROPERTY &
          srcP: dom(associationTable(assoName))}
```

By applying the equality of `inheritedAssoEnds[{cc}] = {}`.

Accordingly, this goal can be discharged □

Implementation proofs

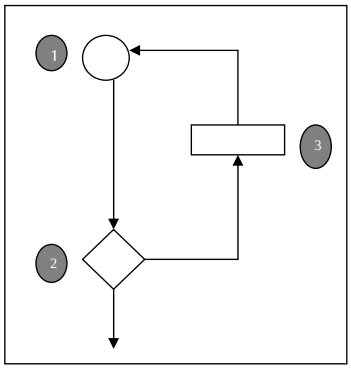
As outlined in Section 2.4, implementation proof obligations are similar to those of refinement. Since we have already demonstrated refinement proofs in the previous section, we do not show implementation proofs. What we need to show, however, is a demonstration of correctness of implementation WHILE-loops, as theses constructs appear only in implementation and have their own proof obligations. Below, we present an example for demonstrating WHILE-loop correctness. Discharging proof obligations of this loop has been done interactively using AtelierB prover based on the five conditions (proof obligations), shown in Figure 2.10 which, together, imply that the loop is correct.

We first present the proof obligation generated by the prover, followed by a brief explanation of the basis on which the proof obligation was discharged. We use the WHILE-loop of `addAttribute` operation, as an example.

Implementation. addAttribute

Within the context of interpreting `addAttribute` operation to corresponding SQL operations, the main purpose of this loop is to add a column (mapped to the attribute being added) to all tables in SQL model which correspond to subclasses of the attribute owning class (identified by input parameter `cName`). For example, adding an attribute officeNo. to superclass Person would require adding the same attribute to Person subclasses : Employee and Freelance. In SQL terms this requires adding a column to tables corresponding to the two subclasses.

Figure D.2 shows the main components of the loop. Loop *initialization* (marked with number 1) consists of declaring a local variable (`counter`) as a loop index and initializing it with 1 as a value. *WHILE-test* (marked with number 2) compares the value of the local variable `counter` with the size of `inh_Tables` as a condition for executing the loop. *Loop-body* (marked with number 3) fetches a table from the




```
1   VAR counter IN
2       counter := 1 ;
3
4   WHILE
5      counter <= size (inh_Tables)
6   DO
7      VAR current_table IN
8          current_table := inh_Tables(counter) ;
9          alterTable (current_table , colName) ;
10         updateTable(current_table)           ;
11         counter    := counter + 1
12     END
13  INVARIANT
14     counter : 1..size(inh_Tables)+1 &
15     inh_Tables = tableHierarchy(cName) &
16     ran(inh_Tables) = classToTable[closure1(superclass~)[{cName}]] &
17     {cc, att | cc : CLASS &
18        att:PROPERTY &
19        classToTable(cc) : ran(tableHierarchy(cc)) &
20        propertyToColumn(att) : tableColumns[{classToTable(cc)}] } =
21        inheritedAttributes \/(inh_Tables;classToTable~)
22                                [1..counter-1]*{attrName}
23  VARIANT size (inh_Tables)-counter
24  END
```

Figure D.2: Illustration of main elements of B Implementation loop

sequence of inherited tables, and performs an `alterTable` and `updateTable` SQL operations before increasing the `counter` by 1. The loop *invariant* consists of four conjuncts (lines 14-22) that asserts conditions which need to remain true before the loop starts, while the loop progresses and after the loop terminates. These predicates play a central role in verifying the loop correctness. For the sake of presentation, we will only consider the fourth invariant conjunct (lines 17-22). Other loop invariant conjuncts can be proved similarly. Finally, the loop *variant* (line 23) consists of an expression that must evaluate to a natural number and used to control the number of loop iterations.

The fourth loop invariant conjunct asserts the equality of two expressions. To the Left Hand Side (LHS) of the equality is a set comprehension charcterizing two properties of inherited attributes. First, it asserts that an *inherited* attribute of a class in the data model is an *owned* attribute of the class superclass. Second, it asserts that mapping an inherited attribute to an SQL column should be in `tableColumns` set union the columns corresponding to attributes owning class. To the Right Hand Side (RHS) of the equality is a set union expression formulating the property that, within the context of `addAttribute` operation, the existing set of `inheritedAttributes` will be unioned with the set consisting of the cartesian product where the first maplet is a class resulting from the mapping of inherited tables to the attribute name.

Given the proof obligations specified in Figure 2.10, to verify the correctness of this loop, it is sufficient to prove that:

$$[S](PO1 - PO5) \tag{D.1}$$

where [S] represents the substitution of the loop (lines 7-11, marked with number 3) and (PO1 - PO5) represents the five proof conditions, associated with WHILE loops. In other words, we need to establish that:

$$[S](PO1 \wedge PO2 \wedge PO3 \wedge PO4 \wedge PO5) \tag{D.2}$$

which is equivalent to establishing each of the conditions separately. We need to consider each of the conditions in turn.

(PO1) $[S_0]I$

The first proof obligation simply requires that the loop invariant holds in the initial state (i.e.*loop-initialization*). This proof obligation can be written as :

```
1 [counter := 1]
2 ({cc,att | cc: CLASS & att: PROPERTY &
3             att: ownedAttributes[closure1(superclass)[{cc}]] &
4              propertyToColumn(att): (tableColumns\/
5                     {classToTable(cName) |->
6                           propertyToColumn (attrName)})[{classToTable(cc)}]}
7 =
8 inheritedAttributes \/(inh_Tables;classToTable~)[1..counter-1]*{attrName})
```

To simplify presentation, we will denote the expression to the left hand side of the loop invariant (defined by the set comprehension, lines 2-6 above) with `IHAttribs`. Hence, we want to prove that:

```
IHAttribs = inheritedAttributes\/
               (inh_Tables;classToTable~)[1..counter -1]*{attrName}
```

$\Leftrightarrow$

```
IHAttribs = inheritedAttributes\/
               (inh_Tables;classToTable~)[1..0]*{attrName}
```

(replacing counter by its value)

$\Leftrightarrow$

```
IHAttribs = inheritedAttributes\/
              (tableHierarchy(cName);classToTable~)[{}]*{attrName}
```

(Cardinality lower bound is greater than cardinality upper bound)

⇔

`IHAttribs = inheritedAttributes\/{}`

(First element of cartisian product is an empty set)

⇔

`IHAttribs = inheritedAttributes`

(set union with an empty set)

Keeping in mind that, before the loop has been initialized, the `attrName` was added as a column to the table corresponding to its owning class (identified by input parameter `cName`), we now need to prove that, at the entry of the loop, `IHAttribs` defining the loop invariant is equivalent to the set comprehension defining `inheritedAttributes` in the linking invariant, i.e.

```
IHAttribs =
   {cc, att |cc : CLASS & att : PROPERTY &
         att : ownedAttributes[closure1(superclass)[{cc}]]
            propertyToColumn(att) : (tableColumns \/
               {classToTable(cName)|->
                  propertyToColumn (attrName)})[{classToTable(cc)}]}
```

which is true because the attribute that was added before the loop starts was not added to tables corresponding to owning class subclasses, i.e.

`attrName /: ownedAttributes[closure1(superclass)[{cName}]]`

Accordingly, this proof obligation can be discharged □

(PO3) $\forall X \cdot (I \Rightarrow v \in \mathbb{N})$

The main purpose of this proof obligation is to ensure that the `variant` expression provided as part of the loop definition evaluates to a natural number. Hence, we need to prove that:

```
1 ({cc,att | cc: CLASS & att: PROPERTY &
2             att: ownedAttributes[closure1(superclass)[{cc}]] &
3              propertyToColumn(att): (tableColumns\/
4                     {classToTable(cName) |->
5                           propertyToColumn (attrName)})[{classToTable(cc)}]}
6 =>
7 size(inh_Tables)-counter+1 : NAT
```

Using deducation, our goal becomes :

`size(inh_Tables) - counter + 1 : NAT`

Assuming that the loop invariant holds, we have the first invariant property of the loop invariant as : `counter : 1..size(inh_Tables)+1`
which implies that the maximum value `counter` can get would be `size(inh_Tables)+1`, as a result, the variant of the loop will remain, indeed, in $\mathbb{N}$ $\square$

(PO4) $\forall X \cdot (I \wedge P \Rightarrow [n := v;\ S](v < n)$

This proof obligation requires that the loop body should decrease the variant. Assuming that the loop invariant is true and the loop condition is true, we need to show that, as the loop progresses, the new value of the variant is less than the old value. Hence, we consider that :
$[n := v;\ S](v < n)$, with n here acting as a temporary variable for holding the old value of the variant before the substitution is performed, i.e. we need to prove that:

```
1 size(inh_Tables) - (counter+1) + 1 < size(inh_Tables) - (counter + 1)
```

which is clearly true. $\square$

(PO2) $\forall$ **X** $\cdot$ **(I** $\wedge$ **P** $\Rightarrow$ **[S] I)**
This proof obligation requires that whenever the invariant `I` and the loop condition `P` both are true, then the substitution `S` in the loop body is guaranteed to establish `I`. This requires that:

```
1  ({cc,att | cc: CLASS & att: PROPERTY &
2           att: ownedAttributes[closure1(superclass)[{cc}]] &
3              propertyToColumn(att): (tableColumns\/
4                 {classToTable(cName) |->
5                    propertyToColumn (attrName)})[{classToTable(cc)}]}
6  =
7  inheritedAttributes \/
8     (inh_Tables;classToTable~) [1..counter-1]*{attrName})
9  &
10 (counter <= size(inh_Tables)))
11 =>
12 ({cc,att | cc: CLASS & att: PROPERTY &
13             att: ownedAttributes[closure1(superclass)[{cc}]] &
14               propertyToColumn(att): (tableColumns \/
15                   {tableHierarchy(cName)(counter)|->
16                        propertyToColumn(attrName)})[{classToTable(cc)}]}
17 =
```

```
18 inheritedAttributes\/
19      (tableHierarchy(cName);classToTable~)[1..counter+1-1]*{attrName})
```

Note that in the above proof obligation the invariant property appearing at the antecedent is the invariant property before applying substitution, while the invariant property at the consequent is the invariant property after applying substitution (one step of a loop iteration). To simplify presentation, we will write `IHAttribs` to denote the set comprehension apprearing in lines 1-5 above.

We have

```
1..counter + 1 - 1
= 1..counter
=1..counter-1 \/ { counter }
```

which gives us

```
IHAttribs  =
inheritedAttributes \/
 (tableHierarchy(cName);classToTable~)[1..counter-1\/{counter}]*{attrName}
```

(replacing ***counter*** *in RHS of the equality)*

$\Leftrightarrow$

```
IHAttribs =
inheritedAttributes \/
    (tableHierarchy(cName);classToTable~)[1..counter-1]  * {attrName} \/
        (tableHierarchy(cName);classToTable~)[{counter}] * {attrName}
```

(By applying `f[A\/B]== f[A]\/f[B]`*)*

$\Leftrightarrow$

```
IHAttribs =
{cc,att | cc: CLASS & att: PROPERTY &
          att: ownedAttributes[closure1(superclass)[{cc}]] &
            propertyToColumn(att): tableColumns{classToTable(cc)}]} \/
                  (tableHierarchy(cName);classToTable~)[{counter }]*{attrName}
```

(expanding the first part of the set union using set comprehension)

$\Leftrightarrow$

```
IHAttribs =
{cc,att | cc: CLASS & att: PROPERTY &
            att: ownedAttributes[closure1(superclass)[{cc}]] &
              propertyToColumn(att): tableColumns [{classToTable(cc)}]} \/
                {classToTable~(tableHierarchy(cName)(counter))}*{attrName}
```

(applying function compositionin the second part of the set union)

$\Leftrightarrow$

```
{cc,att | cc: CLASS & att: PROPERTY &
          att: ownedAttributes[closure1(superclass)[{cc}]] &
            propertyToColumn(att): tableColumns$2[{classToTable(cc)}]} \/
```

(re-writing the second part of the set union as a set comprehension)

which is equivalent to `IHAttribs`. $\square$

(PO5) $\forall X \cdot (I \wedge \neg P \Rightarrow R)$

This proof obligation requires that the postcondition result R holds on exit of the loop. R corresponds to the behavior of equivalent operation (i.e. `addAttribute` in the refinement). For `inheritedAttribute` property that we are interested in, the equivalent refinement substitution is:

```
R = inheritedAttributes \/
          closure1 (superclass ~) [{cName}] * {attrName}
```

considering the loop invariant I, we have the following four predicates:

```
1  I1. counter : 1..size(inh_Tables)+1 &
2  I2. inh_Tables = tableHierarchy(cName) &
3  I3. ran(inh_Tables) = classToTable[closure1(superclass~)[{cName}]] &
4  I4.({cc,att | cc: CLASS & att: PROPERTY &
5             att: ownedAttributes[closure1(superclass)[{cc}]] &
6                   propertyToColumn(att): (tableColumns\/
7                         {classToTable(cName) |->
8                            propertyToColumn (attrName)})[{classToTable(cc)}]})
9  = inheritedAttributes \/
10       (inh_Tables;classToTable~) [1..counter-1]*{attrName}
```

To simplify presentation, we will denote the set comprehension at lines 4-8 above with `IHAttribs`. Assuming that the above invarinat properties hold, our goal can be stated as:

`I & not(P)=> R`, which gives us the following proof obligation:

```
1  not(counter <= size(inh_Tables)) & (not(P)) &
2  counter : 1..size(inh_Tables)+1  &
3  inh_Tables = tableHierarchy(cName) &
4  ran(inh_Tables) = classToTable[closure1(superclass~)[{cName}]] &
5  IHAttribs = inheritedAttributes \/
6     (inh_Tables ; classToTable~) [1..counter-1] * { attrName }
7  =>
8  inheritedAttributes \/ closure1(superclass~)[{cName}]*{attrName}
```

From the linking invariant on `inheritedAttributes`, we have:

```
{cc,att | cc: CLASS & att: PROPERTY &
             att: ownedAttributes[closure1(superclass)[{cc}]] &
                 propertyToColumn(att): (tableColumns\/
                     {classToTable(cName) |->
                          propertyToColumn (attrName)})[{classToTable(cc)}]}
```

which is equivalent to `IHAttribs`, i.e. we have:

`IHAttribs = R`

(using linking invariant on inheritedAttributes)

From `I1.` and `not(P)`, we can deduce:

`counter = size(inh_Tables)+1`

⇔

```
inheritedAttributes \/
    (inh_Tables ; classToTable~) [1..counter-1]*{attrName}
= R
```

(given I4, replacing the LHS of goal equality)

⇔

```
inheritedAttributes \/
   classToTable~[inh_Tables[1..counter-1]] * { attrName }
= R
```

(applying the function composition on the LHS of the equality)

⇔

```
inheritedAttributes \/
  classToTable~[inh_Tables[1..size(Inh_Tables)+1-1]] * { attrName }
= R
```

(replacement of counter)

⇔

```
inheritedAttributes \/
   classToTable~[inh_Tables[1..size(Inh_Tables)]] * { attrName }
= R
```

(applying +1 and -1 on the LHS of the equality)

⇔

```
inheritedAttributes \/
  classToTable~[ran(inh_Tables)]] * { attrName }
= R
```

(replacmeent of inhTables[1..size(InhTables)] by ran(InhTables))

$\Leftrightarrow$

```
InheritedAttributes \/
   classToTable~[classToTable[closure1(superclass~)
      [{cName}]]] * { attrName }
= R
```

(using loop invariant hypothesis I3)

$\Leftrightarrow$

As `classToTable` is injective (property `classToTable~ : TABLE +-> CLASS`), and that `closure1(superclass~[{cName}])` is within the domain of `classToTable`, we have:

```
classToTable~[classToTable[closure1(superclass~)[{cName}]]] =
   closure1(superclass~)[{cName}]
```

Replacing it in the goal gives:

```
inheritedAttributes \/
   classToTable~[classToTable[closure1(superclass~)[{cName}]]]* {attrName}
= R
```

$\Leftrightarrow$

```
inheritedAttributes \/ closure1(superclass~)[{cName}] =R
```

□

Proof Step	Atelier B Command	Description of the command
1	dc - do case	Starts up a proof by case
2	ss - simplify set	Simplifies set expressions appearing in the goal
3	mp - mini proof	Allows to use prover without proof by case
4	fh - false hypothesis	Proves that a hypo. is contradictory to others
5	pp(rp.1) - predicate prover	Applies the prover on selected hypotheses
6	dd - direct deduction	Proves the goal under a stated hypothesis
7	ss - simplify set	same as step 2
8	ph - particulariz hypothesis	Assigns values to variables appearing in hypotheses
9	ah - add hypothesis	Adds a predicate in the hypotheses stack
10	pp(rp.1) - predicate prover	same as step 5
11	pr - prove	Calls the automatic prover
12	se - suggest for existential	Instantiates variables under the scope of exist. quantif.

Table D.12: Overview of AtelierB proof steps for proof obligation addAssociation.PO12

Detailed Proof Illustration Using AtelierB Prover

Overview

This section shows a detailed example of how we used AtelierB [**?**] interactive prover to discharge abstract machine proof obligations. We use the proof obligation of `addAssociation` primitive, outlined in Section 5.3 to illustrate the steps. We first provide a narrative description of the proof goal, and then go through the steps we followed using the tool. The initial goal to be proved is listed below.

This goal refers to link conformance invariant, outlined in Section 5.2. The aim of this invariant is to assert association bi-directionality property in the data model. We would like to verify that this data model property is preserved by `addAssociation` operation. In the data model, each association has an `assoName` and two association `memberEnds`. If `p1` and `p2` are two association `memberEnds`, linked by an association named, e.g. `aa1` and, in addition, these two properties are in *opposite* relationship (i.e. pointing to each other), this implies that there is another association name (e.g. `aa2`), linking the same two properties, in the opposite direction. For example, assume that we have two classes *Employee* and *Department*, and an association between them, named *worksFor* linking *employees* property of *Employee* Class to *department* property of *Department* Class and these two properties are in *opposite* relation. This would imply that there is another association, named e.g. *employedBy* linking the same two properties but in a different order.

Proof Steps

This section explains the corresponding proof steps that has been performed using AtelierB. It goes through each proof command explaining how the command relates

```
1  Local hypotheses
2          srcOID = extent(owningClass(srcProp)) &
3          aa1: dom(memberEnds\/{assoName|->{srcProp|->tgtProp}}) &
4          p1: dom((memberEnds\/{assoName|->{srcProp|->tgtProp}})(aa1)) &
5          p2: ran((memberEnds\/{assoName|->{srcProp|->tgtProp}})(aa1)) &
6          p1|->p2: opposite &
7  =>
8          #aa2.(aa2: dom(memberEnds\/{assoName|->{srcProp|->tgtProp}}) &
9          (memberEnds\/{assoName|->{srcProp|->tgtProp}})(aa2) = {p2|->p1})
```

Table D.13: addAssociation edit vs. link conformance (opposite)

to the illustration.

Step 1 - dc(aa1 = assoName)

```
1  Local hypotheses
2          srcOID = extent(owningClass(srcProp)) &
3          assoName: dom(memberEnds\/{assoName|->{srcProp|->tgtProp}}) &
4          p1: dom((memberEnds\/{assoName|->{srcProp|->tgtProp}})(assoName)) &
5          p2: ran((memberEnds\/{assoName|->{srcProp|->tgtProp}})(assoName)) &
6          p1|->p2: opposite &
7  =>
8          #aa2.(aa2: dom(memberEnds\/{assoName|->{srcProp|->tgtProp}}) &
9          (memberEnds\/{assoName|->{srcProp|->tgtProp}})(aa2) = {p2|->p1})
```

Table D.14: Proof goal and hypotheses after proof step 1

This command starts the case analysis. The current goal is the case where `aa1` is equal to `assoName`, and when this goal will be discharged, the second case `aa1 /= assoName` will have to be discharged. Note that under the current case (i.e. `aa1=assoName`), variable `aa1` has been replaced by `assoName`. This can be observed by comparing hypotheses and goals in Table D.13 to those in Table D.14.

Step 2 - ss

```
1  Local hypotheses &
2          srcOID = extent(owningClass(srcProp)) &
3          p1 = srcProp & p2 = tgtProp & p1|->p2: opposite &
4  =>
5          #aa2.(aa2: dom(memberEnds)\/{assoName} &
6          (memberEnds\/{assoName|->{srcProp|->tgtProp}})(aa2) = {p2|->p1})
```

Table D.15: Proof goal and hypotheses after proof step 2

The command `ss` (simplify set) is used to simplify the goal. It is able to deduce that `p1 = srcProp` and `p2 = tgtProp`, as can be seen in line 3 in Table D.15.

Step 3 - mp

```
Local hypotheses &
  ran(memberEnds) <: PROPERTY +-> PROPERTY &
        dom(memberEnds) <: ASSOCIATION &
        memberEnds: ASSOCIATION <-> (PROPERTY +-> PROPERTY) &
        srcProp|->tgtProp: PROPERTY*PROPERTY & tgtProp|->srcProp: PROPERTY*PROPERTY &
        p1 = srcProp & p2 = tgtProp &
        srcProp|->tgtProp: opposite & srcOID = extent(owningClass(srcProp)) &
=>
  #aa2.(aa2: dom(memberEnds)\/{assoName} &
  (memberEnds\/{assoName|->{srcProp|->tgtProp}})(aa2) = {tgtProp|->srcProp})
```

Table D.16: Proof goal and hypotheses after proof step 3

Command `mp` has normalized the hypotheses and generated additional hypotheses. We can see that new hypotheses exist that were not part of the goal, by comparing Table D.15 to Table D.16.

Step 4 - fh(srcProp,tgtProp:opposite)

```
not(srcProp|->tgtProp: opposite)
```

Table D.17: Proof goal after proof step 4

This command, false hypothesis `fh`, starts a proof by contradiction. It is used to prove that the current case is an impossible case, by proving that one of the hypotheses in Table D.16 introduces a contradiction. The current goal is replaced by the negation of the hypothesis, as can be seen in Table D.17.

Step 5 - pp(rp.1)

```
not(aa1 = assoName) =>
  ("'Local hypotheses'" &
     srcOID = extent(owningClass(srcProp)) &
       aa1: dom(memberEnds\/{assoName|->{srcProp|->tgtProp}}) &
       p1 : dom((memberEnds\/{assoName|->{srcProp|->tgtProp}})(aa1)) &
       p2: ran((memberEnds\/{assoName|->{srcProp|->tgtProp}})(aa1)) &
       p1|->p2: opposite &
=>
       #aa2.(aa2: dom(memberEnds\/{assoName|->{srcProp|->tgtProp}}) &
       (memberEnds\/{assoName|->{srcProp|->tgtProp}})(aa2) = {p2|->p1}))
```

Table D.18: Proof goal and hypotheses after proof step 5

This command runs the predicate prover on the goal, adding all the hypothesis that have one symbol in common with the goal. As the goal is proved, it is replaced

```
1  Local hypotheses &
2          srcOID = extent(owningClass(srcProp)) &
3          aa1: dom(memberEnds\/{assoName|->{srcProp|->tgtProp}}) &
4          p1: dom((memberEnds\/{assoName|->{srcProp|->tgtProp}})(aa1)) &
5          p2: ran((memberEnds\/{assoName|->{srcProp|->tgtProp}})(aa1)) &
6          p1|->p2: opposite &
7  =>
8          #aa2.(aa2: dom(memberEnds\/{assoName|->{srcProp|->tgtProp}}) &
9          (memberEnds\/{assoName|->{srcProp|->tgtProp}})(aa2) = {p2|->p1})
```

Table D.19: Proof goal and hypotheses after proof step 6

by the second case. i.e. the case where `aa1 /= assoName`, as can be seen in line 1 in Table D.18.

Step 6 - dd

This command has moveed the hypothesis `not(aa1 = assoName)`, generated in the previous step as part of the case analysis, to the global hypotheses.

Step 7 - ss

```
1  Local hypotheses&
2          srcOID = extent(owningClass(srcProp)) &
3          aa1: dom(memberEnds)\/{assoName} &
4          p1: dom(memberEnds(aa1)) & p2: ran(memberEnds(aa1)) &
5          p1|->p2: opposite &
6  =>
7          #aa2.(aa2: dom(memberEnds)\/{assoName} &
8          (memberEnds\/{assoName|->{srcProp|->tgtProp}})(aa2) = {p2|->p1})
```

Table D.20: Proof goal and hypotheses after proof step 7

Using the hypothesis `not(aa1 = assoName)`, the set simplifier is able to simplify the hypothesis on variables `aa1`, `p1` and `p2`, this can be observed by comparing these variables in Table D.19 and Table D.20.

Step 8

```
ph(aa1,p1,p2,!(aa1,p1,p2).(aa1: dom(memberEnds) &
      p1: dom(memberEnds(aa1)) &
      p2: ran(memberEnds(aa1)) &
      p1|->p2: opposite
    =>
       #aa2.(aa2: dom(memberEnds) &
        memberEnds(aa2) = {p2|->p1}))) &
```

```
aa1: dom(memberEnds)     &
p1: dom(memberEnds(aa1)) &
p2: ran(memberEnds(aa1)) &
p1|->p2: opposite
```

Table D.21: Proof sub-goals after proof step 8

This command starts the instanciation of the hypothesis, using the values of `aa1`, `p1` and `p2`. It is necessary to prove that the provided values meet the requirements of the forall hypotheses,then, the hypotheses can be instantiated. As a result, the goal is changed into four subgoals, as can be seen in Table D.21.

Step 9 - ah(aa1: dom(memberEnds)assoName)

```
aa1: dom(memberEnds)\/{assoName} => aa1: dom(memberEnds)
```

Table D.22: Adding a hypothesis to the present proof goal

The prover did not succeed in discharging the first subgoal. This step adds the given hypothesis to the goal. The aim of adding this hypothesis to the goal is ensure that the prover has all the required hypotheses so that we are able to run the predicate prover only on this sub-goal (using the command pp(rp.0)).

Step 10 - pp(rp.0)

```
aa1: dom(memberEnds)
```

Table D.23: Current Proof sub-goal at proof step 10

Running the predicate prover, command `pp`, has discharged the previous sub-goal. Now, we need to discharge the remaining three sub-goals, to be able to complete the instantiation of the quantified hypotheses.

Step 11 - pr

```
 p1: dom(memberEnds(aa1)) &
 p2: ran(memberEnds(aa1)) &
 p1|->p2: opposite
```

Table D.24: Current Proof sub-goals at proof step 10

All subgoals required for instantiating the hypothesis has been discharged, and the instantiated hypotheses are added. Running the prover has moved the existential hypothesis to the global hypothesis stack, and generated the corresponding hypotheses for the variables.

Step 12 - se(aa2)

1	`aa2: dom(memberEnds)\/{assoName} & memberEnds(aa2) = {p2|->p1}`

Table D.25: Current Proof sub-goals at proof step 10

The `se` command allows proving the existential by providing a value meeting the required properties. Here, the value `aa2` (the one that has been generated in the previous hypothesis) is suggested. The instantiation of the variables under the scope of the existential quantifier is the last step to verify that this proof obligation can be discharged □