UNIVERSITY
of
GLASGOW

Computing Science
Ph.D. Thesis

On the Formal Specification and Derivation of Relational Database Applications

Roberto Souto Maior de Barros

Submitted for the degree of

Doctor of Philosophy

On the Formal Specification and Derivation of Relational Database Applications

by

Roberto Souto Maior de Barros

Submitted to the Department of Computing Science
on 25th November, 1994
for the degree of
Doctor of Philosophy

Abstract

The development of database applications is usually carried out informally. The derivation of database programs directly from formal specifications is a well known and unsolved problem. Most of the previous work in the area either tried to solve the problem too generally or was restricted to some trivial aspects, for example deriving the database structure and/or simple operations.

This thesis describes an extension to the traditional database design process aimed at formalizing the development of (relational) database applications. Specifically, it gives a complete description of a general method for the specification of relational database applications using Z, as well as a comprehensive description of a set of rules on how to derive database programs from specifications which result from using the method.

The method prescribes how to specify all the important aspects of relational database applications, which includes the definition of relations, the specification of constraints, and querying and updating of relations, including error handling. It also addresses more advanced features such as transactions, sorting of results, aggregate functions, etc.

However difficult in general, deriving relational database applications directly from Z specifications written according to the method is not arduous. With appropriate tool support, writing formal specifications according to the method and deriving the corresponding relational database programs can be straightforward. Moreover, it should produce code which is standardized and thus easier to understand and maintain.

An intrinsic part of the thesis is a prototype which was built to support the method. It provides a syntactic editor for the method and partially implements the mapping for a specific Relational Database Management System (RDBMS), namely the DBPL system.

Thesis Supervisor: Ray C. Welland
Title: Senior Lecturer in Computing Science

To
My wife Roberta,
my son Robertinho,
and my parents
Gelson and Carminha.

Acknowledgments

It is a pleasure to acknowledge the contributions of all the people who somehow helped me to complete this Ph.D.

Firstly, I would like to thank my supervisor, Ray Welland, for his support to my research and for his encouragement during the difficult times. In addition, I am specially indebted for his taking over the task of supervising my work when David Harper left the department for a new job in Aberdeen. Also, I appreciate his effort in convincing the examiners to keep the changes to a minimum. Finally, I thank him for being such a friendly and informal person.

I would also like to thank David Harper for accepting me as a research student here in Glasgow and for supervising me during the first two years. In particular, I am grateful for his warm welcome when I first arrived in Glasgow and for taking me as a friend.

I am also in debt to Kieran Clenaghan for bringing the Synthesizer Generator to my attention, as it turned out to be a great tool which tremendously enhanced the horizon of the practical results of my research.

Alex Gray, Kieran Clenaghan, and David Watt, as members of the examination committee, provided many interesting corrections and suggestions to improve this thesis.

My office mates Campbell Fraser and Alex Bunkenburg provided a most friendly and relaxed atmosphere in G161, making the last two years much more enjoyable than the previous two.

A special thanks to my Brazilian friend Hermano Moura for taking me as a guest in his home and for helping me with all sorts of things when I first came to Glasgow.

I also acknowledge the financial support that allowed me to carry out the research described in this thesis. This was provided by CAPES (Brazilian Federal Agency for Postgraduate Education) and by UFPE (Federal University of Pernambuco). In addition, I could not forget that both Ray Welland and David Harper provided funds which allowed me to attend several conferences and workshops throughout these four years.

Finally, my gratitude and love to my wife Roberta for giving up her job to come to Scotland with me and for all her support, patience, and love.

Roberto S. M. de Barros

Contents

List of Figures

Chapter 1

Introduction

This thesis is about the utilization of formal techniques for the development of relational database applications. In particular, this thesis argues that the formal specification and derivation of relational database programs can be made reasonably simple by the provision of appropriate methods and tool support.

1.1 Motivation

Having worked in the formal specification area for a number of years, my attention was mainly devoted to the application of formal methods in the development of real life software. In particular, my M.Sc. thesis [1] involved the formal specification of a large system, namely UFPE's Student Records Control System.

In addition, it is unlikely that a generic comprehensive solution to the problem of deriving real applications will be proposed in the near future. Hence, it was advisable to restrict the scope of the research to some well understood domain. The database area, and especially the relational database model, seemed to be the perfect target for the utilization of formal methods in this context.

Also, traditional database design processes [2] have typically put much more emphasis on the design of database structures than on the applications that will run against these structures. Because the design of database structures has received much more attention, it is now well understood and established. For instance, the application of formal and/or semi-formal techniques as well as tools during these phases is now common.

In this thesis, the specification of the database structure is done in Z [3, 4] and envisages the use of tool support.

However, the design of database transactions has hardly received any attention in the traditional database design process and is almost always very informal. Usually, it progresses from a very high level specification of transaction requirements directly to code. Thus, the effectiveness of this approach is very dependent on the programmers' experience and on the amount of testing done.

As a result, the requirements of transactions are frequently underspecified and the specifications are often inconsistent with the users requirements, mainly due to lack of precision. The implementations are, therefore, likely to be subject to error. A formal approach demands precision. Hence, it can force designers to consider details which might otherwise be overlooked. This should increase confidence in the correctness of the implementation.

Since formal methods have already been successfully applied to a number of areas, including the design of programming languages, hardware, etc., and in particular to the design of the structure of the database, it should be possible to apply formal methods to the design of database transactions with the same benefits.

So, the more general objective of this thesis is to formalize the design of database transactions (applications), especially for the relational model [5, 6], in a way that it can be used by practitioners in the development of real world applications. More specifically, this thesis proposes a new structure for the database design process, which extends the traditional approach with a number of phases specifically aimed at formalizing the development of (relational) database transactions.

1.2 Scope

A common problem regarding the application of formal methods to real problems is that beginners usually find writing formal specifications difficult. They need support in the form of primitives, methods, etc. to guide them. A critical first part of this work addressed this problem and involved the development of a method aimed at formalizing the design of relational database transactions.

In particular, the method provides a number of rules which prescribe how to specify all the important aspects of relational database applications using Z. These include the definition of relations, the specification of constraints, and querying and updating of relations, including error handling. More advanced features such as transactions, sorting of results, aggregate functions, etc. are also addressed.

Pre-defined operators[1] are used in most parts of the specification in order to make it simpler to write and understand. These operators capture specific aspects of the relational model, e.g. keys, nulls, etc., and some aspects of operations like delete and update.

In the main, the version of Z used in this thesis is the accepted standard [3]. Some extensions are introduced when necessary but they are avoided as much as possible.

It is worth emphasizing that the method is for the specification of relational database applications. So, the aim is *not* to specify either the Relational Model or the operators of the Relational Algebra (or Calculus). In addition, because it is intended to make the use of formal methods more available to practitioners, all aspects of the method need to be as simple as possible.

The other major problem investigated by this work is the derivation of database programs directly from formal specifications. Although some work has already been published, the utilization of formal or semi-formal techniques for the generation of real life database applications has not been seriously attempted yet.

A common drawback in some of the previous attempts has been to try to solve the problem too generally by not restricting it to applications based on a specific database model, or rather trying to refine a wide variety of application programs.

[1]The term 'operators' is used to refer to Z generic definitions throughout the thesis.

Another frequent mistake has been to overlook the vital need to specify constraints and to verify they are satisfied at all times, so that the consistency of the database is guaranteed at all times. This is normally done by only addressing the correct behaviour of simple atomic operations and usually leaves the false impression that deriving database applications is fairly straightforward.

On the other hand, experts on the database area tend to think the automatic derivation of real database applications is too difficult, especially because the programs must guarantee the constraints are satisfied.

The approach described in this thesis is restricted to the specification and reification of relational database applications. Furthermore, it also considers all relevant kinds of constraints as well as more complicated transactions.

Specifically, the thesis partially describes a generic mapping aimed at generating relational database programs directly from formal specifications written according to the method. The mapping addresses the problems involved in such a translation and is independent of any particular database system and/or language.

This thesis also involved a substantial piece of implementation work. Specifically, a prototype tool was developed. It aims to support the method and instantiate the mapping for a particular Relational Database Management System (RDBMS), namely the DBPL [7, 8, 9] system which was developed at the University of Hamburg.

The prototype is composed of a syntactic editor for the method and a built-in tool which translates the specifications to database commands. Since it is only a prototype, it does not cover the complete method. For instance, the syntactic editor accepts a large subset of all possible specifications which are correct according to the method, even though many of the incorrect ones are not rejected.

In addition, the implementation of the mapping for the generation of relational database applications to be run in the DBPL system is also partial. Nevertheless, the prototype produces appropriate pieces of code from a reasonably large subset of the operations, advanced features, and error handling schemas described in the method.

The prototype was developed using the Synthesizer Generator [10, 11], which is a powerful tool for implementing language-based editors and allows for the generation of syntactic editors fairly quickly. The implementation of the mapping was carried out using the view facility of the Synthesizer Generator.

1.3 Contributions

This thesis comprises three major pieces of work. In the first part, a general method for the specification of relational database applications using Z is provided. The primary contributions of the method are:

- It allows for an abstract specification of the applications to be developed, focusing on the important aspects of the relational model and applications, without regard to the fact that some features may not be supported by specific RDBMSs and query languages. It provides the formal basis in terms of which applications can be specified, verified (using formal reasoning), and implemented (reified).

- Using the method ought to help database designers and programmers in finding ambiguities and deficiencies in the requirements specification. Therefore, it should help practitioners in the development of real world applications. Furthermore, it should improve the system documentation and the quality of the application programs which should contain fewer errors. Notice that *database designers* and *programmers* constitute the *users* of the method.

- When implementing database systems without having previously specified them, programmers tend to concentrate only on the correct behaviour of the operations and overlook possible errors. The method also deals with the specification of the behaviour of the system when errors occur and prescribes how to get all possible errors. A summary of all possible errors for the more common operations might be added to the method in the future. In addition, the user may be discharged from proving a number of theorems about relational database applications because general theorems, with their proofs, ought to be added to the method in the future.

- Given that one of the difficulties of specifying a system formally is the choice of an appropriate level of abstraction, the use of a method should also lead the users to choose a suitable level of abstraction.

- The method allows for the standardization of the specifications. Thus, it provides a formal starting point for the investigation of the generation of relational database programs directly from formal specifications.

- Adopting the method also enables the utilization of modularization, reasoning, and refinement techniques. These might also be added to the method in the future and should contribute for improvements in the quality of the programs since it could lead to many errors being detected before the implementation (reification). Furthermore, these could reduce the costs of testing and maintenance.

- Ultimately, the method could be seen as the missing bridge to make the use of formal specification techniques more accessible to developers of real world software and, in particular, relational database applications.

The second part of the research described in this thesis investigates the derivation of relational database programs directly from formal specifications written according to the method and presents a simple mapping. The main contributions of the mapping are:

- It discusses the problems involved in the derivation of relational database programs directly from formal specifications without binding the investigation to any specific database system or language. In other words, the mapping is general and should be applicable to many RDBMSs.

- The investigation is restricted to applications based on the relational model, which means it does not try to refine too wide a variety of applications.

- The mapping is not restricted to the correct behaviour of simple atomic operations. On the contrary, it considers all the relevant kinds of constraints as well as more complicated transactions.

- In general, there is more than one way of writing correct database commands to implement particular operations. The utilization of the mapping allows for the standardization of the database operations contained in the application programs, which ought to lead to programs being easier to understand. As a consequence, the costs of testing and maintenance might be reduced.

Finally, the third part of this work concerns the prototype tool built to support the method and implement the mapping. The main reasons for building such a prototype were:

- To show that the specification of relational database applications using the method and the construction of the corresponding database programs can be reasonably straightforward if appropriate tool support is provided.

- To provide evidence that the syntax and semantics of the method are sound and that it is possible to build a full scale syntactic editor to support the method.

- To demonstrate that the mapping can be adapted to specific RDBMSs, that it is possible to derive database programs automatically, at least for a large number of applications, and that building a tool to implement the mapping for a particular RDBMS is not too difficult.

1.4 Organization

This thesis is divided into four parts. The first part, which comprises this and the following two chapters, introduces the work and puts it in context. The principal part of this work is then described in the next five chapters, i.e. Chapters 4 to 8. Next is Chapter 9, which closes the main body of the thesis. Finally, two appendices complement the presentation. The contents of the remaining chapters are summarized below.

Chapter 2 reviews the traditional database design process and proposes an extension aimed at formalizing the design of (relational) database transactions. In addition, it justifies the several decisions made in the directions of the research, explaining why the investigation was restricted to relational database applications, why Z was chosen for the specifications, why DBPL was chosen for the implementation of the prototype, etc.

In Chapter 3 the existing use of formal methods techniques for the specification and derivation of applications is surveyed. The scope of the survey is restricted to the formal specification of real, large scale applications using Z, and to the specification and derivation of database applications. The emphasis is specifically put on the derivation of relational database transactions from formal specifications. Some of the approaches are described in somewhat more detail and their strengths and weaknesses discussed.

Chapter 4 presents the latest description of the method. It starts by presenting rules for the specification of the database structure, i.e. domains, relations and their attributes, candidate and foreign keys, as well as other constraints to be guaranteed. The specification of basic operations over the database are covered next, which includes operations such as select, project, join, insert, delete, and update. These are followed by the specification of more advanced features, which includes transactions, sorting of results, aggregate functions, composite attributes, and views. Then, it addresses the extension of the applications (transactions) to capture error handling, using two different approaches. The chapter is concluded with the presentation of a number of guidelines on how to use the method realistically.

Chapter 5 presents the formal definition (specification) of the operators used in the specifications written according to the method. These pre-defined operators, informally introduced within the description of the method, capture specific aspects of the relational model, such as nulls and candidate and foreign keys, as well as some specific aspects of update and delete operations. Other operators are provided to simplify the use of some advanced features such as sorting of results, aggregate functions, and composite attributes.

In Chapter 6, a complete example is specified using the method. It starts with an informal description of the chosen transactions. The database structure affected by the transactions is then captured by an Entity-Relationship (ER) diagram. Next, natural language descriptions of the database constraints that must be guaranteed are presented. After that, the relational database structure is formally specified. The specification of common basic operations follow. Finally, the chosen transactions are specified and this includes the calculation of the preconditions and error handling.

Chapter 7 describes the mapping aimed at the derivation of database programs from specifications written according to the method. The exposition of the mapping follows basically the same order used in the presentation of the method in order to make its understanding easier. Thus, it starts with the rules for the mapping of the database structures and constraints, which are followed by the mapping of the database operations, the advanced features, and the extensions to capture error handling, respectively.

Chapter 8 describes the prototype system which was built to support the method and implement the mapping. It provides some details about the problems of adapting the mapping for a particular RDBMS. It also discusses the strategy used to build the prototype as well as a number of design decisions that have been made regarding the functionality and implementation of the prototype. In addition, the chapter presents a quick introduction to the Synthesizer Generator, the tool used to build the prototype.

In Chapter 9 the overall conclusions reached by the research are presented. It starts with a summary of the work done and devotes special attention to the benefits of the method, the mapping, and the prototype. The chapter is closed with suggestions for future extensions and further work.

Finally, the appendices are presented. The first one exhibits the simplification of the precondition of a transaction involving many subtransactions and potentially affecting many of the specified constraints. The second appendix presents selected parts of the Synthesizer Specification Language (SSL) code written to build the prototype.

Chapter 2

Overview

This chapter provides an overview of the work and its context.

It starts with a review of the traditional database design process and proposes an extension aimed at formalizing the design of (relational) database transactions.

In addition, this chapter justifies the several decisions made in the directions of the research, explaining why the investigation was restricted to relational database applications, why Z was chosen for the specifications, and why DBPL was chosen for the implementation of the prototype. This chapter also provides a concise introduction to formal methods and formal specifications as well as a classification of formal methods.

2.1 Database design

Traditional database design processes [2] have typically put much more emphasis on the design of database structures than on the design of the transactions that will run against these structures. In fact, the design of the structure of the database is usually seen as a prerequisite for the development of the applications that will run against it.

In addition, database structures are much more static than the applications and, in many cases, much more difficult to modify. Specifically, changing the structure of a non-relational database invariably means that a number of applications must be changed too. However, this need not be true in the case of relational databases.

For instance, adding a new attribute to one of the relations in a relational database application does *not* mean that all application programs reading the changed relation need to be changed. On the contrary, in most existing relational systems only the programs that manipulate the new attribute need to be changed. On non-relational platforms, all programs that access the changed relation usually have to be changed. Even though the actual changes are frequently limited to updating the record structure associated with the changed entity (file), it usually involves changing and recompiling a considerable number of programs.

Because of these, database designers usually make their best effort to achieve a consistent database structure before the development of the applications is begun. For instance, formal (or semi-formal) techniques as well as tools are usually applied to the design of database structures, as opposed to the development of database applications which is almost always very informal.

A legitimate question that might be asked is then: "Why does only the database structure receive the appropriate attention?". Also, "Why do database applications not receive the same attention?". Apart from the fact that the applications are much more dynamic than the structures and, therefore, need to be changed more frequently, a couple of other reasons contribute to this state of affairs.

To start with, many database designers and programmers underestimate the cost and difficulty of maintaining the applications and regard this activity as a simple one because modifying the structures is more difficult. In fact, maintaining programs directly on the code without updating the corresponding documentation (specifications) is a quite common practice among professional programmers. These are usually sceptical about the importance of the systems documentation.

Also, users of computer systems (either database or non-database systems) are used to low standards in software development and, generally, are likely to accept errors as normal or even inevitable. Fixing errors quickly is usually enough to keep end users satisfied.

In this section the traditional database design process is reviewed and an extension aimed at formalizing the design of (relational) database transactions is proposed.

Because the design of database structures has received much more attention, it is now well understood and established. For instance, the use of formal (or semi-formal) techniques as well as tools during these phases are now very common. Therefore, there is no intention to propose any major changes or contribution in these phases of the database design process.

2.1.1 Traditional database design

Figure 2.1 summarizes the traditional database design process. Because the design of database transactions has hardly received any attention, almost all phases in the process refer to the design of database structures. For completeness, a brief description of each phase is added.

Requirements

This phase refers to the specification of requirements that all potential users of the database may have. Users must be repeatedly interviewed because, in general, there is *no* guarantee that the specifications will meet the user requirements.

The inputs are informal statements written in natural language, produced from the interviews with the users.

The outputs are usually separated into two groups: *data requirements* and *processing requirements*. The first of them, data requirements, refer to which data is needed in the database.

Processing requirements refer to how data is to be processed. These usually include the specification of the inputs and outputs, functionality, frequency of execution, and desired performance of transactions that are to be run against the database. Even though a number of transactions is not normally known at this time, i.e., before the implementation, the more important ones are usually known in advance.

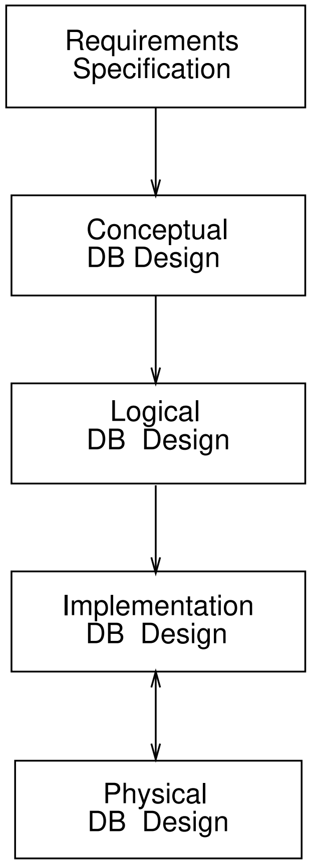


Figure 2.1: Traditional Database Design Process

The outputs of this phase are frequently written using a requirements specification language, e.g. SSADM [12], SADT [13, 14], SAMM [15], HIPO [16], DFDs [17], etc. Restricted versions of natural language are sometimes used as well, especially for the specification of processing requirements.

Although (semi-)formal techniques are usually applied, it is worth repeating that users need to be interviewed repeatedly because, in general, there is *no* guarantee that the specifications will meet the users requirements. For this reason, this phase is really very important and can be quite time consuming. Even though processing requirements are also collected during this phase, the emphasis is usually much more devoted to the data requirements.

Conceptual DB design

In this phase, a data model independent database schema (the conceptual schema) is specified using a very high level data model, e.g. ER [18], EER [19], RM/T [20], etc. It usually involves the specification and integration (merging) of the users' views of the database.

The inputs for this phase are the data requirements of the previous phase. The output is the database conceptual schema.

Logical DB design

During this phase the conceptual schema and external views are translated to the generic data model of the target DBMS. In the specific case of the relational model, the logical database design also includes the normalization of relations [5, pp. 525–560]. In addition, any limitations the target DBMS may impose on the data model are usually *not* taken into account.

The inputs are the conceptual schema (last phase) and the structure and limitations imposed by the chosen data model. The output is the database logical schema. In the case of the relational model, the result is typically the database structure written in an SQL-like [21] language.

Formal generic mappings from the conceptual schema based on the ER model to the relational, hierarchic, and network models are already established [22, pp. 309–409]. Similar mappings from conceptual design specifications written in a formal specification language to corresponding data model dependent formal specifications should be straightforward. Moreover, it should be possible to refine the high level specification of transactions into corresponding data model dependent (but still abstract) specifications.

Finally, it seems reasonable to believe that mapping a well designed ER schema to the relational model should result in relations already normalized. However, this is only possible if all types of dependencies, i.e. functional, multivalued, and join dependencies, are represented in the ER diagram. Anyway, automated tools for normalizing relations are already available [23, 24].

Implementation DB design

After an specific DBMS is chosen to be used in the actual implementation, the structure of the database is implemented using its *Data Description Language* (DDL). Sometimes, because a number of DBMSs include physical parameters in their DDLs, this phase is carried out in parallel with the physical database design (next phase). This is usually not the case of relational DBMSs though.

The input to this phase is the logical database schema. Accordingly, the output is the implemented database structure.

Physical DB design

Finally, appropriate storage structures and access paths for each of the elements of the database are defined in order to achieve good performance. The application programs are usually run to monitor the required performance of the more important transactions thus helping in this phase.

The inputs are the implemented database structure and the constraints, frequencies of execution, and desired performances of transactions. The outputs are the storage structures and access paths.

Optional phases

In the specific case of very large databases, extra phases are sometimes considered in such a process. This usually includes:

Distribution: If the database is not to be managed centrally, than it is necessary to decide which data is to be stored at what location, which is usually based on the results of the transaction design. It may also be the case that different DBMSs are to be used in different sites.

Bench Marking: This refers to the generation of test data and prototype applications in order to aid in measuring the performance of the database before the real data is loaded. Also, the applications used in these tests may be a subset of the applications and/or partial implementations of some of the operations (transactions).

2.1.2 Enhancing the database design process

It has already been mentioned that transactions have hardly received any attention in the traditional database design process. As a result, the design of database transactions is almost always very informal and usually progresses from a very high level specification of transaction requirements directly to code. Thus, the effectiveness of this approach is very dependent on the programmers' experience and on the amount of testing done.

It is believed there is no good reason for this state of affairs. So, the design of database transactions should also be formalized. Moreover, it should be possible to formalize it in much the same levels of abstraction in which the structures were formalized.

This thesis proposes a new structure for the database design process which extends the traditional approach with a number of phases specifically aimed at formalizing the development of (relational) database transactions. Figure 2.2 summarizes the proposed structure. Thicker boxes and lines refer to the proposed (or modified) phases and their corresponding inputs and outputs respectively. A brief description of each proposed or modified phase is again provided.

Conceptual specification of transactions

This refers to the formal specification of database transactions at a very high level of abstraction. The specifications should be DBMS independent and possibly data model independent as well. There is no specific method (or language) established yet. Even though there is a need for such a method, identifying what this might be is not the subject of this research.

In spite of that, it is reckoned that for each transaction such a method would have to specify at least (1) What the inputs and outputs are, (2) what the entities and relationships changed are, (3) for each changed entity or relationship, which attributes are changed, and (4) for each changed attribute, what the changes are.

The inputs are the processing requirements whereas the outputs are the conceptual specifications of the transactions.

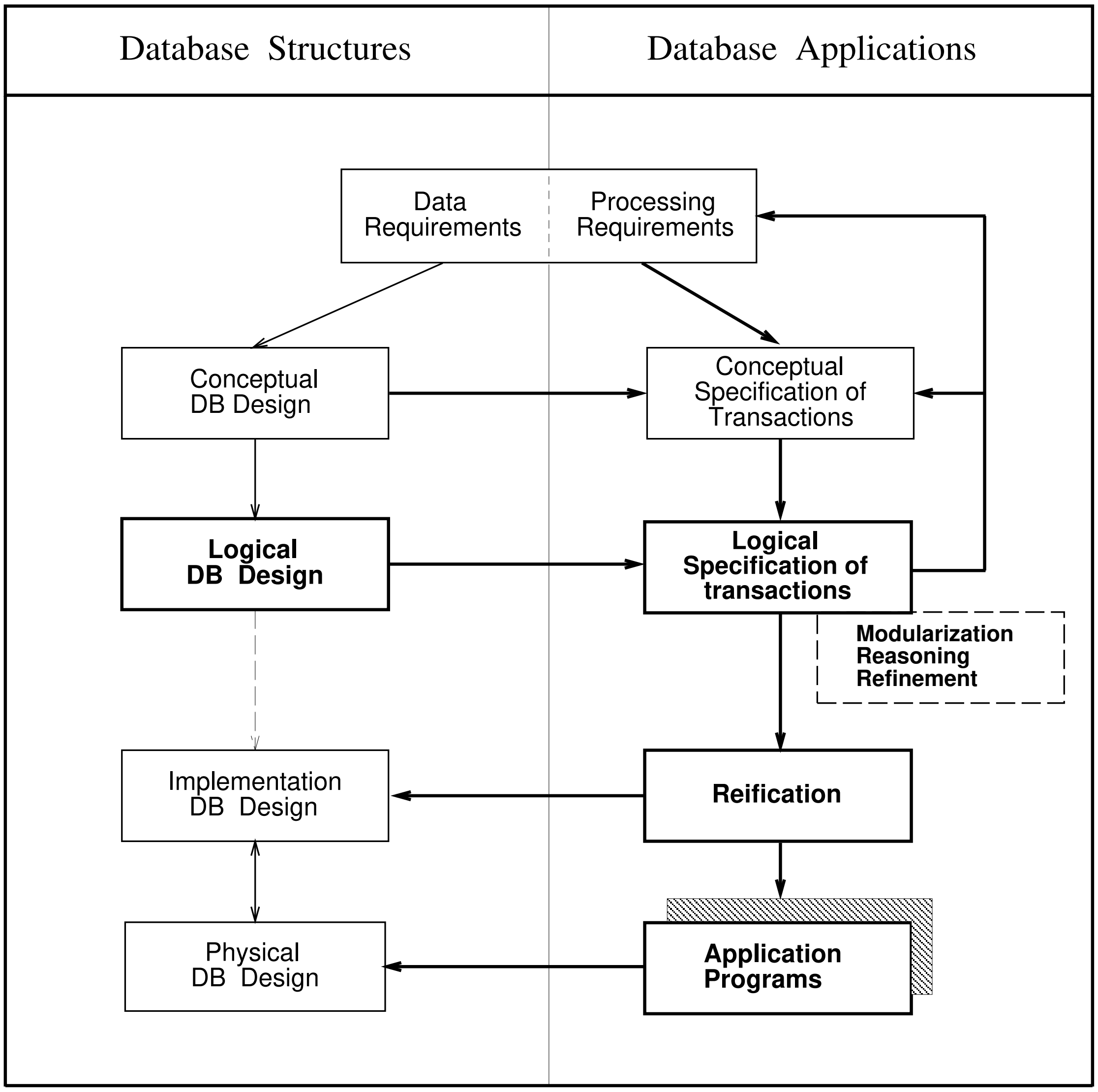


Figure 2.2: Proposed Database Design Process

Logical DB design

As already mentioned, there is no intention to do any contribution to the design of database structures. The main interest here is to be able to specify the transactions that are to be run against the database. Even so, a small change in this phase is proposed. It is aimed at making the logical specification of transactions (next phase) slightly easier.

Basically, instead of using an SQL-like language, the logical database schema is expected to be generated in Z [3, 4], the formal specification language used to specify the transactions at the logical level.

Modifying available tools for logical database design to generate a Z version of the logical database schema should not be too difficult. While such modified tools are not yet available, it might be that generating the Z version of the logical database schema based on a simple translation from the traditional SQL-like specifications would be the best approach.

A number of reasons have been taken into account in the decision of having a Z version of the database structure. Firstly, in a mixed mode, SQL-like specifications of the database structure would not mix well with the Z specifications of transactions. Actually, the Z specifications would be incomplete if the declaration parts were omitted. Secondly, it would be more difficult to reason about such specifications. Finally, this would not be appropriate if the database were not to be implemented in an SQL system.

To tackle the first two problems, the syntax and semantics of what would be a new specification language would have to be provided. In my opinion, this is not necessary because restricting an appropriate existing language by means of a method seems more adequate and simpler.

Logical specification of transactions

During this phase, a data model dependent, but DBMS independent, formal specification of database transactions would be written using a formal specification language. More specifically, a general method for the specification of relational database transactions has been developed. It prescribes how to specify all the important aspects of such transactions using Z.

The use of modularization and reasoning techniques during this phase might make the understanding of the specifications and the proof of desired properties easier.

In addition, these should make more effective the process of identifying errors and ambiguities in the specifications, as well as inconsistencies between the requirements and the specifications, before the implementation (reification) is carried out.

As a consequence, revisions in the previous phases should follow and, therefore, the design (requirements) of database transactions ought to receive at least as much attention as the design of the database structure. As a result, the quality of the programs should be improved and the costs of testing and maintenance reduced.

The inputs are the logical database schema (Z version) and the conceptual specification of transactions. The outputs are the logical specifications of transactions.

Reification of transactions

Application programs that implement the specification of database transactions could be semi-automatically generated in this phase. The approach is to investigate all problems that might arise in such a process, using a specific RDBMS and query/host language (or 4GL) as example. This process should also lead to changes (improvements) in the method used for the logical specification of transactions.

The inputs are the logical specification of transactions from the previous phase, information about which features of the relational model are supported by the chosen RDBMS, and which query/host language or 4GL is to be used. The outputs are the application programs.

2.2 Motivation for using relational databases

The research described in this thesis was restricted to the relational model. This model seemed particularly useful for my purposes for the following reasons:

- The specification method developed is reasonably simple and does not enforce any constraints on either the real implementation of relations or the choice of a specific database system and language.
- The proof of properties about such specifications, though not investigated in detail, seems to be fairly easy, involving only first order logic and set theory.
- The very high-level nature of its query languages means it seems likely that the refinement step is not necessary in this case. Moreover, the reification process did not seem to be arduous.

Basically, what distinguishes relational database applications from other applications is that the former are designed in terms of a much higher-level data model. In the relational model, all files are relations (in the mathematical sense) which limits the possible operations that may be executed against the database and allow them to refer to sets of tuples (records) instead of a single record at a time.

Another very important aspect concerning the relational model is that it permits what is called *data independence*, i.e., application programs that use the database are not dependent on the physical structure of data. So, regardless of how the chosen DBMS implements relations and what structures may be used to improve the performance of the applications, the programs are not changed because they directly manipulate "logical" relations.

In addition, database applications (not only relational) are developed on top of a DBMS which controls the database. The DBMS makes database application programs simpler by doing many controls that, in non-database environments, have to be done by the programs.

Finally, it is important to notice that the formal specification of database applications does not need to be much different from those of other applications. On the contrary, specifications written according to the method can possibly be used to specify non-DB applications, perhaps with slight modifications.

2.3 Formal methods and Z

This section presents a concise introduction to formal methods and formal specifications, as well as a classification of formal methods. More extensive introductions have been published [25, 26]. In addition, this section summarizes the reasons why Z was chosen to be the formal specification language adopted in this work.

2.3.1 What are formal methods?

The term *Formal Methods* has been used to mean a number of different activities in the development of software systems. These include the formal specification of the intended functionality of the systems; the use of formal reasoning to prove properties of the systems, possibly before the implementation is developed; the derivation of a correct implementation in the sense that it is guaranteed to preserve all the properties of the specifications; etc.

However, all these interpretations share a common aspect this being they all imply the use of mathematical (formal) notations. Frequently, these notations are (or should be) complemented with techniques and/or guidelines aimed at permitting a more systematic application of the mathematical notations and/or making their use simpler.

Usually, the use of formal methods includes at least the formal specification of the intended software. Formal specifications are specifications written in formal specification languages, i.e. languages which have well defined and precise syntax and semantics. The need for a formal semantics implies that the meaning of specifications expressed in the language are not ambiguous.

It is generally accepted that the utilization of formal techniques in the development of real application systems provides some useful benefits, as it helps to avoid ambiguity or vagueness and, thus, to provide a better interface for precise communication of ideas between the designer (specifier) and the programmer, as well as between the designer and the end users. Also, it ought to help to reduce maintenance costs, since more of the errors in a system should be discovered before it is implemented, and to detect and correct errors and anomalies in the documentation of such systems.

Hall [27] has pointed out that *"From an economic point of view, the most important part of a formal development is the system specification"*, and also that formal methods *"work largely by making you think very hard about the system you propose to build"*. In a sense, by simply writing formal specifications the users are forcing themselves to be more rigorous.

2.3.2 Classifying formal methods

Formal methods are usually classified according to the semantic foundation of the specification languages they use. The two main approaches are known as the *model-oriented approach* and the *property-oriented approach*. Duce and Fielding [28] provide a detailed comparison of the two.

In the model-oriented approach, the specification and the design are explicit abstract models of the system to be developed. The specification language provides well defined primitives which permit the construction of a mathematical model in terms of abstract data structures such as sets, relations, functions, etc.

The more established model-oriented formal specification methods are Z [3, 4] and VDM [29, 30]. Others include RAISE [31], HOL [32], CSP [33], and CCS [34].

On the other hand, in the property-oriented approach the specifications describe the behaviour of the system in terms of the constraints that must be satisfied, without the design of any specific models. The specifications are axioms which define the relations among the operations, and the properties are the result of the logic manipulation of the axioms. Examples of property-oriented formal methods include OBJ [35], Larch [36], Clear [37], and ANNA [38].

2.3.3 Motivation for using Z

Most of the work published on the formal specification of databases uses algebraic [39] (property-oriented) specification languages [40]. Even so, it seems that model-oriented specification languages are more appropriate to specify database transactions, especially because of the convenient notion of *state*. Moreover, it is possible to write property-like specifications using a model-oriented language like Z, if desired.

After deciding for model-oriented languages, it is necessary to decide which of the two more established ones (Z and VDM) is more appropriate.

In general, Z and VDM are languages based on first order logic and set theory, and allow for very abstract specifications. Modularization techniques for improving the understanding of large specifications were also proposed for both [41, 42]. In particular, the Document/Chapters extension to modularize Z specifications [41] also allows the specification of abstract data types using the same style adopted in property-oriented languages.

On the one hand, the schema calculus of Z allows for the incremental presentation of specifications by including other schemas and/or linking schemas with propositional connectives. Also, its notation seems slightly better to write and understand than that of VDM for it uses the standard mathematical symbols as much as possible and encourages the use of informal prose merged with the formal text.

On the other hand, VDM has a better structure for the transformation of the specifications into implementations because it is necessary to write the preconditions and postconditions of the specifications explicitly. Also, its proof obligations together with explicit preconditions and postconditions make reasoning about specifications more straightforward than in Z.

Even though they have many points in common, Hoare [43] has suggested that Z and VDM should be used for different purposes. According to him, Z would be more suitable when the aim is the specification of the systems only. On the other hand, VDM would be more suitable when the aim is the implementation.

The author is not convinced about such a suggestion and believes both can be used interchangeably without much problem. Also, I notice that, in many cases, the choice for one or the other is merely a matter of convenience, e.g. the existence of people already trained in one of the formalisms but not in the other.

The main differences between Z and VDM are discussed by Hayes, Jones, and Nicholls [44].

The formal specification language chosen to be used in the research described in this thesis is Z, for the following reasons:

- It is model-oriented. As already mentioned, model-oriented specification languages seem to be more appropriate to specify database transactions, especially because of the convenient notion of *state*. Moreover, it has been claimed that, in general, human beings tend to find model-oriented methods easier to understand than their property-oriented counterparts [45, 46, 47].

- It is an established language which has been under development for over a decade and is currently being standardized. An extensive literature is also available and includes a user manual [3], a number of introductory textbooks [4, 48, 49], a book on its semantics [50], a collection of case studies [51], and a book aimed at helping people who understand the basics of Z to become Z users [52]. In addition, Z is probably the most widely used formal specification language and has been adopted in many projects both in academia an in industry [27].

- Regarding the level of abstraction of the specifications, Z is a very flexible language and permits the adoption of different levels of abstraction, even within the same specification document. This gives the specifier the necessary freedom to adopt the most appropriate level of abstraction for each part of the specification.

- My previous experience of using Z and a Z-like language [1, 53] meant I had a great deal of confidence that Z could be used to specify database applications and, in particular, relational applications with good results. Moreover, choosing Z also meant there would be no need to spend time on learning another language.

- The existence of a large users group which promotes annual workshop meetings.

2.4 The method and specific database aspects

Real database applications involve many specific aspects which are usually not considered in the development of more traditional file-based applications. This section lists some of these aspects and explains how they relate to the research described in this thesis. More specifically, it explains how specifications written according to the method deal with such aspects.

2.4.1 Transactions (recovery and concurrency)

The method provides for the specification of transactions, i.e. a group of operations that are to be executed as a unit.

Should any of the components of a transaction fail, the transaction must fail and the database must remain unchanged (recover). Specifications written according to the method capture this behaviour. Most RDBMSs allow for the definition of transactions, but the way they are implemented depends on the RDBMS chosen.

Regarding aspects of concurrency, in general these should not be specified as part of the application programs. This is also a DBMS task and depends on other applications as well. Thus, the method does not address such aspects.

Usually, only one application is allowed to write in a specific relation (or tuple) at a time, although many can read it. In most systems, the DBMS automatically controls this in order to guarantee the integrity of the database, although in some systems the Database Administrator (DBA) has to specify what relations should be locked by some applications.

2.4.2 Security

Security may be described as the protection of data against unauthorized users. One method of restricting access to parts of the database is to use views. The method provides for the specification of views, i.e. horizontal/vertical subsets of relations, join subsets of data, and update restrictions on attributes.

Identifying individual users/groups and relating specific users to views are beyond the scope of the method. This is normally not specified as part of the applications either.

The method does not address the updatability of views either as this is, in its own right, a whole area of research [54]. In general, it is not even possible to decide whether some views are updatable or not [55]. Moreover, it is not always clear what the semantics of updates of specific views should be.

2.4.3 Integrity

This refers to the accuracy/validity of data. Integrity constraints are usually expressed as conditions that should be true at the start and end of a transaction and, possibly, compensating actions for when the constraints are violated. These are both covered by the specification method.

Since no system currently provides adequate integrity support [5, pp. 429], a number of implementation alternatives for each type of constraint are discussed in Chapter 7, the mapping of applications from specifications written according to the method.

2.4.4 Normalization

The method proposed in this thesis only requires the relations to be in first normal form (1NF), though they are generally expected to be in Boyce/Codd Normal Form (BCNF).

2.4.5 Performance

In relational database systems, the physical design of the database structure is usually totally independent of the applications. The utilization of structures/techniques such as indexes, clustering, hashing, etc. to guarantee good performance for one or more applications does not mean the applications have to be changed.

Although physical design is a very important task, it is worth emphasizing that, in relational database systems, the application programs are independent of the physical structure. For this reason, the method does not address such aspects either.

2.4.6 Distribution

This refers to databases not managed centrally. The method does not consider this aspect because application programs should be independent of distribution strategies. A database could even be distributed after its implementation and, even so, application programs should not need to be changed.

2.5 Motivation for using DBPL

As already mentioned, the relational database system chosen to be the target system in the construction of the prototype tool is the DBPL system, which is an academic tool developed at the University of Hamburg, Germany.

The DBPL system extends the programming language Modula-2 [56, 57] with a new persistent data type called *relation* and high-level relational expressions based on the predicate calculus.

The main reasons for adopting DBPL were:

- The new type relation and the corresponding access expressions are well integrated with the Modula-2 language to form the database programming language DBPL. As a consequence, it avoids the impedance mismatch which is common in the case of query languages such as SQL [21] being embedded in programming languages such as C or COBOL.

- The DBPL system implements a bigger subset of the theoretical relational model than most systems currently available. For this reason, in my assessment, from the systems available in the University of Glasgow it was the most well-suited to my purposes.

- Finally, because DBPL is an academic tool, it would be much easier to contact the developers and ask questions, which increased the chances of using the full capabilities of the system without spending too much time reading extensive manuals.

2.6 Conclusion

This chapter provided an overview of the research described in this thesis and put it in context. In addition, the chapter provided justifications for the several design decisions that have been necessary throughout the Ph.D. work.

The following chapter then surveys the existing literature on the utilization of formal methods techniques for the specification and derivation of applications and, in particular, relational database transactions.

Chapter 3

Literature Survey

This chapter presents a literature survey of the existing use of formal methods techniques for the specification and derivation of applications.

The scope of this survey is restricted to the formal specification of real, large-scale applications using Z, and to the specification and derivation of database applications. The emphasis is specifically put on the derivation of relational database transactions from formal specifications. Some of the approaches are described in somewhat more detail and their strengths and weaknesses are discussed.

3.1 The specification of applications using Z

This section covers the use of Z for the formal specification phase in the development of real applications. Unfortunately, the application of formal methods techniques in industry is still rather limited. Pointers to papers which discuss why this is so are also provided at the end of the section.

The most well-known and, probably, the largest and longest running industrial project to use Z is a joint project between IBM (UK) Laboratories at Hursley and the Programming Research Group at Oxford University Computing Laboratory which started in 1981 and is referred to as the CICS project.

The CICS project included the specification of several parts of IBM's transaction processing system CICS. Summaries of how Z was used in the restructure of IBM CICS are presented in [58, 59, 60]. Some of the CICS subsystems already specified in the project are: the CICS Application Programming Interface [61, 62], The CICS exception handling [63], the CICS temporary storage [64], and the CICS message system [65].

Other reported real projects using Z include:

- The development of a new computer control system for a real medical device, namely a cyclotron based clinical neutron therapy system, which is used for cancer treatments at the University of Washington, Seattle. The functionality of the system has been specified using a framework for the formal specification of safety-critical control systems in Z. The framework and an example specification are described in a paper by Jacky [66].

- The development of a formal security policy model for the NATO Air Command and Control System (ACCS). The project included the use of Z for the formal specification of the system together with informal validation of an appropriate subset of the specifications based on more traditional methods. An industrial report by Boswell [67] summarizes the results.

- The specification of British Rail's signalling rules as part of a requirements specification document for a railway interlocking system. The specification was written in Z by a small team from Praxis Systems for British Rail's Network SouthEast (now Railtrack) and their experience is reported by King [68].

- The development of a transaction processing mechanism for a relational DBMS called SWORD [69]. The mechanism is for controlling multi-transactions access to the database without any explicit locking of data. The project is reported by Smith and Keighley [70] and included the Z specification of the mechanism.

Craigen et al. [71] summarize an extensive survey and analysis of the use of formal methods in the development of twelve industrial applications [72]. In addition, the authors discuss the methods and styles of industrial usage in these applications and provide a number of recommendations aimed at making formal methods more palatable to people from industry. Some of those applications have used Z.

Another extensive survey is presented by Austin and Parkin [73]. It comprises a literature survey and the analysis of questionnaires returned by 126 organizations, mainly in the UK.

A very good paper by Hall [27] presents a comprehensive overview of the so-called myths which help to prevent a wider acceptance of formal methods in industry, and disputes them all one by one, refuting most of them. One of these myths refers to formal methods not being used on real large-scale software, which the author refutes by listing a few references. A recent paper by Bowen and Hinchey [74] revisited the subject and discussed another set of those myths.

The problem of marketing formal methods in order to achieve a wider acceptance in industry is discussed by Weber-Wulff [75]. The author discusses a number of problems affecting formal methods, *"from the point of view of the industrial programmer"*, and presents simple suggestions aimed at helping to convince people from industry to invest time and money in learning and applying formal methods.

3.2 The derivation of applications

This section briefly examines the problem of deriving implementation programs from formal specifications without restricting the scope to the database field.

The formal derivation of programs from specifications written in a formal language is usually called *refinement*. This process is often seen as comprising two distinct phases called *data refinement* and *operation refinement* respectively. In short, data refinement refers to the refinement of the data structures whereas operation refinement refers to the refinement of the operations that manipulate the structures.

In the most common approaches, refinement is defined as the application of formal techniques to map (refine) a given formal specification into another specification which satisfies the former but is more concrete in the sense that it is closer to the target programming language. This process is then successively applied and stops when all the features of the specification language are substituted by equivalent constructions of the implementation programming language.

In each of these steps, a number of proof obligations must be discharged. These are essentially the proof that the more concrete specifications indeed satisfy all the properties of the more abstract ones.

There are several similar approaches to this general idea of refinement. Some of the most well known were proposed by Morris [76], by Morgan [77, 78], and by Back [79]. These are all based on extensions to Dijkstra's guarded command language [80].

This theory of refinement has very much been the subject of continuing research for several years. There are whole books written or being written about refinement as well as many research papers published in conference proceedings and research journals. There is even an annual workshop totally dedicated to refinement, with the proceedings being published by Springer-Verlag in the *Workshops in Computing* series.

Nevertheless, many of the central ideas have been around for a long time and, as far as I can see, there is still a long way to go before real, large-scale, generic software can be automatically generated by refinement. Furthermore, I am not convinced that the refinement of programs based on an unlimited platform will ever be feasible.

On the other hand, it is possible that, in the future, the programming languages will efficiently support all the abstract data structures provided by the formal specification languages and will provide much more expressive means of manipulating these structures.

Until this time comes, the derivation of generic programs will probably be limited to the generation of prototypes to be run in systems supporting more advanced features, albeit not being implemented efficiently.

An example of this approach is an experiment described by O'Neill [81, 82]. Specifically, he used the view facility of the synthesizer generator to extend an existing syntactic editor for VDM-SL with a translator which automatically generates Standard ML [83] code from the VDM specifications.

3.3 The formal specification of database applications

This sections surveys the utilization of formal notations in the specification of database systems, languages, and applications. In my opinion, most of the work done in the area can be described as specification exercises rather than work aimed at making contribution to the database field. These are the approaches examined in this section.

More specifically, several people have specified database models (e.g. the relational model), specific database systems, and database operations (e.g. the operators of the relational algebra [84]). Others have addressed the specification of the correct behaviour of database transactions. However, only a few have covered at least an extensive subset of all features needed in the specification of real database applications so far. In these approaches, the resulting specifications are usually used as input for the derivation of database programs that implement the transactions using a specific DBMS. Hence, the material on these approaches is postponed to Section 3.4.

Samson and Wakelin [40] present a comprehensive survey about the use of algebraic methods to specify databases. They compare quite a number of approaches according to the features covered and enumerate some not normally covered by such methods. According to them, the *relational model* per se and the *relational algebra* are not normally formally specified, although "few ideas can be more familiar to the database community than the operators of the relational algebra". However, if the aim is the specification of applications, it is not absolutely necessary to formally specify either the relational model or the relational algebra. They also claim *domains* and *reification*, are not adequately addressed and, in most cases, not addressed at all. They also criticize the solutions for the specification of *state*, but this is applicable to algebraic specifications only.

In [85] Wong and Samsom present the specification of a relational database called PRECI, which is based in abstract data types. According to them, one of the strengths of their work is the fact that their specifications may serve as a prototype, for they present a partial implementation written in HOPE [86]. They also claim "the HOPE implementation provides an ideal vehicle for the investigation of new attribute types (domains)", but they address neither *how* this investigation works nor *why* it is ideal.

Another rather different experiment using a specific DBMS is presented by Fitzgerald and Jones [87]. They use the VDM specification language, referred to as Meta IV in the paper, to modularize the specification of a specific database system called NDB [88]. However, their emphasis is on the modularization techniques used to separate the VDM specifications into modules. The description of the DBMS is merely the chosen example of a realistic specification task.

The approach adopted by Turner and Lowden [89] is to use formal semantics as a means of specifying relational query languages. The authors describe a formal semantic framework for specifying database query languages and use it to specify the semantics of older versions of SQL and QUEL [90] and of a variant of the relational calculus [91]. Again, their aim is not to specify database applications but database query languages.

An interesting though unpublished exercise is described by Sufrin and Hughes [92]. They use an old version of Z to give specifications of the operators of the relational algebra. However, there are some problems. Firstly, the definition of relations depends on a set of all possible names of attributes of relations, because they define relations as a collection of functions from the relation to the attributes. Secondly, they do not cover important aspects of the relational model, such as primary and foreign keys. Finally, joins are not specified conveniently, being based on all attributes with common names. To specify more general joins, it is necessary to define a number of extra functions to rename attributes.

One reasonably common characteristic in several approaches to the specification of databases is that the author(s) usually do not worry about whether the specifications are "easy" to write and understand and, consequently, whether they are going to be used in the development of real databases or not. Samsom and Wakelin [40] even (using their own words) "dare to say" some authors choose the database application area to display their "mathematical virtuosity" and, in most cases, "the results are not of interest to the database community". Some of the work published that fits in this category is commented below.

- Khosla, Maibaum, and Sadler [93] use modal logic to specify database operations. They strongly criticize the use of abstract data types and of the algebraic approach to such specifications, particularly because of the absence of the notion of state. Nevertheless, they only specify a couple of operations and their specifications seem even more unnatural than some of the algebraic approaches. However, there is one of their ideas that seems useful, at least in some cases: they defend the specification of some constraints by default, i.e., providing operations that never put the database in an inconsistent state. For example, provide an "increase salary" operation instead of "change salary", if salary cannot be decreased.

- Fiadeiro and Sernadas [94] develop a rather different approach using temporal logic. They claim their approach covers important aspects such as the specification of operations, transactions, and errors, and also deals with proofs. However, their approach would be more appropriate to an abstract transaction design during the conceptual design rather than to the applications design, because they intentionally do not concentrate on any specific data model. Also, they specify only a couple of operations and their specifications are quite hard to understand.

- Abiteboul and Vianu [95] propose an operational approach to the specification of relational databases. They use transactions to describe valid database states and present a number of proofs about decidability. However, their aim seems to be to provide a framework to decide whether transactions are applicable rather than the development of applications. Also, the specifications and proofs presented are quite hard to follow, even though the theory seems sound.

A number of other approaches can be found in the proceedings of the *International Workshop on Specifications of Database Systems* [96].

3.4 The derivation of database transactions

This section surveys the derivation of database transactions in general and of relational database transactions in particular.

The approaches most related to the work described in this thesis are the work of Xiaolei Qian and, especially, the work done by the database group at the University of Hamburg, Germany, which is referred to as the Hamburg work. These two approaches are discussed in more details than the others in Subsections 3.4.4 and 3.4.5.

3.4.1 The work of Pastor and Olivé

A recent conference paper by Pastor and Olivé [97] proposes a method for the generation of transaction specifications concerned with updating views and guaranteeing the integrity of the database. The context of their work is deductive databases [98, 99] and their method augments the deductive database schema with a set of transition rules and internal event rules. A transition rule is a predicate defined in terms of the current state and the integrity constraints of the database, whereas an event rule is a predicate that specifies which operations (usually insertions and deletions) can happen as a result of a database update operation.

In addition, the authors describe a prototype tool which is implemented in Prolog. The tool is capable of producing pseudo-code written in English and in Catalan, as well as Prolog implementation code written according to their method.

A previous paper by Pastor [100] described a similar method based on an extended version of the relational model which was augmented with the notions of transition rules and internal event rules.

3.4.2 The work of Sheard and Stemple

Sheard and Stemple [101] present a thorough and theoretically sound treatment for the verification of database transactions safety. They describe a theorem prover that can be used to prove that database transactions are safe in the sense that they do not violate the set of specified database constraints.

The formal theory used by the tool is based on the Boyer and Moore [102] style but is extended with higher order functions and theorems. The specification language is called the Abstract Database Type Programming Language (ADABTPL).

The authors claim that both the theory used to build the tool and the ADABTPL specification language are not restricted to the relational model. However, the specification language *does* include a number of features which are specifically based on the relational model and the example presented in the paper is an extensive relational database example.

On the other hand, the ADABTPL language does not cover the specification of dynamic constraints (called transition constraints by the authors), only covers the two simplest aggregate functions (count and sum), and does not provide an explicit structure to capture the foreign key constraints, even though these can still be specified.

3.4.3 The work of Steinberg, Faley, and Chinn

In a recent paper, Steinberg, Faley, and Chinn [103] describe a more practical approach. The main problem they propose to address is the fact that software developers often do not meet the needs of end users in a timely fashion.

The authors claim that one of the approaches to solve the problem is to encourage end users to get more involved in the design and development of the software they use. They also claim that one of the difficulties to achieve this goal is the fact that traditional *Computer Aided Software Engineering* (CASE) tools were developed primarily for the trained professional rather than the end user. The proposed solution is to use their tool, which is called *The Analyst*.

In addition, they assert the tool can be used by novice end users to design and implement customized relational database prototypes. Moreover, that this is achieved by writing *English sentences*.

Allegedly, the user would provide the entities, attributes, and possible queries, in addition to the attributes which should be listed in the results, using some restricted form of English sentences (e.g. pronouns are not accepted). The system then performs some validations and, when the prototype is acceptable to the user, the system generates the corresponding implementation code for either dBASE or Paradox.

According to the authors, the time taken to develop the applications is reduced, because the process depends less on the availability of human developers. Furthermore, they claim this shorter development time, together with standardization and automatic generation, diminishes the possibility of misunderstandings in the systems requirements and reduces the cost of software maintenance.

Finally, they state that the results of an experiment using graduate business students with no previous experience in systems analysis or programming demonstrated that users could match *almost exactly* the model task solution to the problem they were given in *little more than an hour*.

It is very difficult to assess the merits and limitations of this work without actually seeing the tool or the problems used in the described experiment. Nevertheless, it is clear that all these claims seem too good to be true. I suspect the class of problems that can be solved using the tool is very limited. Moreover, the treatment of database constraints must be very rudimentary if at all existent.

3.4.4 The work of Xiaolei Qian

This subsection discusses the work of Qian [104, 105], which is called the *Deductive Synthesis of Database Transactions*.

The general approach adopted involves the use of refinement techniques (called transaction synthesis by the author) to transform the initial declarative specifications into procedural implementations.

In other words, the transaction synthesis is the process of finding a transaction that satisfies the specification. This synthesis is formalized as the process of finding constructive proofs of specification theorems and extracting appropriate transactions from these proofs.

Proofs are represented as tables called deductive tableaux which consist of three lists of formulas: an assertion list, a goal list, and a transaction entry list. The synthesis system consists of deduction rules that manipulate the tableaux preserving its validity.

The proof system used to carry out the transaction synthesis is an extended version of the deductive-tableau proof system for first-order logic developed by Manna and Waldinger [106].

There are a number of aspects of this work which are similar to the research described in this thesis. These are:

- The work is driven by the belief that the automatic generation of database transactions is both desirable and feasible. The author claims the automatic generation of programs in a restricted but well understood and important domain is desirable, to take advantage of the well defined semantics of the database transactions and avoid the violation of the integrity constraints; and feasible, because such transactions are usually dominated by data manipulations rather than complex computations.

- The database state is explicitly characterized as a finite set of relations. The author claims it is "relatively simple" to define it this way and that, often, "it is possible to specify precisely the effect of every language construct on database states".

- The work assumes that "transactions are always executed in valid databases where integrity constraints are satisfied", i.e., the database is assumed to be in a valid state before any transactions are executed.

However, there are a number of important aspects which are different in the two approaches. The main differences are:

- This approach is much more formal than the one adopted in the research described in this thesis, with a lot of emphasis being put on reasoning about state transitions and proving that the resulting transactions satisfy the specifications.

- There is no explicit method and/or guidelines to help the users to write the formal specifications and to carry out the proofs from which the transactions are extracted. In other words, this approach requires a much higher knowledge of mathematics and is unlikely to be usable by developers of real database applications.

- The resulting transactions are not explicitly built to any existing RDBMS, only to a hypothetical system supporting the transaction language described in the paper.

3.4.5 The Hamburg work

Now, the approach adopted by the database group at the University of Hamburg is discussed in detail. A considerable part of this work was part of the DAIDA project[1]

Their approach to the derivation of database application defends the utilization of a formal method together with a conceptual design language as well as an implementation language in an integrated framework [107].

The main approach

In their main approach, they suggest that conceptual designs should be written using an expressive semantic data and transaction model, namely the TDL[2] language [108], which is derived from TAXIS [109]. In particular, TAXIS has been enriched with constructs for a predicative specification style. The extensions include multi-valued attributes, a set-oriented expression language, and the predicative techniques for specifying the dynamic parts of the system, i.e. transactions (atomic state changes), functions, and derived classes and attributes.

[1]DAIDA stands for Development of Advanced Interactive Data-intensive Applications. It was an ESPRIT project funded by EEC under research contract number 892.

[2]TDL stands for Taxis Design Language

Also, the database structures and constraints, initially written in TDL, should then be formally transformed into equivalent abstract machines, as defined by Abrial [110], using the B-Method [111]. The transactions are modelled by operations in the abstract machines. The proof obligations for guaranteeing consistency are semi-automatically verified using the B-Tool [112].

In the following step, these abstract machines should be refined into other abstract machines that are equivalent to programs written in the strongly typed programming language DBPL. In other words, they provide specific B specifications that are sufficiently refined to be directly translated to DBPL. According to the authors, it was the explicit specification of state and invariants and the possibility of stepwise refinement within the same language that made the abstract machine approach a natural choice for the specification of database applications.

Finally, these final B specifications should be translated to DBPL syntax.

The automatic transformation of TDL designs into abstract machines was described in a paper by Schewe, Schmidt, and Wetzel [107]. This paper has also provided a small set of refinement rules which formalize the transformation of these initial abstract machines into other machines which are equivalent to DBPL programs. It also describes which properties must be verified to guarantee transaction consistency and correct refinement, and indicate how to use a mechanical theorem proving assistant to guide the proofs.

A more recent paper by Günther, Schewe, and Wetzel [113] characterized the final B specifications that are equivalent to DBPL programs. In addition, it describes an automatic transformation of final B specifications into DBPL syntax. In the first part, the authors show that DBPL programs are indeed equivalent to certain B specifications. In the second part, they use the algebraic specification language and term rewriting system OBJ [35] to implement the mapping to DBPL syntax.

An alternative approach

An alternative approach based on a slight variation of the aforementioned scenario was also considered by Schewe, Schmidt, and Wetzel [114]. However, I believe it was never investigated in detail. Basically, they proposed a new database specification language called SAMT (Structured Abstract Module Types) that would add strong types to the abstract machine formalism and would support the idea of modules with import and export constructs, similar to modula-2 modules.

The main aim was to design a language that could be used to construct modular strongly-typed specifications already in the conceptual level, and also to refine these specifications into executable database programs. Hence, SAMT would substitute both TDL and the abstract machines in their original approach and, thus, it would eliminate some of the complexity issues of the multi-language approach.

The motivation to design SAMT was their will to overcome two problems in the original approach. These are:

- All objects that are part of the state are necessarily persistent. The reason this was considered a problem is the fact that they do not consider their approach to be restricted to the relational model.

- Their inability to automatically derive appropriate DBPL final data structures. The main problem is that they found it difficult to generate appropriate types for the structures since their specifications are untyped (B does not support types).

Comparison to my approach

There are some similarities between the Hamburg approach and the approach adopted in this thesis. Firstly, it is, in both cases, possible to prove, already at the specification level, that the transactions maintain the consistency of the database. This possibility was not pursued as part of this Ph.D. thesis though.

Secondly, the relational model has, in both approaches, been used as the main target for the generation of database applications. However, they do not provide any method or facilities to support specific features of the relational model, mainly because they do not consider their approach to be restricted to the relational model.

Finally, the implementation language used in both works is DBPL. Nevertheless, their approach to the mapping is specific to DBPL and is not easily adaptable to be used with another implementation language. In this thesis, DBPL is just the chosen example of a target database language which is used to instantiate the generic mapping. For this reason, their approach is less likely to be considered for the development of real relational database applications.

In spite of these similarities, the means used to achieve the main objective are rather different in the two approaches. Their emphasis was on the derivation of efficient DBPL programs and on proving, formally, that these programs do not violate the database constraints. My emphasis was on a specific method aimed at helping practitioners with the formal specification of relational applications and on a generic mapping that can be adapted to generate implementations to be run in any RDBMS.

Regarding the two problems which are present in their approach and were already mentioned, they are not problems in the work described in this thesis. Firstly, the fact that all the state is persistent is not a problem in my approach, because only the relations are part of the state and these must be persistent.

Although Z is not strongly typed, the strategy adopted for the method was to have strongly typed domains based on their names. This avoided the problem in the mapping of the structures, which was their second problem. However, some DBPL commands also use the types of the relations as part of the syntax, while the method does not. In these cases, it was possible to derive the types from the declaration of the structure part of the database.

3.5 Conclusion

This chapter surveyed the existing literature on the utilization of formal methods for the specification and derivation of applications and, in particular, the derivation of relational database transactions, which closes the first part of the thesis.

The next part, which starts with a detailed description of the developed method in Chapter 4 and includes another four chapters, constitutes the principal part of the thesis.

Chapter 4

The specification method

In this chapter a complete description of the method proposed in this thesis is presented. It is basically the description given in [115], with minor corrections, and represents its current status. The method is for the specification of relation database applications and was implemented in Z. Also, the word *schema* is generally used to refer to Z schemas.

This chapter is split into five sections: the first describes the specification of the database structure, i.e. domains, relations and their attributes, and the constraints to be guaranteed. The second describes the specification of basic operations over the database. The third describes the specification of more advanced features such as transactions, sorting of results, aggregate functions, composite attributes, and views. The fourth deals with the extension of the applications (transactions) to capture error handling, using two different approaches. Finally, Section 4.5 introduces a number of guidelines on how to use the method realistically.

The rules of the method will be named using labels of the type Xn, where X can be D, standing for database rules, B for basic operation rules, A for advanced feature rules, or E for extended application rules to capture error handling, and n will be a sequential number within each kind of rule, with subitems when necessary.

The reader may find it useful to refer to the specification of the simple example (Chapter 6) and even to the formal specification of the operators (Chapter 5) while reading the description of the method.

4.1 The database structure and constraints

Relations are specified as sets of tuples and this respects the original relational model defined by Codd [6]. In this model, relations, operations, etc. are expressed simply and this simplicity carries over to the proposed specification method. Also, the method does not enforce any constraints on the way relations and operations may be implemented.

D1 - Domains

Basically, domains are sets of values from which one or more attributes draw their values. The idea is to prevent comparisons of attributes that are not based on the same domain by strongly type-checking domains based on their names.

Domains are specified by abbreviation definitions based on other domains, possibly adding extra constraints, by enumerating the elements in free type definitions, or by given sets. Basic built-in types ($\mathbb{Z}, \mathbb{N}$, etc.) are considered basic domains. In the convention adopted for the method, domain names do not include lower-case letters.

$$DOM1 == \mathbb{N}$$

$$DOM2 == \{ n : \mathbb{N} \mid n < 18 \}$$

$$DOM3 ::= Elem1 \mid Elem2 \mid ... \mid ElemN$$

$$[DOM4]$$

D2 - Relations (Intention)

For each base relation there is a corresponding Z tuple type (record) which represents the intention of the relation. Its attributes ("variables" of the tuple type) must be of a valid domain. In the convention adopted for the method, the names of types do not include lower-case letters and the names of attributes begin with an upper-case letter.

Basically, a tuple type is a schema without its predicate part. According to this extension of Z [116], "only types can be used to define the domain of a variable in a schema definition", schemas cannot be used for this purpose.

$$REL \mathrel{\widehat{=}} [Att1 : DOM1 \; ; \;\; Att2 : DOM2 \; ; \;\; ...]$$

D3 - Relations (Extension)

For each tuple type defined according to D2, there will be a corresponding schema that declares a variable of type SET ($\mathbb{P}$) of the type defined earlier. These schemas will be referred to as the RE schemas elsewhere in the method. Their variables represent the extension of the relations. By convention, the names of schemas begin with an upper-case letter and the names of variables do not include upper-case letters.

Relat

$rel : \mathbb{P}\, REL$
...

Static constraints depending on a single relation are specified in the predicates of each of these RE schemas.

D3.1 - Required attributes

The constraints which state that specific attributes of the relations are required are specified using the operator *REQUIRED*. This operator takes two parameters: the relation and the attribute.

$$REQUIRED \; rel \; Att1$$

In fact, *REQUIRED* is only a syntactic sugaring for a more general operator called *NOT_NULL*, which takes one parameter more: the null value corresponding to the domain of the attribute.

NOT_NULL <null> *rel Att*1

As already mentioned, the formal specification of the operators, together with a more detailed explanation, are presented in Chapter 5.

D3.2 - Candidate Keys

Candidate keys, i.e., attributes or groups of attributes that uniquely identify the tuples of the relations, are specified using the operator *KEY_OF*. This operator takes two parameters: the relation and the attribute that is a candidate key.

*KEY_OF rel Att*1

The specification of composite attribute keys is presented as part of the advanced features in Section 4.3, rule A4.

D3.3 - Static Attribute constraints

The predicates of the RE schemas may also include any other static intra-relation constraints, e.g. specific integrity rules over the attributes of the relations.

The universal quantifier (∀) is used to state that it must be true for all tuples of the relation being defined, as follows, where <condition> is a boolean expression involving one or more attributes of t.

$\forall\, t : rel \;\bullet$ <condition>

D4 - The "Database" schema

A schema, e.g. *DB*, that will represent the Database as a whole, groups all database definitions by including the RE schemas that define the relations.

DB
*Relat*1
*Relat*2
...
Relatn
———
...

Static constraints depending on more than one relation are specified in the predicate of the database schema (*DB*).

D4.1 - Foreign Keys

All foreign keys are specified in the predicate of the database schema (DB), using the FOR_KEY operator.

A foreign key attribute $Fk1$, in relation $rel2$, referring to relation $rel1$, involving its primary key $Pk1$, is specified below. It means that, for all tuples of $rel2$, attribute $Fk1$ must either be null or match the primary key $Pk1$ of some tuple of relation $rel1$.

$FOR\_KEY\ rel2\ Fk1\ rel1\ Pk1$

In fact, FOR_KEY is, once again, a syntactic sugaring for a more general operator called $FOREIGN\_KEY$, which takes one parameter more: the null value corresponding to the domain of the attribute.

$FOREIGN\_KEY$ <null> $rel2\ Fk1\ rel1\ Pk1$

The specification of composite attribute foreign keys is also presented as part of the advanced features in Section 4.3, rule A4.

D4.2 - Other static constraints

The predicate of the database schema may also include the specification of other inter-relations static constraints over the database using the universal quantifier ($\forall$) to state that it must be true for all tuples of one or more relations. In particular, this includes the definition of inter-relation derived attributes.

$\forall\, t : rel1\ \bullet$ <condition>

where <condition> is a boolean expression involving attribute(s) of relation rel and at least another relation.

D5 - The ΔDB schema

A schema that includes the database before and after the operations will be defined to be used in the update operations, in order to make possible the distinction between the relations before the operations and the relations after the operations. Its name will, by convention, be the name of the database schema with the prefix Δ.[1]

ΔDB
DB
DB'
...

[1] In fact, this convention is part of Standard Z [3]. In general, such a schema does not include any predicates and is not explicitly specified, unless it is extended with predicates.

Dynamic constraints are specified in the predicate of the ΔDB schema according to rule D5.1.

D5.1 - Dynamic constraints

Dynamic constraints, i.e. constraints that also depend on the previous values of updated attributes, are specified using the universal quantifier ($\forall$) similarly to the specification of static attribute constraints (D3.3), except for the fact that now the boolean expression <condition> includes variables from the state schemas before and after the operations (DB and DB').

$\forall\, t : rel1;\ t' : rel1' \;\bullet$ <condition>

where <condition> is a boolean expression involving one or more attributes of tuples t and t', i.e. relation $rel1$ before and after the operations.

D6 - The ΞDB schema

The ΔDB schema will be extended by the definition of a new schema that will be used in the specification of the read-only operations. It adds an invariant stating that all its variables will be unchanged after the operations. Its name will, once again by convention, be the prefix Ξ added to the name of the original schema.

$$\Xi DB \;\widehat{=}\; [\ \Delta DB \mid \theta DB = \theta DB'\ ]$$

where θDB gives the binding of the DB schema. A binding is a tuple representing an instance of the values of the variables of a schema.

D7 - The Initial State Schema

A schema, e.g. $Init\_DB$, defines the initial state of the database by including the relations after the operations, i.e. schema DB', and stating that all relations are empty.

$$
\begin{array}{|l}
\hline
\textit{Init\_DB} \\
DB' \\
\hline
rel1' = \{\} \;\land \\
rel2' = \{\} \;\land \\
\ldots \\
reln' = \{\} \\
\hline
\end{array}
$$

4.2 The database Operations

Now, the rules of the method concerned with the specification of the operations are described. For organizational purposes, the operations are divided into two groups: read-only operations, which do not modify the database, and update operations, which modify the database by inserting, updating, or deleting tuples of relations.

B1 - Read-only operations

Read-only operations are specified by schemas such that: (1) they include the corresponding ΞDB schema, (2) they declare the input (if any) and output variables of the operations, (3) their output variables are usually relations, i.e., their types are $\mathbb{P}\, A$, where A is a tuple type that defines the intention (type of the tuples) of some relation, and (4) their predicates describe the result of the operations according to at least one of the three following rules (B2, B3, and B4).

B2 - Select

In the select operation, the set comprehension is used to describe the result as a set of tuples of a given relation (the variable which represents its extension) based on a select condition using its attributes.

$$res! \;=\; \{\; t : rel \;\;|\;\; \text{<condition>} \;\}$$

where <condition> is a boolean expression involving at least one of the attributes of t.

B3 - Theta-Join

Theta-joins, the most general form of joins, are described similarly but more than one relation is used and a join condition is specified, based on attribute(s) of all relations.

$$res! \;=\; \{\; t1 : rel1;\; t2 : rel2 \;\; ... \;\;|\;\; t1.Att1 \;\; \text{<cop>} \;\; t2.Att2 \;\; ... \;\}$$

where <cop> is a comparison operator, and $Att1$ and $Att2$ are attributes of relations $rel1$ and $rel2$ respectively.

B4 - Project

The (extended) project operation is similar to the select operation. The difference is the inclusion of the result, based on computations of some attributes of the qualifying tuples.

$$res! \;=\; \{\; t : rel \;\;\bullet\;\; \text{<result>} \;\}$$

where <result> is an expression that applied to t gives a tuple of type A, such that all its attributes are based on computations of attributes of rel. The type of the result variable ($res!$) is $\mathbb{P}\, A$.

B5 - Update operations

Update operations are specified by schemas that (1) include the ΔDB schema, (2) declare the input (if any) variables of the operations - normally there are no output variables, (3) specify what relations are changed by the operations using a schema expression based on the ΞDB schema, and (4) describe, in their predicates, the updates in one or more relations of the database according to one of the five following rules (B6 to B10) and/or the rules in the next section.

The main parts of such a schema are presented below, where <vars> refers to the declaration of all variables of the schema, \ is the schema hiding operator, and the first occurrence of "..." stands for the relation(s) that are modified by the operation. Notice that the ΞDB expression used in the predicate part of these schemas is an artifice to achieve a clearer way of saying what variables of the state are changed by the operations.

$$
\begin{array}{|l}
\hline
\textit{Op\_rel} \\
\Delta DB \\
< vars > \\
\Xi DB \setminus \ldots \\
\hline
\ldots \\
\hline
\end{array}
$$

B6 - Insert

A schema that describes inserting tuples in a given relation has one input variable - the set of tuples to be inserted - and its predicate states that the updated relation is the set union of the original relation and the input variable. A typical specification of such an operation is presented below.

$$
\begin{array}{|l}
\hline
\textit{Insert\_rel} \\
\Delta DB \\
sr? : \mathbb{P}\, REL \\
\Xi DB \setminus rel \\
\hline
rel' \; = \; rel \cup sr? \\
\hline
\end{array}
$$

In general, both relation variables (rel and rel') should have been hidden in the above specification, i.e. it should have been written $\Xi DB \setminus (rel, rel')$. However, in this specific case (the ΞDB schema expressions), it does not make any difference and avoiding the repetition should make it simpler for the user.

B7 - Delete by primary key

Schemas that specify deletions based on the primary keys of relations have one input variable - the primary keys of tuples to be deleted - and their predicates use the $DELETE$ operator to describe the operation.

For each relation, there will be two schemas: one without the expression $\Xi DB \setminus \ldots$, to be used in schema inclusions, and the other with that expression, to be used as a sub-transaction. The schema without the expression is needed because there are other rules of the method (e.g. B8) that will use the delete schemas in schema inclusions and the schema expression of the included schema could clash with the one in the schema being defined. The convention for naming these extra schemas is to add the suffix "_Pk" to the usual names.

The pair of schemas that specify the deletion from relation $rel1$ of the tuples with keys in the set $sk1?$ based on the primary key $Pk1$ are presented below.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ Delete\_rel1\_Pk \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta DB \\
sk1? : \mathbb{P}\, DOM1 \\
\hline
rel1' \;=\; DELETE\ rel1\ Pk1\ sk1? \\
\hline
\end{array}
$$

$$Delete\_rel1 \;\widehat{=}\; Delete\_rel1\_Pk \;\wedge\; \Xi DB \setminus ...$$

When the primary key of the relation is the target of one or more foreign keys, either in other relations or in the same relation, the predicate of such schemas must also specify what happens to all references for deleted tuples, in order to avoid violations of the *Referential Integrity Rule* [5, pp. 284–285].

In general, there are at least three possibilities [5, pp. 285–288], *Restricted*, *Cascades*, and *Nullifies* that, for each foreign key, are specified according to the rules B7.1, B7.2, and B7.3, respectively.

For this purpose, assume that attribute $Fk1$ of relation $rel2$ is a foreign key targeted at attribute $Pk1$ of relation $rel1$, $rel2$ not necessarily different from $rel1$, and that $sk1?$ is the set of values of $Pk1$ to be deleted.

B7.1 - Deletes restricted

When *Restricted* is chosen, deletes are performed only if there is no foreign key reference to any of the tuples selected ($sk1?$). This is already guaranteed by the specification of the foreign key constraint in the predicate of the corresponding DB schema.

To specify it explicitly and thus highlight this choice, the predicate of the schema that specifies deletes in relation $rel1$ should include the following equation:

$$\forall\, t2 : rel2' \;\bullet\; t2.Fk1 \notin sk1?$$

B7.2 - Deletes cascade

When *Cascades* is specified, every tuple where there is a foreign key reference to a deleted tuple is also deleted. The way this constraint is specified depends on whether the foreign key is part of a cycle of foreign keys that cascade for deletes or not.

(A) If the foreign key $Fk1$ is *not* part of such a cycle - and usually this is the case, the schema that specifies deletes in relation $rel1$ must include the expression

$$\begin{array}{l} \mathbf{let}\;\; sdr2 \;==\; \{\; t2 : rel2 \;\mid\; t2.Fk1 \in sk1? \;\bullet\; t2.Pk2 \;\} \;\bullet \\ \qquad\qquad\qquad\qquad\qquad Delete\_Rel2\_Pk\ [sdr2\ /\ sk2?] \end{array}$$

where $Delete\_Rel2\_Pk$ is, essentially, the schema $Delete\_Rel2$, i.e. the schema that specifies deletes for relation $rel2$ based on its primary key $Pk2$; $sk2?$ is the input variable of that schema; the **let** expression introduces a local variable; and the notation $[b \ / \ a]$ refers to the substitution of variables a by b in the referred schema.

In practice, the equation above means "use *Delete_Rel2* (or *Delete_Rel2_Pk*) to delete the tuples of relation $rel2$ that reference any of the deleted tuples of $rel1$".

(B) When the foreign key *is* part of such a cycle, the effect of *Cascades* cannot be specified in the same way, because there can be no cycles in the use of schemas as predicates. Notice however that the existence of such a cycle should be avoided whenever possible, because it can potentially destroy the database.

If the cycle is really needed, the predicates of schemas which specify deletes based on the primary keys of *all relations involved* state that, after the deletion of the set of keys selected, all relations after the operation are the maximal subsets of the original relations to satisfy the database constraints.

Suppose that (1) there is a foreign key in relation $rel1$ targeted at the primary key $Pk2$ of relation $rel2$, (2) there is a foreign key in relation $rel2$ targeted at the primary key $Pk1$ of relation $rel1$, and (3) *Cascades* is chosen for deletes in both. The predicate of the schema which specifies the deletion of tuples of relation $rel1$ based on its primary key is presented below.

$$
\begin{array}{l}
(\ \mathbf{let}\ \ rel1d\ ==\ DELETE\ rel1\ Pk1\ sk1?\ \bullet \\
\qquad rel1' \subseteq rel1d\ \wedge\ rel2' \subseteq rel2\ \wedge \\
\qquad \neg\ (\ \exists\, r1 : \mathbb{P}\, REL1;\ r2 : \mathbb{P}\, REL2\ \mid\ r1 \subseteq rel1d\ \wedge\ r2 \subseteq rel2\ \bullet \\
\qquad\qquad\qquad (\ (rel1' \subset r1 \vee rel2' \subset r2)\ \wedge \\
\qquad\qquad\qquad\ \ \Delta DB\ [r1\ /\ rel1',\ r2\ /\ rel2']\ \ )\ )\ )
\end{array}
$$

Notice that, in this case, the predicate above is the full predicate of such a schema and therefore it does not follow the general rule (B7) that prescribes the use of the *DELETE* operator. Also, the specification of such schemas for cycles of three or more foreign keys are not going to be presented, but they are similar to the one above.

(C) In the particular case of delete *Cascades* where relations $rel1$ and $rel2$ are identical, i.e. foreign key $Fk1$ refers to the same relation, $Fk1$ represents a particular case of cycle (loop). Consequently, delete *Cascades* in relation $rel1$ may still be specified according to the above rule as follows:

$$
\begin{array}{l}
(\ \mathbf{let}\ \ rel1d\ ==\ DELETE\ rel1\ Pk1\ sk1?\ \bullet \\
\qquad rel1' \subseteq rel1d\ \wedge \\
\qquad \neg\ (\ \exists\, r1 : \mathbb{P}\, REL1\ \mid\ r1 \subseteq rel1d\ \bullet \\
\qquad\qquad\qquad (\ rel1' \subset r1\ \wedge\ \Delta DB\ [r1\ /\ rel1']\ \ )\ )\ )
\end{array}
$$

However, because this particular case is comparatively more common, an operator called *CASC_DELETE* is provided to be applied in this situation. It takes four parameters: the foreign key $Fk1$ and the three parameters of *DELETE*. The resulting predicate of the schema is:

$$rel1' = CASC\_DELETE\ \ Fk1\ \ rel1\ \ Pk1\ \ sk1?$$

B7.3 - Deletes nullify

Finally, *Nullifies* changes all foreign references for deleted tuples to contain the null value and this constraint is specified using the operator *UPDATE*.

The *UPDATE* operator takes four parameters: the relation to be updated and one of its attributes, a set of values of this attribute, and a new value for this attribute. Its effect is to update the attribute to the new value, in all tuples where its old value is a member of the set of values given.

$$rel2' \;=\; UPDATE\ rel2\ Fk1\ sk1?\ \text{<null>}$$

where <null> is the null value for the domain of $Fk1$ (and $Pk1$ as well).

When $rel1$ and $rel2$ are the same relation, a **let** expression and a local variable must be used to join the equation above with the one that specifies the deletions (B7) because, in this case, the result of one operation must be the input to the other. The order of the equations does not make any difference and the case in which the references are nullified before the tuples are deleted was chosen.

$$\textbf{let}\ \ r\ == \ UPDATE\ rel1\ Fk1\ sk1?\ \text{<null>}\ \bullet \\ rel1' \;=\; DELETE\ r\ Pk1\ sk1?$$

B7.4 - Special case (recursive cascade deletes)

When a given relation $rel1$ is subject to recursive cascade deletes, because it is part of a cycle of foreign keys that cascade for deletes - B7.2 (B), the rules B7.1, B7.2 (A), and B7.3 need to be slightly changed.

Basically, the effect of deletes to foreign keys of relations which are not part of the cycle must be specified in terms of the set of tuples effectively deleted, instead of the set of tuples originally selected for deletion ($sk1?$).

The set of keys effectively deleted is, in all cases, the set difference between $rel1$ and $rel1'$ projected over its primary key $Pk1$, which is exactly $sk1?$ in most cases.

$$\{\ t1 : (rel1 \setminus rel1')\ \bullet\ t1.Pk1\ \}$$

In those cases, i.e., when a relation is subject to recursive cascade deletes, the equation above should be written in all places where $sk1?$ appears, in the description of rules B7.1, B7.2 (A), and B7.3.

B8 - Other deletes

Any other deletes are specified in terms of the ones defined by the schemas of rule B7, i.e., the deletes based on the primary key of the relations. This is achieved by using a substitution of variables, a **let** expression, and a selection of tuples of the relation, projected over its primary key as follows:

$$\textbf{let}\ \ sdr1\ ==\ \{\ t : rel1 \mid \text{<condition>}\ \bullet\ t.Pk1\ \}\ \bullet \\ Delete\_Rel1\_Pk\ [sdr1\ /\ sk1?]$$

where <condition> is once again a boolean expression based on one or more attributes of relation $rel1$.

B9 - Update

A schema that describes updating of tuples in a given relation is specified in terms of (1) a select condition, that determines the set of tuples to be updated, and (2) an update rule, that gives the updated tuple for each tuple selected. Its predicate, presented below, states that the relation after the operation is the set of updated tuples together with the set of tuples which were not selected.

$$rel' = \{\ t : rel \bullet \mathbf{if}\ <\mathrm{condition}>\ \mathbf{then}\ <\mathrm{result}>\ \mathbf{else}\ t\ \}$$

where <condition> is a boolean expression based on attributes of relation rel and <result> is an expression that, applied to t, gives the corresponding updated tuple.

Even though <result> may be any expression of type REL, it usually is an expression like the one presented below, such that $Att1$, $Att2$, etc. are the modified attributes and $v1$, $v2$, etc. are expressions which give the updated values for these attributes.

$$t \setminus (Att1 = v1, Att2 = v2, \ ...)$$

The *But* operator (\) is an extension that makes possible the modification of one or more attributes of variables of a tuple type, preserving the values of the other attributes of the tuple. This is a particularly useful extension because those specifications need not be changed if new attributes are included in the corresponding relations.

B10 - Update of keys

In the relational model, the update of the primary keys of one or more tuples of a relation is specified similarly to the update of any other attribute of the relation. Thus, it may still be specified according to the general rule for updates (B9).

However, because in this specific case updates are usually based on the old values of the primary keys and change only one tuple at a time, the operator $UPDATE$ is to be used in such specifications. A schema that changes the primary key $Pk1$ of relation $rel1$ from $old?$ to $new?$ is presented below.

Update_key_rel1

ΔDB

$old?, new? : DOM1$

$\Xi DB \setminus ...$

$rel1' = UPDATE\ rel1\ Pk1\ \{old?\}\ new?$

When the primary key of the relation is the target of one or more foreign keys, either in other relations or in the same relation, the predicate of such a schema must also specify what happens to all references for the updated tuple, in order to avoid violations of the *Referential Integrity Rule* [5, pp. 284–285], similarly to the case of deletes based on the primary keys.

In general, there are at least the same three possibilities [5, pp. 285–288], *Restricted*, *Cascades*, and *Nullifies* that, for each foreign key, are specified according to the rules B10.1, B10.2, and B10.3, respectively.

For the following subsections, assume that attribute $Fk1$ of relation $rel2$ is a foreign key targeted at the primary key attribute $Pk1$ of relation $rel1$, where $rel2$ is not necessarily different from $rel1$.

B10.1 - Updates restricted

When *Restricted* is chosen, updates are performed only if there is no foreign reference to the selected key ($old?$). This is already guaranteed by the specification of the foreign key constraint in the predicate of the corresponding DB schema.

To specify this explicitly, the predicate of the schema that specifies updates in the primary key of relation $rel1$ should include the equation below, which is very similar to the one described in rule B7.1.

$$\forall t2 : rel2' \bullet t2.Fk1 \neq old?$$

B10.2 - Updates cascade

When *Cascades* is specified, every tuple where there is a foreign reference to an updated tuple is also updated and this constraint is specified using the operator $UPDATE$.

$$rel2' = UPDATE\ rel2\ Fk1\ \{old?\}\ new?$$

When $rel1$ and $rel2$ are the same relation, a **let** expression and a local variable must be used to join the equation above with the one that specifies the update of the primary key (B10) because, in this case, the result of one operation must be the input to the other, similarly to the *Nullifies* option in the specification of deletes (B7.3). Again, the order of the equations does not make any difference and the case in which the foreign references are updated before the original tuple is updated was chosen.

$$\begin{array}{l} \mathbf{let}\ \ r == UPDATE\ rel1\ Fk1\ \{old?\}\ new? \bullet \\ \qquad\qquad\qquad rel1' = UPDATE\ r\ Pk1\ \{old?\}\ new? \end{array}$$

B10.3 - Updates nullify

Finally, *Nullifies* changes all foreign references for updated tuples to contain the null value. This constraint is specified very similarly to the way *Cascades* was specified, the difference being that the updated value of the foreign references is the null value.

$$rel2' = UPDATE\ rel2\ Fk1\ \{old?\}\ \text{<null>}$$

where <null> is the null value for the domain of $Fk1$ and hence the domain of $Pk1$.

Again, when $rel1$ and $rel2$ are the same relation, a **let** expression and a local variable must be used and the updates must be specified as follows:

$$\textbf{let}\ \ r\ ==\ UPDATE\ rel1\ Fk1\ \{old?\}\ \text{<null>}\ \bullet$$
$$rel1'\ =\ UPDATE\ r\ Pk1\ \{old?\}\ new?$$

4.3 The advanced features

In this section the rules of the method regarding the specification of more advanced features such as transactions, sorting of results, composite attributes, etc. are presented.

A1 - Transactions

Transactions are specified using the schema piping (>>) of operations written according to other rules of the method.

Notice that the version of the piping operator (>>) used here allows for the output *and* primed state variables (*all results*) of the first schema to be matched against the input *and* unprimed state variables of the second schema, respectively. It is not part of standard Z but there are no technical problems involved in such an extension.

In addition, renaming variables of the component schemas is usually necessary to make variables of different operations be the same variable, avoid name clashes, and/or keep the ? and ! naming conventions for input and output variables valid in the transaction. Extra parentheses are sometimes needed to enforce an order in the association of the schemas and, in this case, the sequential composition ($\fatsemi$) may also be used.

The convention for naming the schemas that specify the correct behaviour of transactions is to add the suffix "_Ok". A typical transaction definition is presented below.

$$Transac1\_Ok\ \widehat{=}\ (\ Oper\_1\ [b1\ /\ a1, ...]\ >>\ ...\ >>$$
$$Oper\_n\ [b2\ /\ a2, ...]\ \mid\ \text{<condition>}\ )$$

where $Oper\_1, ..., Oper\_n$ are the components of the transaction and <condition> is an optional predicate used to make the values of variables of different component schemas refer to other variables. This will be necessary, for example, to specify constraints depending on the inputs of more than one subtransaction and to make the value of a variable refer to the value of an attribute of a tuple variable.

A2 - Sorting of Results

The specification of sorting of results, i.e. the presentation of the results (values of output variables) of read-only operations in a specified order, uses the operator $SORT$.

The $SORT$ operator takes three parameters: a relation, an attribute of the relation, and a comparison operator to compare values of the type of the Attribute. Its result is a sequence formed by the tuples of the relation sorted by the values of the attribute according to the comparison operator.

$$seq\ =\ SORT\ rel\ Att1\ \text{<cop>}$$

where <cop> is a comparison operator compatible with the type of the attribute. In general, the comparison operator $\leq$ is used for sorting in ascending order and $\geq$ for sorting in descending order respectively.

The specification of sorting results based on two or more attributes are presented as part of the composite attributes rule (A4).

Usually, the specification of sorting is done in a separate schema which (1) includes the ΞDB schema, (2) declares the relation to be sorted as input and a sequence of tuples of the appropriate type (the sorted relation) as output, and (3) uses the $SORT$ operator as illustrated above in its predicate. A typical sorting schema is presented below.

Rel_sorted_Att1

$$\Xi DB$$

$$srel? : \mathbb{P}\, REL$$
$$lrel! : \mathrm{seq}\, REL$$

$$lrel! \; = \; SORT \; srel? \; Att1 \; \leq$$

A3 - Aggregate Functions

A facility commonly provided by many RDBMSs is the use of a number of aggregate functions which make describing the functionality of some applications easier.

Aggregate functions are usually applied to the definition of read-only applications, but are not restricted to these. The aggregate functions provided by the method are presented below.

The first one, #, gives the number of tuples of a relation. Note that the standard Z operation for number of elements of sets is used in this case.

The others, $COUNT$, MAX, MIN, SUM, and $AVER$ are operators defined to be used with the method. They all take two parameters, a relation and an attribute of the relation, and give a number as result. Except for $COUNT$, the range of the attribute must be that of one of the numeric types, i.e. $\mathbb{Z}, \mathbb{N}$, and $REAL$.

The $COUNT$ operator returns the number of tuples such that the given attribute is not null. Operators MAX, MIN, and SUM return the maximum, the minimum, and the sum of the values of the given attribute, respectively. Finally, $AVER$ returns the average value of the attribute. Note that none of them takes into account tuples where the value of the attribute is null. An example is presented below.

$$COUNT \; rel \; Att$$

Once again, these operators are only syntactic sugaring for more general operators that take extra parameter(s). For example, $COUNT$ is only syntactic sugaring for the general operator $COUNTS$, which takes one more argument as its first parameter: the null value corresponding to the domain of the attribute.

$$COUNTS \; \text{<null>} \; rel \; Att$$

More information on the operators involved in the specification of aggregate functions are given in Section 5.6 together with the specification of the operators.

A4 - Composite Attributes

Composite attributes are needed to make possible the specification of composite candidate and foreign keys, as well as sorting of results based on more than one attribute.

For the specification of composite attribute candidate and foreign keys, the method prescribes the application of *partial parametrizations* of the operators $CA2$, $CA3$, etc., which are used as *higher-order functions*.

For example, $CA2$ is an operator that takes three arguments: the two attributes that will form the key and a tuple of the relation. Its result is the tuple formed by the values of the two attributes in the given tuple. For instance, the specification of a two-attribute candidate key is presented below, where $Att1$ and $Att2$ are attributes of relation $rel1$.

$$KEY\_OF \;\; rel1 \;\; (CA2 \; Att1 \; Att2)$$

The expression inside parenthesis returns a *function* with the first two parameters of $CA2$ instantiated. Technical details about this are given in Chapter 5, together with the specification of the operators.

Similarly, a two-attribute foreign key targeted at the primary key of the relation $rel1$ defined above is presented below, where $Att3$ and $Att4$ are attributes of relation $rel2$.

$$FOR\_KEY \;\; rel2 \;\; (CA2 \; Att3 \; Att4) \;\; rel1 \;\; (CA2 \; Att1 \; Att2)$$

Sorting results based on two or more attributes are also specified using the operators $CA2$, $CA3$, etc. to represent the composite attributes. Operators called $COP2$, $COP3$, etc. are also used as *higher-order functions* together with *partial parametrization* to represent the composite comparison operators.

For example, $COP2$ takes 4 parameters: the two comparison operators for the types of each of the attributes, and the two pairs of values to be compared. Its result is a boolean, *true* if the pairs satisfy the comparison operator and *false* otherwise.

An example of sorting based on two attributes is presented below.

$$seq! \; = \; SORT \;\; rel1 \;\; (CA2 \; Att1 \; Att2) \;\; (COP2 \;\; \text{<cop1>} \; \text{<cop2>})$$

where <cop1> and <cop2> are comparison operators compatible with the types of attributes $Att1$ and $Att2$ respectively.

A5 - Views

Views are used to restrict the data visible to or updatable by a specific user or group of users. They may be composed of a number of base relations and virtual relations derived from base relations. The use of modularization structures together with the method should provide the means for hiding base relations and, thus, prevent the users from accessing the relations they are not authorized to use.

The updatability of views is not addressed since this is in its own right a whole area of research [54]. In general, it is not even possible to decide whether some views are updatable or not [55]. Moreover, it is not always clear what the semantics of updates of specific views should be.

Views are specified according to rules A5.1 to A5.9 presented below.

A5.1 - View Intention

For each view relation defined to be a projection or join of other relations, there is a tuple type definition which corresponds to the view relation intention. These tuple types are defined similarly to the intentions of base relations as described in rule D2.

$$VREL \mathrel{\widehat{=}} [\ Att1 : DOM1;\ Att2 : DOM2;\ \ldots\ ]$$

A5.2 - View state schema

For each view, there must be a view state schema which (1) includes the *DB* schema, (2) declares the variables that represent the extensions of all derived view relations, similarly to the declaration of the extensions of base relations - rule D3, (3) specifies the contents of these variables in terms of other variables (in general base relations) according to rules A5.3, A5.4, A5.5, and/or A5.6, and (4) may add extra constraints about these variables.

```
┌─ View ──────────────────────────────
│ DB
│
│ vrel1 : ℙ VREL1
│ ...
├──────────────
│ vrel1 = ... ∧
│ ...
└─────────────────────────────────────
```

A5.3 - Selects, joins, and projects

Views based on selects, joins, and projects, presented below, specify the contents of the view variables (relations) as set comprehensions, similar to the ones prescribed for the select, join, and project operations in rules B2, B3, and B4 respectively.

$$vrel1 = \{\ t1 : rel1 \mid \text{<condition>}\ \}$$

$$vrel2 = \{\ t1 : rel1;\ t2 : rel2\ \ldots \mid t1.Att1\ \text{<cop>}\ t2.Att2\ \ldots\ \}$$

$$vrel3 = \{\ t3 : rel3 \bullet \text{<result>}\ \}$$

where <condition> is a boolean expression involving at least one of the attributes of *rel*1, <cop> is a comparison operator, *Att*1 and *Att*2 are attributes of *rel*1 and *rel*2 respectively, and <result> is an expression that applied to $t3$ gives a tuple of type *VREL*3 such that all its attributes are based on computations of attributes of *rel*3.

A5.4 - Updates of Attributes

Views which involve changing the values of attributes specify the contents of the view variables (relations) using a select condition for the updates and an update rule which defines the corresponding updated tuple for each of the selected tuples, similarly to the updates of base relations, rule B9 of the method.

$$vrel \ = \ \{ \ t : rel \ \bullet \ \textbf{if} \ <\text{condition}> \ \textbf{then} \ <\text{result}> \ \textbf{else} \ t \ \}$$

As in rule B9, <result> may be any expression of type *VREL* though it usually is an expression like the one presented below, such that $Att1$, $Att2$, etc. are the modified attributes, $v1$, $v2$, etc. are expressions which give the updated values for these attributes, and \ is the *But* operator as defined in rule B9.

$$t \setminus (Att1 = v1, Att2 = v2, \ ...)$$

A5.5 - Inserts

Even though they are not common, view relations which involve the insertion of new tuples may be specified similarly to the insertion operation for base relations (rule B6). However, the variables containing the new tuples must be local variables introduced by a **let** expression.

$$\textbf{let} \ \ sr \ == \ \{ \ ... \ \} \ \bullet \ vrel \ = \ rel \cup sr$$

A5.6 - Deletes

There is no need for specifying view relations based on deletes since exactly the same result would be achieved using selects.

A5.7 - The $\Delta View$ schemas

For each *View* schema defined according to rule A5.2, there must be a $\Delta View$ schema which includes the *View* schemas before and after the operations and is similar to the ΔDB schema described in rule D5 of the method.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\Delta View\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta DB \\
View \\
View' \\
\hline
\ ... \\
\hline
\end{array}
$$

Note however that the schema above also includes the ΔDB schema and that its predicate may include the specification of extra constraints. In particular, the predicate of $\Delta View$ should specify possible update restrictions on one or more relations of the view according to one of the cases described below.

(A) For each base or view relation $rel1$ that cannot be updated at all by users of $view1$, there must be an equation in the predicate of $\Delta View1$ stating that the relations before and after the operations are equal.

$$rel1' \; = \; rel1$$

(B) Whenever all tuples of relation $rel1$ can be neither deleted nor have their attributes updated by users of the view, i.e. insertion is the only update operation allowed, the following equation should be specified in the predicate of $\Delta View1$.

$$\forall\, t : rel1 \;\bullet\; t \in rel1'$$

(C) Similarly, provided that no tuples of $rel1$ can have their attributes updated and new tuples cannot be inserted, i.e. deletion is the only update operation allowed, the following equation should be specified.

$$\forall\, t : rel1' \;\bullet\; t \in rel1$$

(D) On the assumption that tuples of relation $rel1$ cannot be deleted but insertions and updates of attributes are allowed, the following equation should be specified.

$$\forall\, t : rel1 \;\bullet\; (\exists\, t1 : rel1' \;\bullet\; t.Pk1 = t1.Pk1)$$

(E) Likewise, if new tuples cannot be inserted in $rel1$ but deletions and updates of attributes are both allowed, the equation below should be specified instead.

$$\forall\, t : rel1' \;\bullet\; (\exists\, t1 : rel1 \;\bullet\; t.Pk1 = t1.Pk1)$$

Finally, there are other cases which are possible but were not included in the method because, in these cases, the forbidden operations can be achieved with one or more of the allowed operations. Additionally, it is worth emphasizing that whenever all update operations are allowed no extra equations are needed.

A5.8 - The $\Xi View$ schemas

Similarly, for each $View$ schema defined according to rule A5.2, there is a $\Xi View$ schema, similar to the ΞDB schema described in rule D6 of the method. Again for completeness, the $\Xi View$ schemas must be explicitly specified.

$$\Xi View \;\widehat{=}\; [\; \Delta View \;\mid\; \theta View = \theta View' \;]$$

A5.9 - Specification of operations

Finally, the specification of all the basic operations are basically the same as those described for the base relations (rules B1–B10), except that they use the view state schemas ($\Delta View$ and $\Xi View$) instead of the database state schemas (ΔDB and ΞDB).

Nevertheless, it is worth repeating that the updatability of views is not addressed by the method. Therefore, update operations on view relations may lead to problems which are not investigated in this work.

4.4 The extended operations for error handling

Basically, the method provides two different ways for the specification of the extended operations (transactions) which capture error handling.

In the first one, suitable for specifications targeted at an implementation, the specifications explicitly state what the possible errors[2] are and give a specific error message for each of them. This possibility is described by rules E1, E2, and E3 below.

E1 - The schema *Ok*

The Schema *Ok*, that is used in both possibilities for error handling, is specified below. In fact, *Ok* may be provided together with the pre-defined operators and, so, there is no need to specify it again.

Ok

$result! : STRING$

$result! \;=\; \text{“}\mathit{Success}\text{”}$

E2 - The error schemas

For each transaction using the database, there will be a corresponding error schema, which describes the possible errors that may occur. Basically, each of these schemas (1) include the ΞDB schema, because no change is done in the relations when errors occur, (2) introduce the variable *result*! to keep the error message, (3) declare all input variables (if any) declared by the corresponding schema that deals with the correct behaviour of the transaction, because they will be involved in some of the possible errors, and (4) describe, in its predicate, what the possible errors are and which messages correspond to each of them. A sketch of such an error schema is presented below.

Insert_rel1_Error

ΞDB

$result! : STRING$

<input_vars>

(<error_1> $\wedge$ $result! \;=\; \text{“}\mathit{message}\ 1\text{”}$) $\vee$

...

(<error_n> $\wedge$ $result! \;=\; \text{“}\mathit{message}\ n\text{”}$)

[2] In order to identify all possible error conditions, the negation of the preconditions of the corresponding schema that describes the correct behaviour of the transaction must be simplified.

where <input_vars> is the declaration of all input variables of the corresponding schema that deals with the correct behaviour of the transaction, <error_1>, ..., <error_n> are the possible errors, and "*message* 1", ..., "*message n*" are the corresponding error messages.

Whenever the number of possible errors of a transaction is considered too big, the corresponding error schema may be split into a number of smaller schemas. A possible policy to manage the split is to write one error schema for errors regarding each of the relations involved in the transaction.

Notice that all input variables of the transaction must be declared in all error schemas, although some of them may not be used in all error schemas. The reason for declaring them all is to avoid having to find out which variables are needed in each schema which is unnecessary since it would not change the final transaction.

E3 - The extended operations I

Now, the extended transactions, e.g. $Transac1$, are specified by extending their original specifications, i.e. $Transac1\_Ok$, to describe what happens if any error occurs. Basically, if the preconditions are satisfied the result is "success" (Ok), otherwise no change is done in the database and a specific message is put in $result!$ ($Transac1\_Error$).

$$Transac1 \ \widehat{=} \ (Transac1\_Ok \ \wedge \ Ok) \ \vee \ Transac1\_Error$$

where $Transac1\_Ok$ is the schema that describes the correct behaviour of the transaction and $Transac1\_Error$ is the corresponding error schema (E2).

Should the user decide to split the error schema, all error schemas must be connected by logical disjunctions as follows:

$$\begin{aligned} Transac1 \ \widehat{=} \ & (Transac1\_Ok \ \wedge \ Ok) \ \vee \\ & \quad Transac1\_Error\_1 \ \vee \ ... \ \vee \ Transac1\_Error\_n \end{aligned}$$

Alternatively, the users may adopt the second possibility which is very simple and is suitable for more abstract specifications. In this case, a general error schema is used to specify that the database is not modified if an error occurs, and its variable $result!$ simply says whether the operation was successful or not. The two following rules, E4 and E5, describe this second approach to error handling.

E4 - The schema *Error*

The schema $Error$ states that there will be no change in the relations of the database and that $result!$ is not "*Success*".

$$Error \ \widehat{=} \ \Xi DB \ \wedge \ \neg \ Ok$$

This schema is general in the sense that it is used to extend all applications, but specific for each database being specified.

E5 - The extended operations II

Again, the schemas that describe the extended transactions, e.g. $Transac1$, are then specified similarly to the ones prescribed by rule E3. The difference is that, now, the general schema $Error$ is used in substitution to the specific error schemas. Schemas $Transac1\_Ok$ and Ok were described in rule E3, whereas $Error$ was described in rule E4.

$$\begin{aligned} Transac1 \;\widehat{=}\; & (Transac1\_Ok \;\land\; Ok) \;\lor \\ & \qquad (Error \mid result! = \text{``}Error\ in\ Transaction\ 1\text{''}) \end{aligned}$$

4.5 Guidelines on how to use the method

Now, a number of guidelines on how to write relational database specifications in Z using the method are described. They are virtually identical to the ones given in [115] and represent the last version of the guidelines on how to the use the method sensibly.

The user may find it helpful to look at the example (Chapter 6) while reading these guidelines.

Basically, it is postulated that the users should not write the complete specification at once but rather split the task into a number of steps. They should write a first specification containing only a small subset of the details. The information left out of this first specification would then be gradually added in several steps.

4.5.1 Guidelines for the first specification

Initially, some guidelines on which details should be present in the first specification are presented. Essentially, the aim is to specify the most important transactions, e.g. the ones to be run by many users or perhaps the ones with strict performance requirements.

In fact, the specifications would only include a minimal subset of the database structure, namely that affected by the chosen transactions, i.e., some domains, relations, and attributes, and virtually no constraints but the primary keys.

The proposed steps for writing the first version of the specifications are:

G1 Choose a number of transactions which are to be specified in the first version and write a brief yet precise description of each of them. Preferably, these transactions should be the most important ones. It was decided that *no* target number of transactions would be given and, consequently, it is up to the user to decide what the ideal number would be.

G2 Identify which subset of the database structure is affected by the chosen subset of transactions and draw the restricted ER diagram showing only the structures involved in these transactions. This will only include some entities, relationships, and attributes, as well as the cardinality constraints.

It is assumed that proper tool support for drawing ER diagrams is available and, so, restricted versions of the complete ER diagram can be easily generated using a tool. The idea is to include the restricted ER diagram in the specification document to highlight its important aspects and thus make it easier to understand.

G3 Specify, using Z, the relational database schema equivalent to the resulting ER diagram following the basic steps prescribed by the method. This will only include a limited number of domains (defined as given sets unless they are simple and already known), tuples, and relation schemas, in addition to the database state schemas DB, ΔDB, ΞDB, and $Init\_DB$ (the initial state schema).

Do not specify any constraints except for the null and primary key constraints. Other constraints, e.g. the foreign key constraints, could also be specified at this stage if for some reason this is considered relevant.

G4 Specify the common basic operations which are always (or at least usually) needed, no matter what system is being specified. Since these may be used several times, they are specified before operations needed in specific transactions. Insertion and deletion based on the primary keys of all relations are the ones identified so far.

G5 Specify the basic operations needed in the specification of the chosen transactions, except for the ones already specified. This includes the read-only operations (selects, projects, and joins), update operations (inserts, deletes, and updates), as well as some of the advanced features (sort, aggregate functions, etc.). Constraints which are specific to some subtransactions as well as constraints which will be automatically satisfied by other operations should also be explicitly specified.

G6 Finally, specify the correct behaviour of the database transactions based on the basic operations using the method. The derivation of the preconditions as well as the specification of the extended operations including error handling should be left for later. It appears to be more appropriate to add these as one or more extra steps apart from the specifications, since this is a more or less distinct activity. Further discussion on this matter is presented later.

4.5.2 Guidelines for extending the specification

Now, some guidelines on how to change the specifications resulting from each step, and in particular the first specification, to add more details are given. Basically, the details which should be added in the next steps are enumerated.

Also, no specific order in which these details are to be added is enforced. It might indeed be more convenient to mix different kinds of detail in a given step. Moreover, no sketch of the preferred order for adding the new details are given, since it ought to be up to each user to choose what the best order is. An example of using the method is given in [117, 118, 119] which shows a specification of the company database example. Chapter 6 contains the final version of this specification.

The kinds of detail that can be added in each step are:

Constraints: The users can add specific kinds of constraints and change the affected specifications accordingly. These kinds of constraints include the static attribute constraints, candidate keys (other than the primary keys), foreign key constraints, other static constraints depending on more than one relation (e.g. inter-relation derived attributes), and dynamic constraints.

Even though this could be done in a single step, it is suggested it should be split into some steps based on the kinds of constraints added. Even so, similar constraints should always be added in the same step. Different kinds of constraints can be added in a single step though.

Error Handling: Users would add the specification of the extended transactions which allow for error handling. This includes the derivation of the preconditions of each transaction. It can either be split into a number of steps or done at once for all transactions as a final step. Although this could also be done as part of every step that adds the specification of new transactions, it should be clearer to do it using separate steps.

Other Transactions: Users can also choose a number of other transactions that are to be specified and this will lead to more details being added to the specification of the database structure. In theory, it should not lead to changes in the transactions already specified. Again, the users do not have to specify all remaining transactions in a single step and indeed they should not do so.

There are two ways in which the above extensions can be added to the specifications. The first one is to use the facilities of schema inclusion provided by the schema calculus of Z to add the new details and generate a new set of specifications. This allows the user to record the order in which details were added.

Its disadvantage is the fact that the specifications need to be completely rewritten in each step, even the bits that are not affected by the new details. This is so because the schema inclusion of Z is merely a textual inclusion. It seems however that a version control facility for Z schemas would help to overcome this problem.

The other possibility is to change the specifications in much the same way application programs are usually changed. Essentially, this means that new details are added directly in the previous version of the specifications. In this case, only the bits affected by the new details are changed. Its disadvantage would be the fact that, even if copies of the previous phases are made (and this is recommended), the order in which details were added may not be clearly recorded.

If a version control facility is not available, using the second approach might be more suitable because, most of the time, a clear view of the resulting specifications is what the users need. Recording exactly which details are added in each step does not seem particularly relevant. Moreover, a brief description of what details are added in each step can be recorded in the textual parts of the specification.

Nevertheless, both alternatives can be used without problems. It might be that they could even be used together in different parts of the specifications.

4.6 Conclusion

In this chapter a complete description of the method for the specification of relation database applications was presented.

The following chapter then provides the formal specifications of all the operators informally defined in the present chapter.

Chapter 5

The operators

In this chapter, the formal definition of the operators applied in the specification of relational databases and applications, informally introduced last chapter, is presented. Again, these descriptions are basically the same as those introduced in [115], with minor corrections, and represent the final version of the operators.

Before the specification of the operators, two given sets and an enumeration type are introduced. These are used in the specifications written according to the method to represent strings, reals, and booleans respectively.

$$[STRING, REAL]$$

$$BOOL ::= true \mid false$$

5.1 Primary and candidate key operator

The KEY_OF operator is used to specify that a specific attribute of a relation is a candidate key and, in particular, this includes the primary key. It is defined below and takes two parameters: a relation (rel - its type is $\mathbb{P}\,A$) and an attribute of the relation (Att - its type is $(A \rightarrow B)$, a function from the tuple to the value of the attribute), where A and B are generic types.

$$[A, B]$$

$$KEY\_OF : \mathbb{P}\,A \rightarrow (A \rightarrow B) \rightarrow BOOL$$

$$\begin{array}{l} \forall\, rel : \mathbb{P}\,A;\ Att : (A \rightarrow B) \bullet \\ \qquad KEY\_OF\ rel\ Att \Leftrightarrow \\ \qquad\qquad (\forall\, t1, t2 : rel \bullet Att\ t1 = Att\ t2 \Leftrightarrow t1 = t2) \end{array}$$

The predicate of KEY_OF states that, for every pair of tuples of the relation having the same value for the attribute, the tuples are identical, i.e., they are in fact the same tuple. This means that every tuple of the relation must have a different value for the chosen attribute.

5.2 The *FOREIGN_KEY* operator

The operator *FOREIGN_KEY*, used to define Foreign Keys, is presented below. This operator takes five arguments: a value of type C, that represents the null value for this type; the relation where the foreign key is; the foreign key attribute; and the relation and its primary key referred to by the foreign key.

$$
\begin{array}{l}
[A, B, C] \\
\hline
FOREIGN\_KEY : C \rightarrow \mathbb{P}\, A \rightarrow (A \rightarrow C) \rightarrow \\
\qquad\qquad\qquad\qquad \mathbb{P}\, B \rightarrow (B \rightarrow C) \rightarrow BOOL \\
\hline
\forall\, null : C;\ rel2 : \mathbb{P}\, A;\ Fk1 : (A \rightarrow C);\ rel1 : \mathbb{P}\, B;\ Pk1 : (B \rightarrow C) \bullet \\
\qquad FOREIGN\_KEY\ null\ rel2\ Fk1\ rel1\ Pk1 \Leftrightarrow \\
\qquad\qquad (\ KEY\_OF\ rel1\ Pk1\ \wedge \\
\qquad\qquad \forall\, t2 : rel2 \bullet Fk1\ t2 = null\ \vee \\
\qquad\qquad\qquad\qquad \exists\, t1 : rel1 \bullet Pk1\ t1 = Fk1\ t2\ )
\end{array}
$$

The above predicate states that attribute $Fk1$ of relation $rel2$ is a foreign key targeted at attribute $Pk1$ of relation $rel1$. This means that $Pk1$ is a key (in fact the primary key) of $rel1$, and that the value of $Fk1$ in all tuples of $rel2$ is null or matches the value of $Pk1$ for some tuple of $rel1$, $rel1$ being not necessarily different from $rel2$.

5.3 Null value operators

There is no universally accepted approach to null (missing attribute) values. Also, giving a different treatment for each possibility is not intended here. On the contrary, the aim is simply to show a basic approach and leave the possibility of others being specified by different users of the method. In addition, the treatment for null values is in general dependent on the specific DBMS and the method is intended to be independent of specific DBMSs. For these reasons, the treatment of nulls presented here is quite concise.

Regardless of the approach chosen for nulls, an operator called *NOT_NULL* will be used to specify that a specific attribute of a relation is mandatory, i.e., it cannot be null. It seems this operator will be useful for many treatments of nulls, but it is by no means universal. Thus, it may need to be changed depending on the approach chosen for nulls.

Operator *NOT_NULL*, specified below, takes three arguments: a value of type B, that represents the null value for this type, a relation, and an attribute of the relation.

$$
\begin{array}{l}
[A, B] \\
\hline
NOT\_NULL : B \rightarrow \mathbb{P}\, A \rightarrow (A \rightarrow B) \rightarrow BOOL \\
\hline
\forall\, null : B;\ rel : \mathbb{P}\, A;\ Att : (A \rightarrow B) \bullet \\
\qquad NOT\_NULL\ null\ rel\ Att \Leftrightarrow (\forall\, t : rel \bullet Att\ t \neq null)
\end{array}
$$

In this work, Date's proposal called the *default values* approach [120] is adopted. It was chosen because it is very simple, and is presented below.

(A) Null values

There is a constant definition for each basic type ($\mathbb{Z}, \mathbb{N}, REAL$, and $STRING$), which represents the null value of these types. Their values are assumed to be zero, for $\mathbb{Z}, \mathbb{N}$ and $REAL$, and the empty string for $STRING$, but these may be changed by the users.

$$
\begin{array}{|l}
NULLINT : \mathbb{Z} \\
NULLNAT : \mathbb{N} \\
NULLREAL : REAL \\
NULLSTR : STRING \\
\hline
NULLINT = 0 \ \wedge \\
NULLNAT = 0 \ \wedge \\
NULLREAL = 0 \ \wedge \\
NULLSTR = \text{“”}
\end{array}
$$

According to this approach, different domains based on the same basic type may have different null values. However, for the sake of simplicity, this case is not considered. Anyway, there is nothing to prevent users from specifying one null constant for each domain if this is needed.

(B) The *REQUIRED* operators

In order to simplify the specification of constraints which state that specific attributes cannot be missing, the operator $REQUIRED$ is introduced below. In fact there are four definitions of $REQUIRED$ (overloaded) that, for each of the basic types, instantiate the operator NOT_NULL by giving the corresponding constant as the first parameter.

$$
\begin{aligned}
REQUIRED\ [A] &\mathrel{\widehat=} NOT\_NULL\ [A, \mathbb{Z}]\ \ NULLINT \\
REQUIRED\ [A] &\mathrel{\widehat=} NOT\_NULL\ [A, \mathbb{N}]\ \ NULLNAT \\
REQUIRED\ [A] &\mathrel{\widehat=} NOT\_NULL\ [A, REAL]\ \ NULLREAL \\
REQUIRED\ [A] &\mathrel{\widehat=} NOT\_NULL\ [A, STRING]\ \ NULLSTR
\end{aligned}
$$

Note that the expressions above return other operators (functions) with the first parameter of NOT_NULL instantiated and exemplify the use of the operators as higher-order functions.

One of the $REQUIRED$ operators resulting from these definitions is presented below to facilitate its understanding and must be seen as a comment only. It corresponds to the instantiation of the first parameter of NOT_NULL using $NULLINT$.

$$
\begin{array}{|l}
[A] \\
\hline
REQUIRED : \mathbb{P}\, A \ \rightarrow \ (A \rightarrow \mathbb{Z}) \ \rightarrow \ BOOL \\
\hline
\forall\, rel : \mathbb{P}\, A;\ Att : (A \rightarrow \mathbb{Z}) \ \bullet \\
\qquad REQUIRED\ rel\ Att \ \ \Leftrightarrow \ \ (\forall\, t : rel \ \bullet \ \ Att\ t \neq NULLINT)
\end{array}
$$

Obviously, the *REQUIRED* operators are only syntactic sugar. However, using them allows the specification of this type of constraints to be done uniformly and independently of the types of the attributes [121, pp. 169–206].

(C) The *FOR_KEY* operators

Similarly, overloaded operators called *FOR_KEY* are defined by partial parametrizations of *FOREIGN_KEY*, already defined in Section 5.2, on its first parameter.

$$\begin{aligned}
FOR_KEY\ [A,B] &\mathrel{\widehat=} FOREIGN_KEY\ [A,B,\mathbb{Z}]\ \ NULLINT \\
FOR_KEY\ [A,B] &\mathrel{\widehat=} FOREIGN_KEY\ [A,B,\mathbb{N}]\ \ NULLNAT \\
FOR_KEY\ [A,B] &\mathrel{\widehat=} FOREIGN_KEY\ [A,B,REAL]\ \ NULLREAL \\
FOR_KEY\ [A,B] &\mathrel{\widehat=} FOREIGN_KEY\ [A,B,STRING]\ \ NULLSTR
\end{aligned}$$

In this approach, nulls are in fact valid values which are used for this specific purpose. Consequently, the users must be aware that comparison operators treat these null values identically to all other values. So, these values must be explicitly excluded from the context of the comparisons, when necessary.

In particular, joins based on attributes such that both of them allow nulls, as well as joins *not* based on the equality, usually require the explicit exclusion of nulls. However, because most joins are equi-joins based on a primary-key, and primary-keys do not accept nulls, this will rarely be necessary.

5.4 Update and delete operators

Now, auxiliary operators used to simplify the specification of operations such as update and delete are introduced.

The first of them, *UPDATE*, simplifies the specification of updates for a specific attribute *Att* of a given relation *rel*. It is used in the specification of the *Nullifies* compensating action of deletes based on the primary key which might violate the foreign key constraints - rule B7.3, and in the update of primary keys - rule B10.

The effect of *UPDATE*, presented below, is to change the value of attribute *Att* to *new* for those tuples in *rel* where the value of *Att* is a member of *old*. Operator $\setminus$ is the *But* operator as described in rule B9 of Chapter 4.

$$\begin{array}{l}
[A,B] \\
UPDATE : \mathbb{P}\,A \rightarrow (A \rightarrow B) \rightarrow \mathbb{P}\,B \rightarrow B \rightarrow \mathbb{P}\,A \\
\hline
\forall\, rel : \mathbb{P}\,A;\ Att : (A \rightarrow B);\ old : \mathbb{P}\,B;\ new : B \bullet \\
\quad UPDATE\ rel\ Att\ old\ new\ = \\
\qquad \{\ t : rel \bullet \textbf{if}\ \ Att\ t \in old \\
\qquad\qquad \textbf{then}\ t \setminus (Att = new) \\
\qquad\qquad \textbf{else}\ \ t\ \}
\end{array}$$

The next one, *DELETE*, is used to specify deletes in a given relation *rel* based on a given set of values *sv* of a specific attribute *Att*, usually the primary key of the relation. The result is the relation after the operation.

$$
\begin{array}{l}
[A, B] \\
\hline
DELETE : \mathbb{P}\, A \ \rightarrow \ (A \rightarrow B) \ \rightarrow \ \mathbb{P}\, B \ \rightarrow \ \mathbb{P}\, A \\
\hline
\forall\, rel : \mathbb{P}\, A;\ Att : (A \rightarrow B);\ sv : \mathbb{P}\, B \ \bullet \\
\qquad DELETE\ rel\ Att\ sv \ = \ \{\ t : rel \ \mid \ Att\ t \notin sv\ \}
\end{array}
$$

The effect of *DELETE* is to remove all tuples of relation *rel* where the value of attribute *Att* is a member of the set *sv*.

The operator *CASC_DELETE*, specified below, is recursive and represents recursive cascade deletes over a relation, i.e., when there is a foreign key for the relation where it is defined and *Cascades* is specified for deletes.

Basically, *CASC_DELETE* represents recursive applications of *DELETE* such that the set of foreign key references for deleted tuples in each step ($sb1$) will be the set of primary keys of tuples to be deleted in the following step. The recursion stops when there is no foreign reference to be deleted, i.e., when $sb1$ is the empty set.

$$
\begin{array}{l}
[A, B] \\
\hline
CASC\_DELETE : \ (A \rightarrow B) \ \rightarrow \ \mathbb{P}\, A \ \rightarrow \ (A \rightarrow B) \ \rightarrow \ \mathbb{P}\, B \ \rightarrow \ \mathbb{P}\, A \\
\hline
\forall\, rel : \mathbb{P}\, A;\ Fk, Pk : (A \rightarrow B);\ sb : \mathbb{P}\, B \ \bullet \\
\quad CASC\_DELETE\ Fk\ rel\ Pk\ sb \ = \\
\qquad (\ \mathbf{let}\ \ sb1 \ == \ \{\ t : rel \ \mid \ Fk\ t \in sb \ \bullet \ Pk\ t\ \} \ \bullet \\
\qquad\quad \mathbf{if}\ \ sb1 = \{\ \} \\
\qquad\quad \mathbf{then}\ DELETE\ rel\ Pk\ sb \\
\qquad\quad \mathbf{else}\ \ (\ \mathbf{let}\ \ r \ == \ DELETE\ rel\ Pk\ sb \ \bullet \\
\qquad\qquad\qquad CASC\_DELETE\ Fk\ r\ Pk\ sb1\ )\ )
\end{array}
$$

This operator offers a simpler alternative for specifying the deletes cascade loop over a relation, rule B7.2 (B) of the method. However, it cannot be applied if there is another foreign key with such property in the same relation, in which case the general rule should be applied.

5.5 The sorting operator

Now, the *SORT* operator is specified. It is used to present the results (output variables) of read-only operations in a specific order, as described in rule A2 of the method.

The *SORT* operator takes three parameters: a relation, an attribute, and a comparison operator (its type is ($B \rightarrow B \rightarrow BOOL$)) to compare elements of the type of the Attribute. Its result is a sequence formed by the elements of the relation sorted by the values of the attribute according to the comparison operator.

According to the predicate of the $SORT$ operator, presented below, the resulting sequence (seq) contains exactly the same elements (tuples) as the input relation (rel) in such an order that the cop comparison operator applied to the values of the given attribute (Att) of two consecutive tuples is always $true$.

$$
\begin{array}{l}
[A, B] \\
\hline
SORT : \mathbb{P}\, A \ \rightarrow \ (A \rightarrow B) \ \rightarrow \ (B \rightarrow B \rightarrow BOOL) \ \rightarrow \ \mathrm{seq}\, A \\
\hline
\forall\, rel : \mathbb{P}\, A;\ Att : (A \rightarrow B);\ cop : (B \rightarrow B \rightarrow BOOL);\ seq : \mathrm{seq}\, A \ \bullet \\
\qquad SORT\ rel\ Att\ cop \ = \ seq \ \ \Leftrightarrow \\
\qquad\qquad (\ rel \ = \ \mathrm{ran}\, seq \ \ \wedge \ \ \#rel \ = \ \#seq \ \ \wedge \\
\qquad\qquad\quad \forall\, i : 1 \,..\, (\#seq - 1) \ \bullet \\
\qquad\qquad\qquad cop \ \ (Att\ (seq\ i)) \ \ (Att\ (seq\ (i+1))) \ \ )
\end{array}
$$

5.6 Aggregate function operators

Now, the specification of the operators $COUNT$, MAX, MIN, SUM, and $AVER$ are presented. These are used in the specification of the aggregate functions as described in rule A3 of the method.

Once again, these operators are in fact syntactic sugar defined in terms of more general operators called $COUNTS$, $MAXMIN$, $SUMS$, and $AVERAGE$ respectively, which are presented below.

The first one, $COUNTS$, gives the number of tuples of a relation such that the value of the given attribute is not null. It takes three parameters: a value of type B, that represents the null value of this type, a relation, and an attribute of the relation.

$$
\begin{array}{l}
[A, B] \\
\hline
COUNTS : \ B \ \rightarrow \ \mathbb{P}\, A \ \rightarrow \ (A \rightarrow B) \ \rightarrow \ \mathbb{N} \\
\hline
\forall\, null : B;\ rel : \mathbb{P}\, A;\ Att : (A \rightarrow B) \ \bullet \\
\qquad COUNTS\ null\ rel\ Att \ = \ \#\ \{\ t : rel \ \mid \ Att\ t \neq null\ \}
\end{array}
$$

In order to simplify the use of $COUNTS$, the operator $COUNT$ is introduced below. Once again, there are in fact four definitions of $COUNT$ (overloaded) that, for each of the basic types, instantiate the operator $COUNTS$ by giving the corresponding constant as the first parameter.

$$COUNT\ [A] \ \widehat{=} \ COUNTS\ [A, \mathbb{Z}] \ \ NULLINT$$

$$COUNT\ [A] \ \widehat{=} \ COUNTS\ [A, \mathbb{N}] \ \ NULLNAT$$

$$COUNT\ [A] \ \widehat{=} \ COUNTS\ [A, REAL] \ \ NULLREAL$$

$$COUNT\ [A] \ \widehat{=} \ COUNTS\ [A, STRING] \ \ NULLSTR$$

The other operators can only be applied if the type of (the range of) the attribute is numeric, i.e., it is one of the following types: $\mathbb{Z}, \mathbb{N}$, or $REAL$.

The next one, *MAXMIN*, is used to calculate either the maximum or the minimum value of the attribute in the tuples of the relation, without considering the tuples where the value of the attribute is null. It takes four parameters: a value of type B, which represents the null value of this type, a comparison operator for values of this type, a relation, and an attribute of the relation.

$$
\begin{array}{l}
[A, B] \\
\hline
MAXMIN : B \rightarrow (B \rightarrow B \rightarrow BOOL) \rightarrow \mathbb{P}\, A \rightarrow (A \rightarrow B) \rightarrow B \\
\hline
\forall\, null : B;\ OP : (B \rightarrow B \rightarrow BOOL); \\
\qquad\qquad\qquad rel : \mathbb{P}\, A;\ Att : (A \rightarrow B);\ res : B \bullet \\
\quad MAXMIN\ null\ OP\ rel\ Att = res \Leftrightarrow \\
\qquad ((\exists\, t : rel \bullet Att\ t = res \wedge \\
\qquad\quad \neg\,(\exists\, t2 : rel \mid Att\ t2 \neq null \bullet OP\ (Att\ t2)\ res)) \vee \\
\qquad (\forall\, t : rel \bullet Att\ t = null \wedge res = null))
\end{array}
$$

According to the predicate of *MAXMIN*, the result *res* is such that it is the value of the attribute *Att* for at least one tuple of *rel* and, for no tuple of *rel*, the result of comparing *Att* to *res* using *OP* evaluates to *true*.

Note that according to the specification, when the value of the attribute is null in all tuples of *rel* (this includes the empty relation), the result of *MAXMIN* is null.

In order to simplify the use of *MAXMIN*, the operators *MAX* and *MIN* are introduced below. In fact, there are three definitions of each (overloaded) which instantiate the first two parameters of *MAXMIN* with the constants corresponding to one of the basic numeric types and the relevant comparison operator, respectively. For specifying *MAX*, $>$ is the appropriate comparison operator, whereas $<$ is used to specify *MIN*.

$$MAX\ [A] \mathrel{\widehat=} MAXMIN\ [A, \mathbb{Z}]\ NULLINT\ >$$

$$MAX\ [A] \mathrel{\widehat=} MAXMIN\ [A, \mathbb{N}]\ NULLNAT\ >$$

$$MAX\ [A] \mathrel{\widehat=} MAXMIN\ [A, REAL]\ NULLREAL\ >$$

$$MIN\ [A] \mathrel{\widehat=} MAXMIN\ [A, \mathbb{Z}]\ NULLINT\ <$$

$$MIN\ [A] \mathrel{\widehat=} MAXMIN\ [A, \mathbb{N}]\ NULLNAT\ <$$

$$MIN\ [A] \mathrel{\widehat=} MAXMIN\ [A, REAL]\ NULLREAL\ <$$

The *SUMS* operator is presented next and returns the sum of the values of the attribute in all tuples of the relation where it is not null. This operator takes four parameters: a value of type B, that represents the null value of this type, the sum operator for values of this type, a relation, and an attribute of the relation.

According to its predicate, the result is the sum ($+/$) of the elements of the sequence $seq2$ formed by (1) selecting the tuples of relation *rel* such that the value of attribute *Att* is not null, (2) transforming the result into a sequence ($sq1$), and (3) projecting it over attribute *Att*.

$$
\begin{array}{l}
[A, B] \\
\hline
SUMS : B \rightarrow (B \rightarrow B \rightarrow B) \rightarrow \mathbb{P}\,A \rightarrow (A \rightarrow B) \rightarrow B \\
\hline
\forall\, null : B;\; OP : (B \rightarrow B \rightarrow B);\; rel : \mathbb{P}\,A;\; Att : (A \rightarrow B) \bullet \\
\quad \exists\, r : \mathbb{P}\,A;\; sq1 : \operatorname{seq} A;\; sq2 : \operatorname{seq} B \bullet \\
\quad\quad (\; r = \{t : rel \mid Att\ t \neq null\} \wedge \\
\quad\quad\; r = \operatorname{ran} sq1 \wedge \#r = \#sq1 \wedge \\
\quad\quad\; sq2 = \{s : sq1 \bullet (\operatorname{dom} s \mapsto (Att(\operatorname{ran} s)))\} \wedge \\
\quad\quad\; SUMS\ null\ OP\ rel\ Att = +/\ OP\ sq2\ )
\end{array}
$$

The formal definition of the auxiliary operator $+/$ is presented below. As already mentioned, it returns the sum of the elements of a sequence.

$$
\begin{array}{l}
[A] \\
\hline
+/ : (A \rightarrow A \rightarrow A) \rightarrow \operatorname{seq} A \rightarrow A \\
\hline
\forall\, OP : (A \rightarrow A \rightarrow A) \bullet \\
\quad +/\ OP\ \langle\ \rangle = 0 \wedge \\
\quad \forall\, v : A \bullet +/\ OP\ \langle v \rangle = v \wedge \\
\quad \forall\, s1, s2 : \operatorname{seq} A \bullet \\
\quad\quad +/\ OP\ (s1 \frown s2) = OP\ (+/\ OP\ s1)\ (+/\ OP\ s2)
\end{array}
$$

Notice that (1) the predicate of $+/$ guarantees that, when *rel* is the empty relation, the result of *SUMS* is zero; and (2) the operator "+" is not used directly in the predicates of *SUMS* and $/+$ because of the generic type of its operands (instances of attribute *Att*).

Once again the use of *SUMS* is simplified by the specification of overloaded operators called *SUM* which instantiate the first two parameters of *SUMS* as follows:

$$SUM\ [A] \mathrel{\widehat{=}} SUMS\ [A, \mathbb{Z}]\ NULLINT\ +$$

$$SUM\ [A] \mathrel{\widehat{=}} SUMS\ [A, \mathbb{N}]\ NULLNAT\ +$$

$$SUM\ [A] \mathrel{\widehat{=}} SUMS\ [A, REAL]\ NULLREAL\ +$$

The last one, *AVERAGE*, gives the average value of attribute *Att* in the tuples of relation *rel*. Basically, the value returned is the result of *SUMS* divided by the result of *COUNTS*. However, if the attribute *Att* is null in all tuples of relation *rel* (and this includes the case of an empty relation), *AVERAGE* is explicitly defined to be zero.

$$
\begin{array}{l}
[A, B] \\
\hline
AVERAGE : B \rightarrow (B \rightarrow B \rightarrow B) \rightarrow \mathbb{P}\,A \rightarrow (A \rightarrow B) \rightarrow B \\
\hline
\forall\, null : B;\; OP : (B \rightarrow B \rightarrow B);\; rel : \mathbb{P}\,A;\; Att : (A \rightarrow B) \bullet \\
\quad AVERAGE\ null\ OP\ rel\ Att = \\
\quad\quad \mathbf{if}\ COUNTS\ null\ rel\ Att = 0 \\
\quad\quad \mathbf{then}\ 0 \\
\quad\quad \mathbf{else}\ (SUMS\ null\ OP\ rel\ Att\ /\ COUNTS\ null\ rel\ Att)
\end{array}
$$

Alternatively, a different version of $AVERAGE$ may be defined as the result of $SUMS$ divided by the number of tuples of the relation (#).

To conclude this section, the specification of the overloaded operators $AVER$ are presented. Yet again, these are syntactic sugar which instantiate the first two parameters of $AVERAGE$ and simplify its use.

$$AVER\ [A] \mathrel{\widehat{=}} AVERAGE\ [A, \mathbb{Z}]\ \ NULLINT\ \ +$$

$$AVER\ [A] \mathrel{\widehat{=}} AVERAGE\ [A, \mathbb{N}]\ \ NULLNAT\ \ +$$

$$AVER\ [A] \mathrel{\widehat{=}} AVERAGE\ [A, REAL]\ \ NULLREAL\ \ +$$

5.7 Composite attribute operators

Now, operators called $CA2$, $CA3$, etc. are defined and this will make it possible to use operators KEY_OF and FOR_KEY when the key of the relation is formed by two or more attributes. They are also used together with operator $SORT$ for sorting results based on more than one attribute.

Operator $CA2$ takes three arguments: the two attributes that will form the key and a tuple of the relation. Its result is the tuple formed by the values of the two attributes in the given tuple.

$$[A, B, C]$$
$$CA2 : (A \to B) \ \to\ (A \to C) \ \to\ A \ \to\ (B \times C)$$
$$\forall F : (A \to B);\ G : (A \to C);\ a : A \ \bullet$$
$$CA2\ F\ G\ a \ =\ (F\ a, G\ a)$$

The operator $CA2$ is used to specify two-attribute keys for relations like for example in $KEY\_OF\ \ rel1\ \ (CA2\ Att1\ Att2)$.

The expression in parenthesis returns a *function* with the first two parameters of $CA2$ instantiated and exemplifies the use of the operators as *higher-order functions*.

So, the result is a function that takes a parameter of type $Rel1$ (A) and returns a tuple of type $DOM1 \times DOM2$ $(B \times C)$, where $DOM1$ and $DOM2$ are the domains of $Att1$ and $Att2$ in type $Rel1$, respectively.

Operator $CA3$ may be used for three-attribute keys. Its definition is presented below and is similar to the definition of $CA2$.

$$[A, B, C, D]$$
$$CA3 : (A \to B) \ \to\ (A \to C) \ \to\ (A \to D) \ \to\ A \ \to\ (B \times C \times D)$$
$$\forall F : (A \to B);\ G : (A \to C);\ H : (A \to D);\ a : A \ \bullet$$
$$CA3\ F\ G\ H\ a \ =\ (F\ a, G\ a, H\ a)$$

Others called $CA4$, $CA5$, etc. can also be defined in a similar way, the main difference being the number of attributes in the arguments and, consequently, in the result.

The next one, $COP2$, is used together with operators $SORT$ and $CA2$ to specify sorting of results based on two attributes.

The $COP2$ operator takes 4 parameters: the two comparison operators for the types of each of the attributes, and the two pairs of values to be compared. Its result is a boolean, *true* if the pairs satisfy the comparison operator and *false* otherwise.

$$
\begin{array}{l}
[A, B] \\
\hline
COP2 : (A \to A \to BOOL) \to (B \to B \to BOOL) \to \\
\qquad\qquad\qquad\qquad (A \times B) \to (A \times B) \to BOOL \\
\hline
\forall\, OP1 : (A \to A \to BOOL);\ OP2 : (B \to B \to BOOL); \\
\qquad\qquad\qquad\qquad (a1, b1), (a2, b2) : A \times B \ \bullet \\
\qquad COP2\ OP1\ OP2\ (a1, b1)\ (a2, b2) \ \Leftrightarrow\ \textbf{if}\ \ a1 \neq a2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ OP1\ a1\ a2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ OP2\ b1\ b2
\end{array}
$$

According to the above specification, the result of comparing two pairs is the same of comparing their first components except when their first components are equal. In this case, the result is that of the comparison of their second components.

Sorting relation rel based on attributes $Att1$ and $Att2$ giving the sequence $seq!$ uses a *partial parametrization* of operators $CA2$ and $COP2$ as follows:

$$seq! \ = \ SORT \ \ rel \ \ (CA2\ Att1\ Att2) \ \ (COP2\ OP1\ OP2)$$

Similarly, operators $COP3$, $COP4$, etc. may also be written to be used together with $CA3$, $CA4$, etc. in the specification of sorts based on more than two attributes.

5.8 Foreign key transitive closure

The operator $FKTC$, presented below, is recursive and returns the foreign key transitive closure of a given set of keys, i.e., the set of primary keys of tuples which directly or indirectly reference any of the keys in the set sv by means of the foreign key. It was not mentioned in the description of the method but is potentially useful in the specification of some applications and will be used in the company database example (Chapter 6).

$$
\begin{array}{l}
[A, B] \\
\hline
FKTC : \mathbb{P}\, A \to (A \to B) \to (A \to B) \to \mathbb{P}\, B \to \mathbb{P}\, B \\
\hline
\forall\, rel : \mathbb{P}\, A;\ Pk, Fk : (A \to B);\ sv : \mathbb{P}\, B \ \bullet \\
\qquad \textbf{let}\ \ spk == \{\ t : rel \mid Fk\ t \in sv \ \bullet\ Pk\ t\ \} \ \bullet \\
\qquad\qquad FKTC\ rel\ Pk\ Fk\ sv \ = \\
\qquad\qquad\qquad\qquad \textbf{if}\ \ spk \subseteq sv \\
\qquad\qquad\qquad\qquad \textbf{then}\ sv \\
\qquad\qquad\qquad\qquad \textbf{else}\ \ FKTC\ rel\ Pk\ Fk\ (sv \cup spk)
\end{array}
$$

Basically, the operator $FKTC$ represents recursive applications of the foreign key (Fk) and primary key (Pk) attributes to a set of primary key values (sv). In each step, the set (spk) of primary keys of tuples that reference keys of sv will be added to the set of primary keys to be used in the following step. The recursion stops when there is no new key to be added ($spk \subseteq sv$).

It is worth emphasizing that the foreign key attribute must refer to tuples of the same relation and that the original set of keys will be included in the final result.

In addition, note that this operator is more general than it might seem at first sight. In theory, it could be used together with any function Fk of the same type ($A \rightarrow B$) of the primary key function Pk, and the foreign key function is just one example. For instance, this function could be the result of composing a number of other functions possibly involving other relations as well.

5.9 Conclusion

This chapter provided the formal definitions of all the operators used in the specification of relational databases and applications, and this completes the theoretical material about the method.

The next chapter presents the specification of a simple database example written according to the method introduced in Chapter 4.

Chapter 6

The company database example

In this chapter, the method for the specification of relational database applications as well as the guidelines on how to use it sensibly, presented in Chapter 4, are applied to the specification of a simple *Company Database* and its applications.

In this example, employees are hired by departments to work on one or more projects controlled by a given department for a certain number of hours. Departments must have managers whereas employees may or may not have supervisors. This is an updated version of the example presented in [115]. It adds a few constraints and modifies one of the transactions in order to address a few cases which were left out of the previous version.

In order to make the understanding of the method easier, the formal specification of the example is merged with informal comments which describe the specifications and make references to the rules of the method used in each step.

Although it would probably be fruitful to specify a real life system from a suitable company in order to evaluate the method more effectively, it was decided this would not be feasible within the time scale of a Ph.D.

The chosen example, however artificial and incomplete, involves the specification of typical database transactions and covers virtually all features of the method, which includes all kinds of database constraints, all possible database operations, and a number of advanced features. Therefore, it is complex enough to be used in the investigation of the reification step, addressed in Chapter 7. This example is also the base for building the prototype implementation of the mapping process described in Chapter 8.

The specification of the example is not split into a number of steps as suggested in the general guidelines (Section 4.5). Even so, the guidelines on how to write the first specification, given in Subsection 4.5.1, were followed. The previous version of the company database example was specified in three separate steps [117, 118, 119], strictly according to the guidelines.

This chapter is subdivided into six sections. The first one gives an informal description of the chosen transactions. The second presents the database structure affected by these transactions, which includes the corresponding ER diagram. The third describes all database constraints that are to be satisfied. The fourth specifies the relational database schema in Z according to the method. The fifth introduces the specification of the common basic operations. Finally, the sixth specifies the transactions, which includes the construction and simplification of the preconditions and error handling.

6.1 The chosen transactions

Following the first guideline (G1), a number of transactions was chosen and their brief descriptions are presented below. These were chosen because they are typical transactions of real life database systems and cover all the basic relational database operations.

Salary_dept: Calculate the total salary of department d, i.e., the sum of the salaries of all employees hired by the department.

Move_empls_proj: Move employees working on project $p1$ to work on project $p2$ and delete $p1$. If $p2$ does not exist, then it should be inserted first.

This transaction is to be used when a new project is initiated with employees from two or more existing projects.

Set_empls_dept_proj: Insert a set of employees *sempl* in department d and assign them to work on project p. If d does not exist, then it should be inserted first. If p does not exist, then it should also be inserted.

This transaction is to be used when a department hires a number of employees to work on a specific project.

Empl_supervised_sorted_salary: List all employees supervised directly or indirectly by employee e sorted by *Salary* in descending order.

Weighted_salary_proj: Calculate the average salary of employees working on project p weighted by the number of hours worked by each employee.

Fire_selected_empls: Fire all male employees whose salaries are greater than a certain limit *high_salary* and remove them from the projects they work on. Managers of departments satisfying these conditions cannot be fired but should be returned as a separate output.

Notice that the number of chosen transactions is rather small. Nevertheless, these transactions together with the chosen database constraints (section 6.3) are enough to provide the opportunity to use all the important aspects of the method.

6.2 The database structure

Following the second guideline (G2), the subset of the database structure affected by the chosen transactions and/or by the database constraints have been identified and the corresponding ER diagram drawn.

The resulting ER diagram, presented in figure 6.1, includes attributes *ENum*, *Sex*, and *Salary* of entity *Employee*; attributes *DNum* and *NEmp* of entity *Department*; attribute *PNum* of entity *Project*; and attribute *Hours* of relationship *Works_on*. It also includes relationships *Is_Supervisor_of*, *Is_Manager_of*, *Hires*, and *Controls*.

As usual, the primary keys of the entities and the cardinality constraints of the relationships are explicitly represented in the ER diagram.

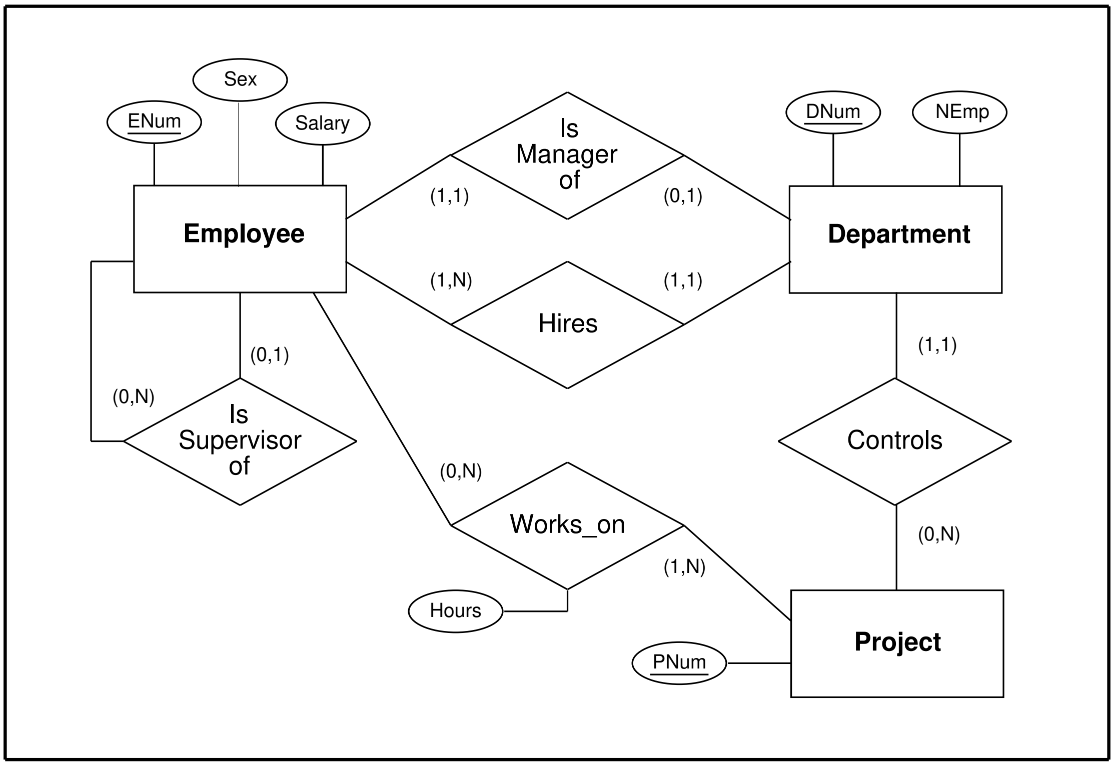


Figure 6.1: Entity-Relationship diagram

6.3 The database constraints

In this section a brief description of all database constraints is presented.

From the ER diagram, the primary keys of *Employee*, *Department*, and *Project* are *ENum*, *DNum*, and *PNum* respectively. Moreover, relationship *Works_on* is to be represented as relation *works* and its primary key is the tuple (*ENum*, *PNum*).

Also, all attributes of all relations are required, i.e., they cannot be null. The exception is attribute *SupENum* of relation *Employee*, which will represent the relationship *Is_Supervisor_of*.

In addition to these, the following constraints are to be satisfied:

- All employees must work at least four hours on each project they work on, which is an intra-relation static attribute constraint.

- All foreign key constraints, i.e., all foreign key attributes must reference the primary keys of existing tuples. The foreign key attributes are *SupENum* and *DNum* in employees, *ManENum* in departments, *DNum* in projects, and *ENum* and *PNum* in relation *works*.

  Also, for all foreign key attributes, the chosen compensating action for deletions of referenced tuples is *Restricted*, i.e., referenced tuples cannot be deleted. The exception is attribute *SupENum* (supervisor of employees), which has *Nullifies* as its compensating action.

For simplicity, the *Cascades* option is not used since it would not add any complexity to the example, though its use is briefly illustrated. Moreover, transaction *Fire_selected_empls* explicitly specifies that deletions of employees cascade based on the foreign key attribute *ENum* of relation *works*.

- All employees must work on at least one project, which is an inter-relations static constraint.
- Attribute *NEmp* of departments, which records the number of employees hired by the department, is in fact a derived attribute.
- The salaries of all employees can never decrease, which is a dynamic attribute constraint.

6.4 The relational database structure

Now, the relational database schema is specified in Z according to the method, which includes the definition of domains, relations, with their attributes, and the specification of the constraints.

At this level, seven domains are specified: one for the primary key of each entity, one for the number of employees of departments, and another for the number of hours of relationship *Works_on*, all represented as natural numbers; one for sex, defined by enumeration; and the other for the salary of employees, which is represented as a real number. Their formal definitions according to rule D1 of the method are:

$$ENUM == \mathbb{N}$$

$$DNUM == \mathbb{N}$$

$$PNUM == \mathbb{N}$$

$$SEX ::= Male \mid Female \mid NULLSEX$$

$$SALARY == REAL$$

$$NEMP == \mathbb{N}$$

$$HOURS == \mathbb{N}$$

The types of the tuples of each relation of the database being specified are defined below according to rule D2 in the description of the method. Note that 1:1 and 1:N relationships were represented as foreign key attributes.

$$\begin{aligned} EMPL \mathrel{\widehat{=}} [&ENum : ENUM \ ; \ Sex : SEX \ ; \ Salary : SALARY \ ; \\ &SupENum : ENUM \ ; \ DNum : DNUM \ ] \end{aligned}$$

$$DEPT \mathrel{\widehat{=}} [DNum : DNUM \ ; \ ManENum : ENUM \ ; \ NEmp : NEMP \ ]$$

$$PROJ \mathrel{\widehat{=}} [PNum : PNUM \ ; \ DNum : DNUM \ ]$$

$$WORK \mathrel{\widehat{=}} [ENum : ENUM \ ; \ PNum : PNUM \ ; \ Hours : HOURS \ ]$$

Each relation is then specified, using a separate schema, as a set of tuples of the appropriate type (rule D3) and including the specification of invariants.

Before that, the *REQUIRED* family of operators must be extended with an extra overloaded operator which will allow for its use together with domain *SEX*.

$$\mathit{REQUIRED}\,[A] \;\widehat{=}\; \mathit{NOT\_NULL}\,[A, \mathit{SEX}]\;\; \mathit{NULLSEX}$$

The employees relation and constraints are specified in the schema *Employee*, using the operators *REQUIRED* and *KEY_OF*, according to rules D3.1 and D3.2 of the method, respectively.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Employee}} \\
\hline
\mathit{empls} : \mathbb{P}\,\mathit{EMPL} \\
\hline
\mathit{REQUIRED}\ \mathit{empls}\ \mathit{ENum} \ \wedge \\
\mathit{REQUIRED}\ \mathit{empls}\ \mathit{Sex} \ \wedge \\
\mathit{REQUIRED}\ \mathit{empls}\ \mathit{Salary} \ \wedge \\
\mathit{REQUIRED}\ \mathit{empls}\ \mathit{DNum} \ \wedge \\
\mathit{KEY_OF}\ \mathit{empls}\ \mathit{ENum} \\
\hline
\end{array}$$

Schema *Depart*, presented below, is similar. Notice however that although attribute *NEmp* (the number of employees hired by the department) cannot be *null*, an explicit *REQUIRED* equation is not needed because *NEmp* is in fact a derived attribute.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Depart}} \\
\hline
\mathit{depts} : \mathbb{P}\,\mathit{DEPT} \\
\hline
\mathit{REQUIRED}\ \mathit{depts}\ \mathit{DNum} \ \wedge \\
\mathit{REQUIRED}\ \mathit{depts}\ \mathit{ManENum} \ \wedge \\
\mathit{KEY_OF}\ \mathit{depts}\ \mathit{DNum} \\
\hline
\end{array}$$

Schema *Project*, also similar, is presented below without any further explanation.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Project}} \\
\hline
\mathit{projs} : \mathbb{P}\,\mathit{PROJ} \\
\hline
\mathit{REQUIRED}\ \mathit{projs}\ \mathit{PNum} \ \wedge \\
\mathit{REQUIRED}\ \mathit{projs}\ \mathit{DNum} \ \wedge \\
\mathit{KEY_OF}\ \mathit{projs}\ \mathit{PNum} \\
\hline
\end{array}$$

Finally, the schema *Work* is also defined in a similar way but uses the *CA*2 operator to define a composite attribute key following rule A3 of the method. The last predicate specifies that employees must work at least four hours on each project they work on, which is a static attribute constraint.

Work

$$works : \mathbb{P}\ WORK$$

$$\begin{array}{l}
REQUIRED\ works\ ENum\ \wedge \\
REQUIRED\ works\ PNum\ \wedge \\
REQUIRED\ works\ Hours\ \wedge \\
KEY_OF\ works\ (CA2\ ENum\ PNum)\ \wedge \\
\forall\, w : works \bullet w.Hours \geq 4
\end{array}$$

Now, the database state schemas are specified. The first of them, *DB*, is specified according to rule D4 of the method. It aggregates all the relation extension schemas and specifies all the static constraints that involve more than one relation. This includes the foreign key constraints, which are specified using the operator *FOR_KEY* (rule D4.1), as well as other inter-relations static constraints (rule D4.2).

DB

$$\begin{array}{l}
Employee \\
Depart \\
Project \\
Work
\end{array}$$

$$\begin{array}{l}
FOR_KEY\ empls\ SupENum\ \ empls\ ENum\ \wedge \\
FOR_KEY\ depts\ ManENum\ empls\ ENum\ \wedge \\
FOR_KEY\ works\ \ ENum\ \ empls\ ENum\ \wedge \\
FOR_KEY\ empls\ \ DNum\ \ depts\ DNum\ \wedge \\
FOR_KEY\ projs\ \ DNum\ \ depts\ DNum\ \wedge \\
FOR_KEY\ works\ \ PNum\ \ projs\ PNum\ \wedge \\
\forall\, e : empls \bullet (\exists\, w : works \bullet w.ENum = e.ENum)\ \wedge \\
\forall\, d : depts \bullet d.NEmp = \#\{\, e : empls \mid e.DNum = d.DNum \,\}
\end{array}$$

Note that the last two equations of the predicate of the above schema specify that all employees must work on at least one project, and that the number of employees of departments (*NEmp*) is a derived attribute, respectively.

Schema ΔDB is then defined to be used in the specification of the update operations. It includes the database states before and after the operations (D5) and introduces a dynamic constraint (D5.1) which says that the salaries of employees cannot be decreased.

ΔDB

$$\begin{array}{l}
DB \\
DB'
\end{array}$$

$$\begin{array}{l}
\forall\, e' : empls';\ e : empls \bullet \\
\qquad (e'.ENum = e.ENum) \Rightarrow (e'.Salary \geq e.Salary)
\end{array}$$

For completeness, schema ΞDB, which will be used in the specification of the read-only operations, is specified according to rule D6.

$$\Xi DB \mathrel{\widehat=} [\ \Delta DB \mid \theta DB' = \theta DB\ ]$$

Finally, schema *Init_DB* defines the initial state of the database according to rule D7 of the method.

$$
\begin{array}{|l}
\textit{Init\_DB} \\
\hline
DB' \\
\hline
empls' = \{\} \wedge \\
depts' = \{\} \wedge \\
projs' = \{\} \wedge \\
works' = \{\} \\
\hline
\end{array}
$$

6.5 Common basic operations

Now, the common basic operations are specified as suggested in the fourth guideline for the first specification (G4, Subsection 4.5.1). These include insertion and deletion based on the primary keys of all relations and are always (or at least usually) needed, no matter what system is being specified. For this reason, they are specified before operations needed in specific transactions.

The specification of insertions to relation *empls* based on its primary key is presented below. It uses the set union ($\cup$) to add new tuples (input) to the relation, as prescribed by rule B6 of the method.

$$
\begin{array}{|l}
\textit{Insert\_empls} \\
\hline
\Delta DB \\
sempl? : \mathbb{P}\, EMPL \\
\Xi DB \setminus empls \\
\hline
empls' = empls \cup sempl? \\
\hline
\end{array}
$$

The specification of insertions to relations *depts*, *projs*, and *works* are similar to the insertion of employees and are presented below without any further explanation.

$$
\begin{array}{|l}
\textit{Insert\_depts} \\
\hline
\Delta DB \\
sdept? : \mathbb{P}\, DEPT \\
\Xi DB \setminus depts \\
\hline
depts' = depts \cup sdept? \\
\hline
\end{array}
$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Insert_projs}} \\
\Delta DB \\
sproj? : \mathbb{P}\, PROJ \\
\Xi DB \setminus projs \\
\hline
projs' \ = \ proj \cup sproj?
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Insert_works}} \\
\Delta DB \\
swork? : \mathbb{P}\, WORK \\
\Xi DB \setminus works \\
\hline
works' \ = \ works \cup swork?
\end{array}$$

Now, schemas to delete a set of tuples of relations based on their primary keys are specified according to rules B5 and B7. In fact there are two schemas for each relation: one without the expression $\Xi DB \setminus rel$, to be used in schema inclusions, and the other with the expression, to be used as a sub-transaction. The first pair of schemas, presented below, delete a set of tuples of *empls* based on its primary key *ENum*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Delete_empls_Pk}} \\
\Delta DB \\
se? : \mathbb{P}\, ENUM \\
\hline
\textbf{let}\ \ sempl\ == \ UPDATE\ empls\ SupENum\ se?\ NULLNAT\ \bullet \\
\qquad\qquad\qquad\qquad\qquad empls' \ = \ DELETE\ sempl\ ENum\ se?
\end{array}$$

$$Delete\_empls \ \widehat{=} \ Delete\_empls\_Pk \ \wedge \ \Xi DB \setminus empls$$

Notice that these delete schemas also specify what happens to tuples of other relations where there is a foreign key reference to deleted tuples - rules B7.1, B7.2, and B7.3 of the method. In the above schema, the *Nullifies* effect based on the foreign key *SupENum* is specified according to rule B7.3 and uses a **let** expression because this foreign key is in the same relation (*empls*). However, because *Restricted* was the compensating action chosen for violations of all the remaining foreign keys and is usually specified implicitly, no other equations are required.

Now, the foreign key *ManENum* in relation *depts* is going to be used as an example to illustrate the other options for the specification of the compensating actions. For instance, to specify *Restricted* explicitly and thus highlight this choice, the equation below could be included (redundantly) in the predicate of schema *Delete_empls_Pk*.

$$\forall\, d : depts' \ \bullet \ d.ManENum \notin se?$$

If *Cascades* had been chosen, the equation below would have been added to the predicate of schema *Delete_empls_Pk*. It is assumed that *ManENum* is not part of a cycle of foreign keys that cascade for deletes - rule B7.2 (A).

$$\mathbf{let}\ \ sdp\ ==\ \{\ d : depts\ \mid\ d.ManENum \in se?\ \bullet\ d.DNum\ \}\ \bullet$$
$$Delete\_depts\_Pk\ [sdp\ /\ sd?]$$

Finally, except when *Restricted* is chosen, the specification of schema *Delete_empls* would have to reflect the fact that relation *depts* would also change as follows:

$$Delete\_empls\ \widehat{=}\ Delete\_empls\_Pk\ \ \wedge\ \ \Xi DB \setminus (empls, depts)$$

The pairs of Schemas *Delete_depts_Pk* (and *Delete_depts*) and *Delete_projs_Pk* (and *Delete_projs*) specify deletions by the primary key in relations *depts* and *projs* respectively. These are similar to *Delete_empls_Pk* (and *Delete_empls*), though no explicit compensating action is needed, and are presented without any further explanation.

Delete_depts_Pk

$$\Delta DB$$
$$sd? : \mathbb{P}\, DNUM$$

$$depts' \ =\ DELETE\ depts\ DNum\ sd?$$

$$Delete\_depts\ \widehat{=}\ Delete\_depts\_Pk\ \ \wedge\ \ \Xi DB \setminus depts$$

Delete_projs_Pk

$$\Delta DB$$
$$sp? : \mathbb{P}\, PNUM$$

$$projs' \ =\ DELETE\ projs\ PNum\ sp?$$

$$Delete\_projs\ \widehat{=}\ Delete\_projs\_Pk\ \ \wedge\ \ \Xi DB \setminus projs$$

The last of these pairs or schemas, *Delete_works_Pk* and *Delete_works*, delete a set of tuples of relation *works* and is also similar. The only difference is the fact that relation *works* has a composed attribute primary key.

Delete_works_Pk

$$\Delta DB$$
$$sw? : \mathbb{P}\ (ENUM \times PNUM)$$

$$works' \ =\ DELETE\ works\ (CA2\ ENum\ PNum)\ sw?$$

$$Delete\_works\ \widehat{=}\ Delete\_works\_Pk\ \ \wedge\ \ \Xi DB \setminus works$$

6.6 The transactions and their basic operations

In this section, the chosen transactions are described based on their basic operations (sub-transactions) and then extended to capture error handling.

Basically, for each chosen transaction, (1) the identified basic operations are enumerated, (2) the ones that were not used before are formally specified, (3) the specification of the correct behaviour of the transaction is written, (4) its precondition is specified and simplified[1] (5) the corresponding error schema is written, and (6) the total transaction, which captures the error handling, is specified.

A simple convention for naming variables, adopted throughout the specification, is described below.

- Longer names (at least four letters) are used to refer to relations and tuple variables, whereas shorter names (usually one or two letter names) frequently refer to key variables and local variables.
- When necessary, numbers are used to make two or more variables distinct. For example, $dept$, $dept1$, etc. are used for department tuples and d, $d1$, etc. as well as dp, $dp1$, etc. for department keys or local variables.
- Additionally, a prefix s is used in the names of variables that represent sets while l is used in the ones that represent sequences (lists).
- Finally, Z's convention of adding the suffixes ? and ! to the names of input and output variables is also respected.

6.6.1 Transaction *Salary_dept*

The first of the transactions to be specified is *Salary_dept* which gives the total salary $tot\_sal!$ of employees hired by department $d?$.

Its identified basic operations are (1) *Empls_of_dept*, which returns all employees $sempl!$ hired by department $d?$, and (2) *Sum_Salary_empls*, which returns the sum of the salaries $tot\_sal!$ of a given set of employees $sempl?$.

The first of these operations, *Empls_of_dept*, is now presented. It is a simple select operation specified according to rule B2. The first equation of its predicate states that $d?$ must refer to the primary key $DNum$ of a valid department.

Empls_of_dept

ΞDB

$d? : DNUM$
$sempl! : \mathbb{P}\, EMPL$

$\exists\, dp : depts \bullet dp.DNum = d? \;\wedge$

$sempl! \;=\; \{\, e : empls \mid e.DNum = d? \,\}$

[1] Actually, the simplification of the precondition of most transactions is omitted: only the results are presented. A couple of simplifications are presented to illustrate the process though.

The operation *Sum_salary_empls*, which returns the sum of the salaries *tot_sal*! of the set of employees *sempl*?, exemplifies the use of the *SUM* aggregate function operator according to rule A3 of the method.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Sum_salary_empls}} \\
\Xi DB \\
\\
sempl? : \mathbb{P}\, EMPL \\
tot_sal! : SALARY \\
\hline
tot_sal! \;=\; SUM\ sempl?\ Salary \\
\hline
\end{array}$$

The correct behaviour of the transaction is then described in terms of its basic operations. Notice that, according to rule A1 of the method, the version of the piping operator (>>) used here allows for the output *and* primed state variables (*all results*) of the first schema to be matched against the input *and* unprimed state variables of the second schema, respectively; whereas the standard Z piping operator does *not* match the state variables of the two schemas.

$$Salary\_dept\_Ok \;\widehat{=}\; Empls\_of\_dept \;>>\; Sum\_salary\_empls$$

To give a better idea of what the schema *Salary_dept_Ok* means, the expanded schema resulting from its definition is presented below. Note that the new schema contains all declarations of the piped schemas (*Empls_of_dept* and *Sum_salary_empls*) except for the matched components.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Salary_dept_Ok}} \\
\Delta DB \\
\\
d? : DNUM \\
tot_sal! : SALARY \\
\hline
\exists\, DB''; \ sempl!? : \mathbb{P}\, EMPL \;\bullet \\
\qquad\quad (\ Empls_of_dept\ [DB'' \,/\, DB',\ sempl!? \,/\, sempl!] \;\wedge \\
\qquad\qquad Sum_salary_empls\ [DB'' \,/\, DB,\ sempl!? \,/\, sempl?]\ ) \\
\hline
\end{array}$$

Expanding the predicate of the above schema to replace the schema inclusions with their corresponding components, and renaming the piped variable *sempl*!? to *se* gives us the following predicate:

$$\begin{array}{l}
\exists\, DB''; \ se : \mathbb{P}\, EMPL \;\bullet \\
\qquad\quad (\ \theta DB'' \;=\; \theta DB \;\wedge \\
\qquad\qquad \exists\, dp : depts \;\bullet\; dp.DNum = d? \;\wedge \\
\qquad\qquad se = \{\ e : empls \;\mid\; e.DNum = d?\ \} \;\wedge \\
\qquad\qquad \theta DB' \;=\; \theta DB'' \;\wedge \\
\qquad\qquad tot_sal! \;=\; SUM\ se\ Salary\ )
\end{array}$$

Now, because $\exists\, x \bullet (x = y \land P(x))$ is equivalent to $P(y)$, DB'' can be removed from the existential quantifier. Also, the inner existential quantifier is independent of the outer one and thus the former can be put outside the latter. These simplify the predicate to:

$$
\begin{array}{l}
\theta DB' \;=\; \theta DB \;\land \\
\exists\, dp : depts \;\bullet\; dp.DNum = d? \;\land \\
\exists\, se : \mathbb{P}\, EMPL \;\bullet \\
\qquad\quad (\; se = \{\, e : empls \;\mid\; e.DNum = d? \,\} \;\land \\
\qquad\quad\;\; tot\_sal! \;=\; SUM\ se\ Salary \;)
\end{array}
$$

Finally, the first equation can be expressed in the declaration part of the schema. Also, the last part of the above predicate can be more clearly expressed using a *let* expression. The simplified schema *Salary_dept_Ok* is then presented below.

$$
\begin{array}{|l}
\hline
\textit{Salary\_dept\_Ok} \\
\Xi DB \\
d? : DNUM \\
tot\_sal! : SALARY \\
\hline
\exists\, dp : depts \;\bullet\; dp.DNum = d? \;\land \\
\textbf{let}\;\; se \;==\; \{\, e : empls \;\mid\; e.DNum = d? \,\} \;\bullet \\
\qquad\qquad\qquad\qquad tot\_sal! \;=\; SUM\ se\ Salary \\
\hline
\end{array}
$$

In the above example the resulting schema is simple enough to be written without being split into sub-transactions. Even though it is up to the user to decide how much the transactions are to be split, splitting them will in general contribute to a greater reuse and to a better understanding of the specifications.

The next step in the specification is then to write and simplify the precondition of the above transaction. Basically, the predicate of *pre Salary_dept_Ok* is given by

$$\exists\, DB';\ tot\_sal! : SALARY \;\bullet\; Salary\_dept\_Ok$$

which is equivalent to the following:

$$
\begin{array}{l}
\exists\, DB';\ tot\_sal! : SALARY \;\bullet \\
\qquad\quad \theta DB' \;=\; \theta DB \;\land \\
\qquad\quad \exists\, dp : depts \;\bullet\; dp.DNum = d? \;\land \\
\qquad\quad \textbf{let}\;\; se \;==\; \{\, e : empls \;\mid\; e.DNum = d? \,\} \;\bullet \\
\qquad\qquad\qquad\qquad\qquad tot\_sal! \;=\; SUM\ se\ Salary
\end{array}
$$

Because $\exists\, x \bullet (x = y \land P(x))$ is equivalent to $P(y)$, and *let* expressions are in fact syntactic sugared existential quantifiers, the variable *se* can be removed. In addition, the inner existential quantifier is independent of the outer one and thus the former can be put outside the latter. These simplify the predicate to:

$$\exists\, dp : depts \;\bullet\; dp.DNum = d? \;\wedge$$
$$\begin{aligned}&\exists\, DB';\ tot\_sal! : SALARY \;\bullet\\ &\qquad \theta DB' = \theta DB \;\wedge\\ &\qquad tot\_sal! = SUM\ \{\, e : empls \mid e.DNum = d?\,\}\ Salary\end{aligned}$$

Moreover, the second existential quantifier can be removed because $\exists\, x \bullet x = y$ is always *true*, since it can be seen as $\exists\, x \bullet (x = y \wedge true)$, which is equivalent to *true*. Consequently, the simplified precondition of the above transaction is simply:

$$\exists\, dp : depts \;\bullet\; dp.DNum = d?$$

Now, the corresponding error schema *Salary_dept_Error*, presented below, is specified according to rule E2 of the method.

$$\begin{array}{|l}\hline \textit{Salary\_dept\_Error}\\ \Xi DB\\ result! : STRING\\ d? : DNUM\\ \hline (\,\neg\,(\exists\, dp : depts \;\bullet\; dp.DNum = d?) \;\wedge\\ \qquad result! = \text{``Invalid department number''}\,)\\ \hline\end{array}$$

Finally, the total transaction *Salary_dept* is specified in terms of *Salary_dept_Ok*, *Ok*, and *Salary_dept_Error*, according to rule E3 of the method.

$$Salary\_dept \;\widehat{=}\; (Salary\_dept\_Ok \;\wedge\; Ok) \;\vee\; Salary\_dept\_Error$$

6.6.2 Transaction *Move_empls_proj*

The second transaction, *Move_empls_proj*, is described as follows: *"Move employees working on project p1? to project p2? and delete p1?; If p2? does not exist, then it should be inserted first"*.

Its identified basic operations are: (1) *Insert_projs*, that inserts a set of new projects *sproj*?; (2) *Insert_proj_opt*, that uses *Insert_projs* to insert project *proj*? with primary key *p*? if it does not exist (optional insert); (3) *Works_on_proj*, that gets from relation *works* the set of tuples *swork*! referring to a given project *p*?; (4) *Change_works_proj*, that changes the project a set of employees work on, i.e., changes to *p*? the project attribute *PNum* of a set of tuples *swork*? of relation *works*; and (5) *Delete_projs*, that deletes projects based on a set of primary keys *sp*?.

A schema for the insertion of a set of new projects was already specified. The optional insertion of projects is then specified in terms of the general insert schema.

$$
\begin{array}{|l}
\hline
\textit{Insert\_proj\_opt} \\
\Delta DB \\
p? : PNUM \\
proj? : PROJ \\
\hline
\textbf{if } \neg (\exists\, pj : projs \bullet pj.PNum = p?) \\
\textbf{then } (\ proj?.PNum = p? \ \wedge \\
\qquad\quad Insert\_projs\ [\{proj?\} \ / \ sproj?]\ ) \\
\textbf{else } \ \Xi DB \\
\hline
\end{array}
$$

Schema *Works_on_proj* specifies a simple select operation that gets from relation *works* the set of tuples $swork!$ referring to a given project $p?$. Its first predicate states that $p?$ must be the primary key of an existing project.

$$
\begin{array}{|l}
\hline
\textit{Works\_on\_proj} \\
\Xi DB \\
p? : PNUM \\
swork! : \mathbb{P}\ WORK \\
\hline
\exists\, pj : projs \bullet pj.PNum = p? \ \wedge \\
swork! \ = \ \{\ w : works \ \mid \ w.PNum = p?\ \} \\
\hline
\end{array}
$$

Schema *Change_works_proj*, presented below, changes the project a set of employees work on, i.e., changes to $p?$ the project attribute *PNum* of a set of tuples $swork?$ of relation *works*. It exemplifies the specification of updates of tuples, according to rules B5 and B9 of the method.

$$
\begin{array}{|l}
\hline
\textit{Change\_works\_proj} \\
\Delta DB \\
p? : PNUM \\
swork? : \mathbb{P}\ WORK \\
\Xi DB \setminus works \\
\hline
works' \ = \ \{\ w : works \bullet \textbf{if } \ w \in swork? \\
\qquad\qquad\qquad\qquad\quad \textbf{then } w \setminus (PNum = p?) \\
\qquad\qquad\qquad\qquad\quad \textbf{else } \ w\ \} \\
\hline
\end{array}
$$

The last of its sub-transactions, *Delete_projs*, deletes a set of tuples of relation *projs* based on its primary key *PNum* and has already been specified.

Schema *Move_empls_proj_Ok*, which describes the correct behaviour of transaction *Move_empls_proj*, is now specified in terms of its basic operations.

$$
\begin{array}{l}
Move\_empls\_proj\_Ok \ \widehat{=} \\
\qquad Insert\_proj\_opt\ [p2?\ /\ p?] \quad >> \\
\qquad\qquad Works\_on\_proj\ [p1?\ /\ p?] \quad >> \\
\qquad\qquad\qquad Change\_works\_proj\ [p2?\ /\ p?] \quad >> \\
\qquad\qquad\qquad\qquad Delete\_projs\ [\{p1?\}\ /\ sp?]
\end{array}
$$

To give a better idea of what schema *Move_empls_proj_Ok* means, the simplified version of the corresponding expanded schema is presented below, before the presentation of its precondition. The expansion of the above schema expression as well as the simplification of the resulting schema are omitted here since the process has already been illustrated.

$$
\begin{array}{|l}
\hline
Move\_empls\_proj\_Ok \\
\Delta DB \\
p1?, p2? : PNUM \\
proj? : PROJ \\
\Xi DB \setminus (projs, works) \\
\hline
\exists\, pj : projs \ \bullet\ pj.PNum = p1? \ \wedge \\
projs' \ = \ \textbf{if}\ \neg\,(\exists\, pj : projs \ \bullet\ pj.PNum = p2?) \\
\qquad\qquad \textbf{then}\ (\ proj?.PNum = p2? \ \wedge \\
\qquad\qquad\qquad DELETE\ (projs \cup proj?)\ PNum\ p1?\ ) \\
\qquad\qquad \textbf{else}\ \ DELETE\ projs\ PNum\ p1? \ \wedge \\
works' \ = \ \{\ w : works \ \bullet\ \textbf{if}\ \ w.PNum = p1? \\
\qquad\qquad\qquad\qquad \textbf{then}\ w \setminus (PNum = p2?) \\
\qquad\qquad\qquad\qquad \textbf{else}\ \ w\ \} \\
\hline
\end{array}
$$

Now, the precondition of the above schema is introduced. Although the simplification process is similar to that of the previous transaction, it is much more extensive since *Move_empls_proj* changes the contents of the database and this involves the validation of all database constraints. For this reason, the simplification process, omitted here, is introduced in Appendix A. The simplified precondition is presented below.

$$
\begin{array}{l}
p2? \neq NULLNAT \ \wedge \\
\exists\, pj : projs \ \bullet\ pj.PNum = p1? \ \wedge \\
\neg\,(\exists\, pj : projs \ \bullet\ pj.PNum = p2?) \ \Rightarrow \\
\qquad (\ proj?.PNum = p2? \ \wedge \\
\qquad\ \ proj?.DNum \neq NULLNAT \ \wedge \\
\qquad\ \ \exists\, dp : depts \ \bullet\ dp.DNum = proj?.DNum\ ) \ \wedge \\
\neg\,(\ \exists\, w1, w2 : works \ \bullet\ w1.ENum = w2.ENum \ \wedge \\
\qquad\qquad\qquad w1.PNum = p1? \ \wedge\ w2.PNum = p2?\ )
\end{array}
$$

The corresponding error schema *Move_empls_proj_Error* is then presented below according to rule E2 of the method.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{Move\_empls\_proj\_Error} \\
\Xi DB \\
result! : STRING \\
p1?, p2? : PNUM \\
proj? : PROJ \\
\hline
(\ p2? = NULLNAT \ \wedge \\
\qquad\qquad result! = \text{``Project number p2? cannot be null''}\ )\ \vee \\
(\ \neg\ (\exists\, pj : projs \ \bullet\ pj.PNum = p1?)\ \wedge \\
\qquad\qquad result! = \text{``Invalid project Number p1?''}\ )\ \vee \\
(\ \neg\ (\exists\, pj : projs \ \bullet\ pj.PNum = p2?)\ \wedge \\
\qquad (\ (\ proj?.PNum \neq p2? \ \wedge \\
\qquad\qquad result! = \text{``Project number of proj? must be p2?''}\ )\ \vee \\
\qquad (\ proj?.DNum = NULLNAT \ \wedge \\
\qquad\qquad result! = \text{``Attribute DNum of proj? cannot be null''}\ )\ \vee \\
\qquad (\ \neg\ (\exists\, dp : depts \ \bullet\ dp.DNum = proj?.DNum)\ \wedge \\
\qquad\qquad result! = \text{``Invalid department number in proj?''}\ )\ )\ )\ \vee \\
(\ (\ \exists\, w1, w2 : works \ \bullet\ w1.ENum = w2.ENum \ \wedge \\
\qquad\qquad\qquad w1.PNum = p1? \ \wedge \\
\qquad\qquad\qquad w2.PNum = p2?\ )\ \wedge \\
\qquad\qquad result! = \text{``Violation of works' primary key''}\ ) \\
\hline
\end{array}
$$

Again, the total transaction *Move_empls_proj* is specified according to rule E3.

$$
\begin{aligned}
& Move\_empls\_proj \ \widehat{=} \\
& \qquad (Move\_empls\_proj\_Ok \ \wedge\ Ok) \ \vee\ Move\_empls\_proj\_Error
\end{aligned}
$$

6.6.3 Transaction *Set_empls_dept_proj*

The third transaction is *Set_empls_dept_proj*. It is described as: *"Insert a set of employees sempl? in department d? and assign them to work on project p?; If d? does not exist it should be inserted first; If p? does not exist it should also be inserted first"*.

Its identified basic operations are: (1) *Insert_depts*, that inserts a set of new departments *sdept?*; (2) *Insert_dept_opt*, that uses *Insert_depts* to insert department *dept?* with primary key *d?* if it does not exist (optional insert); (3) *Insert_projs*, that inserts a set of new projects *sproj?*; (4) *Insert_proj_opt*, that inserts a project if it does not exist; (5) *Insert_empls*, that inserts a set of new employees *sempl?*; and (6) *Insert_works*, that inserts a set of tuples *swork?* in relation *works*.

Except for the optional insertion of a department, *Insert_dept_opt*, all the above sub-transactions have already been specified. That is presented below and is similar to the optional insertion of projects specified before.

$$
\begin{array}{|l}
\hline
\textit{Insert\_dept\_opt} \\
\Delta DB \\
\\
d? : DNUM \\
dept? : DEPT \\
\hline
\textbf{if} \;\; \neg\,(\exists\, dp : depts \;\bullet\; dp.DNum = d?) \\
\textbf{then} \;\; (\; dept?.DNum = d? \;\wedge \\
\qquad\qquad Insert\_depts\;[\{dept?\} \,/\, sdept?]\;) \\
\textbf{else} \;\; \Xi DB \\
\hline
\end{array}
$$

The correct behaviour of transaction *Set_empls_dept_proj* is then specified in terms of its basic operations as usual, but it includes the specification of extra constraints which involve variables of more than one subtransaction, as described in rule A1 of the method.

$$
\begin{array}{l}
Set\_empls\_dept\_proj\_Ok \;\widehat{=} \\
\qquad (\; Insert\_dept\_opt \;\; >> \\
\qquad\qquad Insert\_proj\_opt \;\; >> \\
\qquad\qquad\quad Insert\_empls \;\; >> \\
\qquad\qquad\qquad Insert\_works \;\; | \\
\qquad\qquad (\forall\, em : sempl? \;\bullet\; em.DNum = d?) \;\; \wedge \\
\qquad\qquad (\forall\, wk : swork? \;\bullet\; wk.PNum = p?) \;\; \wedge \\
\qquad\qquad \{\; em : sempl? \bullet em.ENum \;\} \;=\; \{\; wk : swork? \bullet wk.ENum \;\}\;)
\end{array}
$$

The simplified precondition of the above transaction is presented below. The details of the simplification are omitted.

$$
\begin{array}{l}
d? \neq NULLNAT \;\; \wedge \\
\\
\neg\,(\exists\, dp : depts \;\bullet\; dp.DNum = d?) \;\Rightarrow \\
\qquad (\; dept?.DNum = d? \;\wedge \\
\qquad\;\; dept?.ManENum \neq NULLNAT \;\;\wedge \\
\qquad\;\; \exists\, em : (empls \cup sempl?) \;\bullet\; em.ENum = Dept?.ManENum \;)\;\;\wedge \\
\\
p? \neq NULLNAT \;\; \wedge \\
\\
\neg\,(\exists\, pj : projs \;\bullet\; pj.PNum = p?) \;\Rightarrow \\
\qquad (\; proj?.PNum = p? \;\wedge \\
\qquad\;\; proj?.DNum \neq NULLNAT \;\;\wedge \\
\\
\qquad\;\; (\; proj?.DNum = d? \;\vee \\
\qquad\;\;\; \exists\, dp : depts \;\bullet\; dp.DNum = proj?.DNum \;)\;)\;\;\wedge
\end{array}
$$

$$
\begin{array}{l}
(\forall em : sempl? \bullet\ em.ENum \neq NULLNAT\ \wedge \\
\quad em.Sex \neq NULLSEX\ \wedge \\
\quad em.Salary \neq NULLREAL\ \wedge \\
\quad em.DNum = d?\ \wedge \\
\quad \neg\,(\exists\, em2 : empls \bullet\ em2.ENum = em.ENum)\ \wedge \\
\quad (\ em.SupENum = NULLNAT\ \vee \\
\qquad \exists\, em2 : (empls \cup sempl?) \bullet \\
\qquad\qquad em2.ENum = em.SupENum\ )\ )\ \wedge
\end{array}
$$

$$(\forall wk : swork? \bullet\ wk.Hours \geq 4\ \wedge\ wk.PNum = p?)\ \wedge$$

$$\{\ em : sempl? \bullet\ em.ENum\ \} = \{\ wk : swork? \bullet\ wk.ENum\ \}$$

Notice that, because transaction $Set\_empls\_dept\_proj$ has many preconditions to be satisfied, the corresponding error schema is split into four smaller schemas, based on the relations affected by each possible error (as suggested in rule E2), in order to make its understanding easier. The first of these schemas, $Set\_empls\_dept\_proj\_Error1$, covers all errors regarding relation $depts$ and is presented below.

$$
\begin{array}{|l}
\hline
Set\_empls\_dept\_proj\_Error1 \\
\Xi DB \\
result! : STRING \\
d? : DNUM; \\
dept? : DEPT \\
p? : PNUM; \\
proj? : PROJ \\
sempl? : \mathbb{P}\, EMPL \\
swork? : \mathbb{P}\, WORK \\
\hline
(\ d? = NULLNAT\ \wedge \\
\qquad result! = \text{“Department number cannot be null”}\ )\ \vee \\
(\ \neg\,(\exists\, dp : depts \bullet\ dp.DNum = d?)\ \wedge \\
\quad (\ (\ dept?.DNum \neq d?\ \wedge \\
\qquad result! = \text{“Department number of dept? must be d?”}\ )\ \vee \\
\quad\ (\ dept?.ManENum = NULLNAT\ \wedge \\
\qquad result! = \text{“Manager of department cannot be null”}\ )\ \vee \\
\quad\ (\ \neg\,(\exists\, em : (empls \cup sempl?) \bullet \\
\qquad\qquad em.ENum = dept?.ManENum)\ \wedge \\
\qquad result! = \text{“Invalid manager number in dept?”}\ )\ )\ ) \\
\hline
\end{array}
$$

The next error schema, $Set\_empls\_dept\_proj\_Error2$, covers all errors involving relation $projs$ and is presented below.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;Set\_empls\_dept\_proj\_Error2\;}}\\
\Xi DB\\
result! : STRING\\
d? : DNUM; \quad dept? : DEPT\\
p? : PNUM; \quad proj? : PROJ\\
sempl? : \mathbb{P}\, EMPL; \quad swork? : \mathbb{P}\, WORK\\
\hline
(\ p? = NULLNAT \ \wedge\ result! = \text{“Project number cannot be null”}\ ) \ \vee\\
(\ \neg\, (\exists\, pj : projs \bullet pj.PNum = p?) \ \wedge\\
\qquad (\ (\ proj?.PNum \neq p? \ \wedge\\
\qquad\qquad result! = \text{“Project number of proj? must be p?”}\ ) \ \vee\\
\qquad\ (\ proj?.DNum = NULLNAT \ \wedge\\
\qquad\qquad result! = \text{“Attribute DNum of proj? cannot be null”}\ ) \ \vee\\
\qquad\ (\ proj?.DNum \neq d? \ \wedge\\
\qquad\quad \neg\, (\exists\, dp : depts \bullet dp.DNum = proj?.DNum) \ \wedge\\
\qquad\qquad result! = \text{“Invalid department number in proj?”}\ )\ )\ )\\
\hline
\end{array}
$$

Similarly, schema *Set_empls_dept_proj_Error3* covers all errors regarding relation *empls* and is presented below.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;Set\_empls\_dept\_proj\_Error3\;}}\\
\Xi DB\\
result! : STRING\\
d? : DNUM; \quad dept? : DEPT\\
p? : PNUM; \quad proj? : PROJ\\
sempl? : \mathbb{P}\, EMPL; \quad swork? : \mathbb{P}\, WORK\\
\hline
(\ (\exists\, em : sempl? \bullet em.ENum = NULLNAT) \ \wedge\\
\qquad\qquad result! = \text{“Employee numbers cannot be null”}\ ) \ \vee\\
(\ (\exists\, em : sempl? \bullet em.Sex = NULLSEX) \ \wedge\\
\qquad\qquad result! = \text{“Sex of employees cannot be null”}\ ) \ \vee\\
(\ (\exists\, em : sempl? \bullet em.Salary = NULLREAL \ \wedge\\
\qquad\qquad result! = \text{“Salary of employees cannot be null”}\ ) \ \vee\\
(\ (\exists\, em : sempl? \bullet em.DNum \neq d?) \ \wedge\\
\qquad\qquad result! = \text{“Department of employees must be d?”}\ ) \ \vee\\
(\ (\exists\, em : sempl? \bullet \exists\, em2 : empls \bullet em2.ENum = em.ENum) \ \wedge\\
\qquad\qquad result! = \text{“Violation of employees primary key”}\ ) \ \vee\\
(\ (\exists\, em : sempl? \bullet em.SupENum \neq NULLNAT \ \wedge\\
\qquad \neg\, (\exists\, em2 : (empls \cup sempl?) \bullet em2.ENum = em.SupENum)\ ) \ \wedge\\
\qquad\qquad result! = \text{“Violation of SupENum foreign key”}\ )\\
\hline
\end{array}
$$

The last error schema, *Set_empls_dept_proj_Error4*, is presented below and covers the remaining possible errors, which are the ones involving relation *works*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;Set\_empls\_dept\_proj\_Error4\qquad\qquad\qquad\qquad\qquad\qquad\qquad}}\\
\Xi DB\\
result! : STRING\\
d? : DNUM;\\
dept? : DEPT\\
p? : PNUM;\\
proj? : PROJ\\
sempl? : \mathbb{P}\, EMPL\\
swork? : \mathbb{P}\, WORK\\
\hline
(\ (\exists\, wk : swork? \bullet wk.Hours < 4)\ \land\\
\qquad\qquad result! = \text{“}Employees\ must\ work\ at\ least\ 4\ hours\text{”}\ )\ \lor\\
(\ (\exists\, wk : swork? \bullet wk.PNum \neq p?)\ \land\\
\qquad\qquad result! = \text{“}New\ employees\ must\ work\ on\ project\ p?\text{”}\ )\ \lor\\
(\ \{\ em : sempl? \bullet em.ENum\ \} \neq \{\ wk : swork? \bullet wk.ENum\ \}\ \land\\
\qquad result! = \text{“}sempl?\ and\ swork?\ must\ refer\ to\ same\ employees\text{”}\ )\\
\hline
\end{array}
$$

Notice that, as suggested in rule E2, all input variables of the transaction were declared in all four error schemas, even though some of the variables were not used in all error schemas. The reason for declaring them all is to avoid having to find out which variables are needed in each schema.

Finally, the total transaction *Set_empls_dept_proj* is specified according to rule E3 of the method in the usual way except for the fact that there are four error schemas.

$$
\begin{array}{l}
Set\_empls\_dept\_proj\ \widehat{=}\\
\qquad\qquad (Set\_empls\_dept\_proj\_Ok\ \land\ Ok)\ \lor\\
\qquad\qquad\qquad Set\_empls\_dept\_proj\_Error1\ \lor\\
\qquad\qquad\qquad Set\_empls\_dept\_proj\_Error2\ \lor\\
\qquad\qquad\qquad Set\_empls\_dept\_proj\_Error3\ \lor\\
\qquad\qquad\qquad Set\_empls\_dept\_proj\_Error4
\end{array}
$$

6.6.4 Transaction *Empl_supervised_sorted_salary*

The next transaction to be specified is *Empl_supervised_sorted_salary*, which returns the list *lempl!* of all employees supervised directly or indirectly by employee *e?*, sorted by attribute *Salary* in descending order.

Its identified sub-transactions are: (1) *Empls_supervised*, that returns all employees *sempl!* supervised directly or indirectly by any employee whose primary key is in the set of keys *se?*; and (2) *Empls_sorted_salary*, that orders a set of employees *sempl?* by salary in descending order and returns the list *lempl!*.

The first of these sub-transactions is specified below. It uses the operator $FKTC$, presented in Section 5.8, to get what was called the foreign key transitive closure of the set of employees $se?$ based on the foreign key $SupENum$. Excluding $se?$ from the transitive closure gives the primary keys of the supervised employees. A simple select operation then retrieves the employees tuples and builds the result ($sempl!$).

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ Empls\_supervised\ }}\\
\Xi DB\\
\\
se? : \mathbb{P}\, ENUM\\
sempl! : \mathbb{P}\, EMPL\\
\hline
\forall e : se? \bullet (\exists em : empls \bullet em.ENum = e) \land\\
\textbf{let } se1 == (\ FKTC\ empls\ ENum\ SupENum\ se?\ ) \setminus se? \bullet\\
\qquad\qquad\qquad sempl! = \{\ em : empls \mid em.ENum \in se1\ \}\\
\hline
\end{array}
$$

The other operation, $Empls\_sorted\_salary$, exemplifies the use of the $SORT$ operator according to rule A2 of the method and is presented below.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ Empls\_sorted\_salary\ }}\\
\Xi DB\\
\\
sempl? : \mathbb{P}\, EMPL\\
lempl! : \operatorname{seq} EMPL\\
\hline
sempl? \neq \{\ \} \land\\
lempl! = SORT\ sempl?\ Salary \geq\\
\hline
\end{array}
$$

Observe that the first equation in the predicate of the above schema defines a pre-condition which states that the set of employees to be sorted ($sempl?$) cannot be empty. In the context of transaction $Empl\_supervised\_sorted\_salary$ it means that employee $e?$ must be the supervisor of at least one employee.

The correct behaviour of the transaction ($Empl\_supervised\_sorted\_salary\_Ok$) is then described in terms of the two operations in the usual way.

$$
\begin{array}{l}
Empl\_supervised\_sorted\_salary\_Ok \mathrel{\widehat{=}}\\
\qquad Empls\_supervised\ [\{e?\} / se?] \gg Empls\_sorted\_salary
\end{array}
$$

The simplified precondition of the above transaction is given by:

$$\exists em : empls \bullet em.ENum = e? \land$$

$$\exists em : empls \bullet em.SupENum = e?$$

The error schema $Empl\_supervised\_sorted\_salary\_Error$ is then presented below.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{Empl\_supervised\_sorted\_salary\_Error} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi DB \\
\\
result! : STRING \\
e? : ENUM \\
\hline
(\ \neg\ (\exists\ em : empls \ \bullet \ em.ENum = e?) \ \wedge \\
\qquad\qquad result! = \text{“}Invalid\ employee\ number\text{”}\ ) \ \vee \\
(\ \neg\ (\exists\ em : empls \ \bullet \ em.SupENum = e?) \ \wedge \\
\qquad\qquad result! = \text{“}Employee\ does\ not\ supervise\ anybody\text{”}\ ) \\
\hline
\end{array}
$$

Finally, the total transaction *Empl_supervised_sorted_salary* is specified as usual.

$$
\begin{array}{l}
Empl\_supervised\_sorted\_salary \ \widehat{=} \\
\qquad\qquad (Empl\_supervised\_sorted\_salary\_Ok \ \wedge \ Ok) \ \vee \\
\qquad\qquad\quad Empl\_supervised\_sorted\_salary\_Error
\end{array}
$$

6.6.5 Transaction *Weighted_salary_proj*

The next transaction is *Weighted_salary_proj*. It calculates the average salary of employees working on project $p?$ weighted by the number of hours worked by each employee.

It has two sub-transactions. The first, *Empls_salary_hours*, selects the tuples of relation *works* that refer to project $p?$ and joins them with relation *empls* by the employee number *ENum*. The result is then projected to build an intermediate relation containing the employee number, the salary, and the number of hours worked by each employee.

The other, *Weighted_salary*, uses the intermediate relation, built by the previous operation, to calculate the weighted salary. It adds the product of the salaries by the numbers of hours worked and divides the result by the total number of hours worked.

Before these are specified, an auxiliary relation intention (*EMPL_WORK*) is defined as usual (rule D2) to represent the intermediate relation.

$$EMPL\_WORK \ \widehat{=} \ [ENum : ENUM \ ; \ \ Salary : SALARY \ ; \ \ Hours : HOURS \ ]$$

Now, the specification of the first subtransaction is presented. It shows how simple select, join, and project operations (rules B1 to B4) can be applied in the same schema.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{Empls\_salary\_hours} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi DB \\
\\
p? : PNUM; \\
sempl\_work! : \mathbb{P}\ EMPL\_WORK \\
\hline
\exists\ pj : projs \ \bullet \ pj.PNum = p? \ \wedge \\
sempl\_work! \ = \ \{\ e : empls; \ w : works \ \mid \\
\qquad\qquad\qquad w.PNum = p? \ \wedge \ e.ENum = w.ENum \\
\qquad\qquad\qquad\qquad \bullet \ (e.ENum, e.Salary, w.Hours)\ \} \\
\hline
\end{array}
$$

Operation *Weighted_salary* uses the intermediate relation *sempl_work?* to calculate the weighted salary. The salary and number of hours worked by each employee are multiplied and the results are added to be divided by the total number of hours worked. Notice that relation *sempl_work?* cannot be empty, i.e. it must have at least one tuple, and that $*$ and $/$ refer to the multiplication and division of numbers, respectively.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Weighted_salary}} \\ \hline
\Xi DB \\[4pt]
sempl_work? : \mathbb{P}\, EMP_WORK \\
weighted_sal! : SALARY \\ \hline
sempl_work? \neq \{\ \} \land \\[4pt]
\mathbf{let}\ \ sum_salary == SUM\ sempl_work?\ (Salary * Hours); \\
\qquad\qquad sum_hours == SUM\ sempl_work?\ Hours \bullet \\
\qquad\qquad\qquad weighted_sal! = sum_salary\ /\ sum_hours \\ \hline
\end{array}$$

Again, the correct behaviour of the transaction is simply specified as the result of piping its two sub-transactions and is presented below.

$$Weighted\_salary\_proj\_Ok \mathrel{\widehat{=}} Empls\_salary\_hours >> Weighted\_salary$$

The simplified precondition of the above transaction is given by:

$$\exists\, pj : projs \bullet pj.PNum = p? \land$$

$$\exists\, w : works \bullet w.PNum = p?$$

The corresponding error schema *Weighted_salary_proj_Error* is presented below.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Weighted_salary_proj_Error}} \\ \hline
\Xi DB \\[4pt]
result! : STRING \\
p? : PNUM \\ \hline
(\ \neg\ (\exists\, pj : projs \bullet pj.PNum = p?) \land \\
\qquad result! = \text{“}Invalid\ project\ number\text{”}\ ) \lor \\[4pt]
(\ \neg\ (\exists\, w : works \bullet w.PNum = p?) \land \\
\qquad result! = \text{“}No\ employees\ working\ on\ project\ p\text{”}\ ) \\ \hline
\end{array}$$

Once again, the total transaction *Weighted_salary_proj* is specified in terms of *Weighted_salary_proj_Ok* and *Weighted_salary_proj_Error* in the usual way.

$$\begin{array}{l}
Weighted_salary_proj \mathrel{\widehat{=}} \\
\qquad (Weighted_salary_proj_Ok \land Ok) \lor Weighted_salary_proj_Error
\end{array}$$

6.6.6 Transaction *Fire_selected_empls*

Finally, transaction *Fire_selected_empls* is going to be specified. It will be used to fire all male employees whose salaries are greater than a certain limit (*high_salary?*) and remove them from the projects they work on. Managers of departments satisfying these conditions cannot be fired but should be returned as a separate output.

Its proposed sub-transactions are: (1) *Empls_sex_salary*, that selects from relation *empls* tuples where attribute *Sex* is *Male* and *Salary* is greater than the limit *high_salary?* and returns the ones which refer to managers of departments (*sman!*) as well as the primary keys of the ones which do not (*se!*); (2) *Delete_empls_Pk*, that delete employees based on a set of primary keys *se?*; and (3) *Delete_works_empls*, that deletes tuples of relation *works* based on a set of employees' primary keys *se?*.

The first of these operations, *Empls_sex_salary*, is presented below.

$$
\begin{array}{l}
\textit{Empls\_sex\_salary} \\
\hline
\Xi DB \\
high\_salary? : SALARY \\
se! : \mathbb{P}\, ENUM \\
sman! : \mathbb{P}\, EMPL \\
\hline
high\_salary? \neq NULLREAL \;\wedge \\
\textbf{let}\;\; sempl == \{\, em : empls \mid em.Sex = Male \;\wedge \\
\qquad\qquad\qquad\qquad em.Salary > high\_salary? \,\} \;\bullet \\
\quad (\, sman! = \{\, em : sempl \mid \\
\qquad\qquad (\exists\, dp : depts \;\bullet\; dp.ManENum = em.ENum) \,\} \;\wedge \\
\quad\;\; se! = \{\, em : sempl \mid \\
\qquad\qquad \neg\, (\exists\, dp : depts \;\bullet\; dp.ManENum = em.ENum) \\
\qquad\qquad\qquad \bullet\; em.ENum \,\} \,) \\
\hline
\end{array}
$$

Schema *Delete_empls* specifies deletions of employees based on its primary key *ENum* and has already been specified.

The last sub-transaction, *Delete_works_empls*, deletes all tuples of relation *works* which refer to employees whose primary keys are in the set *se?*. Following rule B8 of the method, this is specified in terms of the schema that specifies deletions based on the primary key, i.e., schema *Delete_works_Pk*.

$$
\begin{array}{l}
\textit{Delete\_works\_empls} \\
\hline
\Delta DB \\
se? : \mathbb{P}\, ENUM \\
\Xi DB \setminus works \\
\hline
\textbf{let}\;\; sdw == \{\, w : works \mid w.ENum \in se? \;\bullet\; (w.ENum, w.PNum) \,\} \\
\qquad\qquad \bullet\; Delete\_works\_Pk\; [sdw \,/\, sw?] \\
\hline
\end{array}
$$

It is important to emphasize that the application of the above schema represents exactly the same effect of the delete *Cascades* option for the foreign key *ENum* of relation *works*. The difference is that, here, it is used in a particular transaction instead of being applied after all deletions of employees.

Now, the correct behaviour of *Fire_selected_empls* is described in terms of its three basic operations in the usual way. Notice however that the order of the combination is important since the output of the first of the operations is to be used as input for the other two operations. In addition, the piping operator ($>>$) could have been used where the sequential composition ($\fatsemi$) was used, but the order of application would still have to be the same.

$$
\begin{array}{l}
Fire\_selected\_empls\_Ok \ \widehat{=} \\
\qquad Empls\_sex\_salary \ >> \ (Delete\_empls \ \fatsemi \ Delete\_works\_empls)
\end{array}
$$

The simplified precondition of the above transaction is simply:

$$high\_salary? \neq NULLREAL$$

The corresponding error schema (*Fire_selected_empls_Error*) is presented below.

$$
\begin{array}{|l}
\multicolumn{1}{l}{Fire\_selected\_empls\_Error} \\
\hline
\Xi DB \\
result! : STRING \\
high\_salary? : SALARY \\
\hline
high\_salary? \ = \ NULLREAL \ \wedge \\
\qquad result! = \text{“}Limit\ high\_salary\ cannot\ be\ null\text{”} \\
\hline
\end{array}
$$

Finally, the total transaction *Fire_selected_empls* is specified in the usual way.

$$
\begin{array}{l}
Fire\_selected\_empls \ \widehat{=} \\
\qquad (Fire\_selected\_empls\_Ok \ \wedge \ Ok) \ \vee \ Fire\_selected\_empls\_Error
\end{array}
$$

6.7 Conclusion

In this chapter, the method for the specification of relational database applications, described in Chapter 4, was applied to the specification of a simple *Company Database* and its applications.

Chapter 7 now proceeds with the description of the theoretical mapping of formal specifications written according to the method to a generic relational database system.

Chapter 7

The mapping

In this chapter, the mapping process for the derivation of relational database programs directly from formal specifications written according to the method is presented. It is basically the description introduced in [122], with minor corrections.

The chapter is split into four sections: the first describes the mapping of the database structures, i.e. domains, relations and their attributes, and the constraints to be guaranteed. The second describes the mapping of basic operations over the database. The third covers more advanced features, such as transactions, sorting of results, and views. Finally, the fourth deals with the extension of the applications to capture error handling.

The topics of the mapping are named similarly to the rules of the method, i.e. using labels of the type Xn, where X can be D, standing for database structure, B for basic operations, A for advanced features, or E for extended applications to capture error handling, and n is a sequential number within each topic, with subitems when necessary.

The efficiency of the generated code, though taken into account, is not a primary concern. In fact, it is sometimes disregarded in order to make the mapping as smooth as possible. However, this does not mean the generated programs are terribly slow because a number of these operations are optimized by the compiler.

There is no intention to bind the mapping process to specific DBMSs and query/host languages (or 4GLs). Even so, the DBPL [9] system is extensively used together with parts of the company database example (presented in Chapter 6) to illustrate the process. Occasionally, SQL [21] and dBASE-IV [123] are also mentioned in the discussion.

Finally, although an effort is made to keep the generated programs as close to the specifications as possible so that the mapping is simple, it is not always possible to achieve this simplicity. In some cases, in addition to the relevant data from the corresponding section of the specification, the implementation includes data from other parts of the specification method. It is also sometimes necessary to incorporate design decisions into the mapping so that the generated programs are syntactically correct.

7.1 Mapping the database structures and constraints

In this section, a general discussion on the mapping of the structure part of database specifications written according to the method is presented. It includes the specification of domains, relations and their attributes, and the constraints to be guaranteed.

In the specific case of relational systems which do not support constraints directly, the necessary constraints will have to be enforced explicitly in the implementation of the transactions that can possibly violate the integrity of the database (it is assumed the database is in a valid state before the operations).

The most direct way of generating the necessary conditional expressions in the application programs is from the preconditions of the transactions, more specifically, from the error schema(s) associated with the transaction. In fact, even when the constraints are supported, it is usually necessary to test the DBMS return codes so that specific error messages can be reported to the user. Therefore, part of the discussion about mapping the constraints is postponed to Section 7.4.

D1 - Domains

If the DBMS/query language does not support domains (or user type definitions), then all domains should be enforced by means of explicit constraints. Note that, in this case, avoiding operations between attributes and/or variables of different domains which are implemented by the same basic data type is not going to be possible.

All domains in the specification (attributes and variables as well) will, ultimately, be implemented by one of the basic data types offered by the DBMS and/or query language. Sometimes, these data types include a parameter giving the size they will occupy and, so, the mapping will have to incorporate some specific value as default.

If necessary, explicit constraints are to be enforced on the values that attributes and variables (drawn from the domains) can take. This depends on which kind of domain definition is used in the specification, which is briefly discussed below.

In theory, there should be no domain defined as a given set in the final version of the specification. If the correct implementation for a specific domain is not known at this stage, the basic type STRING should be used.

D1.1 - Domains defined as basic types

These refer to the simplest domain definitions possible, e.g. $ENUM == \mathbb{N}$. If domains or type definitions are supported, the translation of syntax is trivial. For example, in DBPL it would be implemented by the type definition `ENUM = CARDINAL;`.

If these are not supported, then all attributes and variables drawn from domain $ENUM$ are to be mapped as if they were of type $\mathbb{N}$ in the specification, and no other constraint is needed. In this case all attributes and variables specified, for example $Att : ENUM$, would have the type natural numbers (`CARDINAL`) in the implementation.

Some DBMSs and query languages ask for the size occupied by the attribute. So, a default value may be needed. For example, in dBASE IV such attributes must be written as `Att NUMERIC n,` where `n` is the number of bytes used to represent the attribute.

D1.2 - Domains defined as a subset of a basic type

An example of this kind of domain is given by the range definition $AGE == 0..18$. If this kind of syntax is supported, a straightforward translation is done. For example, it would be implemented as the type `AGE = 0..18;` in DBPL.

If this kind of definition is not supported then the domain should be implemented as if the specification were $AGE == \{n : \mathbb{N} \mid n \leq 18\}$, which is discussed below.

This more general form of subsets, e.g. $DNUM == \{n : \mathbb{N} \mid n > 100\}$, is unlikely to be supported by any current DBMS and, hence, it should be implemented as if the specification were $DNUM == \mathbb{N}$ (case D1.1) and a specific constraint (i.e. $var > 100$) had been written for all variables var drawn from domain $DNUM$.

D1.3 - Domains defined as an enumeration of values

An example of this kind of domain is given by $SEX ::= Male \mid Female \mid NULLSEX$. If enumeration types are supported, the translation is again trivial. For example, the corresponding DBPL implementation is the type `SEX = (Male, Female, NULLSEX);`.

If this specific kind of domains/types are not supported, it should be simulated by a convention using the natural numbers 0..N-1, where N is the number of different values of the domain representing its values. Therefore, it is reduced to case D1.2.

D2 - Relations

It is assumed that all RDBMSs provide a way of introducing relations. In some systems, the intention of relations (D2) and their corresponding extensions (D3) are created by separate definitions, just like in the specification method. Other systems provide a single definition. In both cases mapping the relations should involve a simple translation, and the types of the attributes would be either the domains implemented (if supported) or the basic types chosen to implement them, as described in rule D1. An example of relation specification equations is presented below:

$$PROJ \;\widehat{=}\; [\; PNum : PNUM \;;\; DNum : DNUM \;]$$

$$projs : \mathbb{P}\, PROJ$$

In DBPL, the equations above can be defined in a single step or in two separate steps. The one using two separate equations is preferred because (1) it is closer to the specifications and (2) it provides a name for the record type (the tuple) which will make it easier to map other parts of the specification.

The intention of the relation above would then be written as:

```
TYPE Proj = RECORD PNum: PNUM; DNum: DNUM; END;
```

In dBASE IV, the relation intention and extension are defined in a single step. In this case, the above specifications would be implemented as a file called `Proj.dbf`, which would contain the information about the structure of the relation (attribute definitions) and would also store the actual data.

D3 - Relations (extension)

In the case of DBMSs which support the definition of the relations in two steps, mapping the specification of the extension of relations (e.g. $projs : \mathbb{P}\, PROJ$) should also be simple.

In DBPL the relation extension would be implemented as follows:

```
TYPE REL_PROJ = RELATION OF PROJ;
```

D3.1 - Required attributes

When the DBMS supports nulls, this should again involve a simple translation to the appropriate syntax. In SQL systems, for example, it is usually done by writing the keywords `NOT NULL` after each attribute definition.

If nulls are not supported, they should be simulated in a style similar to the *default values* approach [120] by enforcing an extra constraint on the attributes that cannot be null saying that each of them cannot assume the chosen null value for its domain.

Neither DBPL nor dBASE IV support null constraints and, thus, they have to be simulated. Basically, for each basic type provided by the DBMS, a null constant must be chosen, though appropriate default values for the usual basic types are already provided by the method. The conditional expressions needed to enforce the constraint would then be generated from the error schema associated with the transactions and are discussed later (Section 7.4).

D3.2 - Candidate keys

For the sake of the implementation, it is assumed that the first use of the *KEY_OF* operator in each relation schema refers to the primary key whereas the others (if any) are secondary keys.

In theory, all DBMSs should provide a way of saying which attribute(s) form the primary keys and, so, a simple translation should be enough. Some systems do not enforce the primary key constraints though.

In DBPL, the complete definition of the relation extensions include the primary key attributes. So, the full definition of relation Proj is given by:

```
TYPE REL_PROJ = RELATION PNum OF PROJ;
```

However, in order to be able to inform the user of any violations of the primary key constraints detected by the DBMS, it is necessary to test the relevant DBMS return code (`RESTRICTED()`, in DBPL) after *all* insert operations as well as any updates of the primary key attribute(s)[1].

Thus, as far as real database applications are concerned, the fact that primary key constraints (and more generally any kind of constraints) are supported by the DBMS does not necessarily mean the mapping to an implementation is going to be simpler.

Secondary keys should be defined similarly when supported.

If any of these is not supported, the uniqueness constraint ought to be enforced explicitly. Unique indexes are usually supported and should be used in these cases. In some DBMSs, e.g. DB2 [124], the use of indexes is compulsory. Otherwise, key constraints should be enforced from the error schemas associated with the transactions, similarly to the null constraints.

[1] In fact, updates of the primary key attribute(s) are not allowed in DBPL.

D3.3 - Static attribute constraints

If the DBMS supports the specification of such constraints, the appropriate syntax should be used. Otherwise, they should be enforced from the error schemas associated with the transactions, as any other constraints.

Even though constraints of the form $\forall t : rel \bullet \langle\text{condition}\rangle$ can be expressed in DBPL as `ALL t IN rel ( <condition> )`, where `<condition>` must be of type `BOOLEAN` and might use `OR`, `AND`, `NOT`, `ALL` ($\forall$), and `SOME` ($\exists$), the DBPL system does not provide a way of enforcing them. Thus, the appropriate tests that guarantee the consistency of the database should still be generated from the error schemas.

D4 - The database schema

Usually, there is nothing extra to be done in this case. In DBPL the relation extensions must be declared inside a database module.

```
DATABASE MODULE DB;
    ... relation extensions ...
END DB.
```

D4.1 - Foreign key constraints

Strictly speaking, these are a special case of static attribute constraints. Thus, if not supported they should be treated as any other constraints otherwise the translation should be simple. Most DBMSs currently available, including DBPL and dBASE IV, do not support foreign key constraints.

D4.2 - Other static constraints

Strictly speaking, these are a special case of static attribute constraints (D3.3) where more than one relation is involved. Thus, the same comments as above apply.

In the special case of *derived attributes*, the DBMS might support a way of specifying them and, in this case, a simple translation would be used. If it does not, there are basically two possibilities to implement derived attributes: as queries that are called to calculate their values every time they are needed, and as explicit attributes that cannot be updated by the users. These are presented below.

(A) If derived attributes are implemented as queries that are called to calculate their values every time they are needed, the specifications should, in principle, map more or less easily to the implementation, at least in DBPL. However, this might not be a good idea if the derived attribute is used frequently.

In DBPL, the main part of such a procedure for calculating the number of employees of department d ($d.NEmp = \# \{ e : empls \mid e.DNum = d.DNum \}$, specified in the company database example) would be written as below, where `CARD` returns the number of tuples of a relation and `REL_EMPL` is the type of the tuples of the relation.

```
d.NEmp  :=  CARD  (  REL_EMPL  { EACH  e  IN  empls :
                                       e.DNum  =  d.DNum }  );
```

(B) The other possibility is to implement derived attributes as explicit attributes which cannot be updated by the users. In this case, the attribute will be set to a consistent value (usually zero for numeric attributes) every time a new tuple is created. Each insert, delete or update operation in relations affecting the value of the derived attribute will then cause it to be updated if necessary.

This approach is not as simple as the previous one but should be more efficient, especially if there are few updates to the derived attribute.

Again, in the case of the number of employees of department d, the following steps would be performed: (1) for each new department d inserted, $d.NEmp$ would be set to zero; (2) for each new employee e inserted, the number of employees of the relevant department ($e.DNum$) would be updated; (3) for each employee e deleted, the number of employees of the relevant department would also be updated; and (4) updates of employees do not affect the derived attribute and, so, no extra update is needed.

(B1) A possible way of implementing these updates is to call a subtransaction that sets the value of the derived attribute similarly to the one presented in case (A).

The advantage of this approach is its simplicity since the subtransaction is the same for all operations being executed against the employees relation and the mapping from the specifications is simple. Its disadvantage is its efficiency because it involves a number of I/O operations that can be avoided.

(B2) An alternative solution to the example is to add one to the number of employees of the relevant department after each insert, and subtract one after each delete in relation employees.

To implement this in DBPL, the following piece of code would be included in the *Insert_empls* subtransaction, where `sempl` is the set of new employees tuples inserted.

```
FOR EACH t IN sempl : TRUE DO
     depts[t.DNum].NEmp := depts[t.DNum].NEmp + 1;
END;
```

The *Delete_empls* subtransaction would then include the code below, where `sekey` is the set of primary keys of deleted employees.

```
FOR EACH t IN empls : t.ENum IN sekey DO
     depts[t.DNum].NEmp := depts[t.DNum].NEmp - 1;
END;
```

Obviously, this last option is the best for this kind of derived attributes, *but* it does *not* cover all cases. When it is not possible to use this simpler approach, it should always be possible to use one of the other two approaches presented before.

However, this approach covers many cases with slight changes. For example, if the derived attribute is based on a sum (or product) of other attributes, it may be easily implemented similarly, by adding and subtracting (or multiplying and dividing by) the value(s) of the attribute(s) it is derived from. It would also be necessary to update its value after every update in the attribute(s) it is derived from.

D5 - The ΔDB schema

Usually, there is nothing extra to be done in this case.

D5.1 - Dynamic constraints

Dynamic constraints also depend on the previous values of the updated attributes, but are essentially similar to other constraints. Thus, in the unlikely case they are supported, the appropriate syntax should be used, otherwise the appropriate tests to enforce them are generated from the error schemas associated with the transactions.

D6 - The ΞDB schema

Again, there is nothing extra to be done in this case.

D7 - Initialization

Most DBMSs create empty relations. Otherwise, the user must ensure that all relations are created empty. In DBPL the user must provide an initialization transaction for each database module. As in other parts of the translation, the need to explicitly write the types of the tuples of each relation is the only difficulty.

```
TRANSACTION Init_DB;
BEGIN
    empls  :=  REL_EMPL { };
    ... other relations initialization ...
END Init_DB;
```

7.2 Mapping the database operations

Now, the discussion moves on to the translation of the operations specified according to the method. For organizational purposes, the operations are divided into two groups: read-only operations, which do not modify the database, and update operations, which modify the database by inserting, updating, or deleting tuples of relations.

The input and output variables declared in the specifications are to be translated to variable declarations and input and output commands of the implementation language respectively. Notice that only simple input/output commands are to be generated. In languages which support transactions as special procedures (e.g. DBPL), these variables may alternatively be passed as value and variable parameters respectively. Again, these should only involve straightforward translation (details are omitted).

B1 - Read-only operations

If the DBMS/query language supports set at a time operations, the translation should be reasonably simple. Mapping set at a time to record at a time operations should not be too difficult but it is not going to be investigated at this point, even though it is needed for the case of query languages embedded in imperative programming languages. From now on, it is assumed that set at a time operations are supported.

B2 - Select

Mapping the set comprehension used in the specification of selects to the supported implementation syntax should be reasonably straightforward.

The selection $res! = \{ t : rel \mid \langle\text{condition}\rangle \}$ should be translated to DBPL as

```
res  :=  REL_RES  {  EACH  t  IN  rel :  <condition>  };
```

where `REL_RES` is the type of the tuple of relation variable `res` and `<condition>` is a boolean expression involving at least one of the attributes of `t`, i.e., relation `rel`.

B3 - Theta-join

Mapping the set comprehension used in the specification of theta-joins should also be simple. The expression $res! = \{ t1 : rel1;\ t2 : rel2\ \dots \mid t1.Att1\ \langle\text{cop}\rangle\ t2.Att2\ \dots \}$ should be translated to DBPL as

```
res  :=  REL_RES  {  {a, b, ...}  OF
                    EACH  t1  IN  rel1,  EACH  t2  IN  rel2,  ...  :
                                              (t1.Att1 <cop> t2.Att2 ...)  };
```

where `REL_RES` is the type of the tuple of relation extension variable `res`, `{a, b, ...}` is the list of all attributes of all relations joined, `<cop>` is a comparison operator, and `Att1` and `Att2` are attributes of `rel1` and `rel2` respectively.

Obviously, the need to list all attributes `{a, b, ...}` of all relations joined is a drawback of DBPL, but this should *not* add any difficulty to the mapping. In addition, in practice most joins are used together with projects and, so, this drawback should not be particularly relevant.

B4 - Project

The mapping of the project operations should also be straightforward. The equation $res! = \{ t : rel \bullet \langle\text{result}\rangle \}$ should be translated to the following DBPL code

```
res  :=  REL_RES  {  { <result> }  OF  EACH  t  IN  rel :  TRUE  };
```

where `REL_RES` is the type of the tuple of relation extension variable `res`, and `<result>` is the list of attributes of `rel` on which the project operation is based.

B5 - Update operations

In general, the ΔDB and ΞDB expressions are not going to be translated at all because, in the implementation, (1) the relations before and after the operations are usually denoted by the same name and (2) relations not used remain unchanged.

The details of the mapping of specific operations are described in the five following rules (B6 to B10) and in the following section.

B6 - Insert

The translation of the insert operations should also be straightforward for most DBMSs, even if set at a time insertion is not supported.

In DBPL, the equation $empls' = empls \cup sempl?$ is to be translated to the one below, where `sempl` is the set of new employees to be inserted.

```
empls  :+  sempl;
```

B7 - Delete by primary key

In general, current DBMSs and query languages do not support the definition of higher order functions with generic types. If these were supported, the approach would be to write a generic higher order function to implement the operator *DELETE* used in the specifications, and this would make the mapping very simple.

Assuming these are *not* supported, the solution is then to translate all occurrences of *DELETE* to the appropriate delete command of the DBMS/query language chosen. Some problems might arise if, for instance, set at a time deletions are not supported or if the delete command takes complete tuples to delete instead of their primary keys only (like in the specifications), but the translation should still be reasonably simple.

In DBPL, the deletion $empls' = DELETE\ empls\ ENum\ sekey?$ should be translated to the following delete command,

```
empls  :-  REL_EMPL  { EACH  t  IN  empls :  t.ENum  IN  sekey };
```

where `REL_EMPL` is the type of the tuple of relation `empls`, and `sekey` is the set of primary keys of employees to be deleted.

When the primary key of the relation is the target of one or more foreign keys, either in other relations or in the same relation, the foreign key compensating actions should also be written as part of the delete by primary key operations, similarly to the way it is done in the specification level. Although the mappings of these compensating actions were not fully investigated, a partial discussion on these mappings is presented in rules B7.1, B7.2 and B7.3.

B7.1 - Deletes restricted

The specification of *Restricted* is normally done by default and, so, no implementation is needed. Explicit redundant specification of *Restricted* should thus be ignored in the translation.

B7.2 - Deletes cascade

In the case of *Cascades*, the specification depends on whether the foreign key is part of a cycle of foreign keys that cascade for deletes or not, and so does the translation.

(A) If the foreign key $Fk1$ in relation $rel2$ is *not* part of such a cycle, and usually this is the case, the schema that specifies deletes based on the primary key of relation $rel1$ includes the expression below,

$$\textbf{let}\ \ sdr2 == \{\ t2 : rel2 \mid t2.Fk1 \in sk1? \bullet t2.Pk2\ \} \bullet \\ Delete\_Rel2\_Pk\ [sdr2\ /\ sk2?]$$

which says that the primary keys of tuples of relation $rel2$ referring to deleted tuples of relation $rel1$ by the foreign key $Fk1$ are collected and passed to the schema that specifies deletes by the primary key in relation $rel2$, i.e., $Delete\_Rel2\_Pk$.

If relation $rel2$ has a composed attribute primary key, the specification still follows the same rule. For example, suppose the foreign key attribute *ENum* in relation *works* of the company database example had *Cascades* chosen for deletions on relation *empls*. Then, schema *Delete_empls_Pk* which deletes employees by the primary key would include the following expression:

$$\textbf{let}\ \ swkey\ ==\ \{\ w : works\ \mid\ w.ENum \in sekey?\ \bullet\ (w.ENum, w.PNum)\ \}\ \bullet \\ Delete\_works\_Pk\ [swkey\ /\ sw?]$$

The general idea is to implement these specifications as combined select and project operations which return a set of primary keys of the relevant relations according to rules B2 and B4 already presented. These keys should then be passed to the operations that implement deletes based on the primary keys of the corresponding relations.

In DBPL, the translation is carried out quite naturally and uses an auxiliary variable (mapped from the **let** expression) and a parameterized procedure call to the relevant delete procedure (`Delete_works`). The implementation code derived from the above specification is given below:

```
swkey  :=  REL_WORK_K  { {w.ENum, w.PNum}  OF
                                  EACH  w  IN  works :  w.ENum IN sekey };
Delete_works  ( swkey );
```

where `swkey` is an auxiliary variable of type `REL_WORK_K`, which is the relation type of the *set of primary keys* of relation `works`, and `sekey` is the set of primary keys of employees to be deleted.

(B) When the foreign key *is* part of a cycle of foreign keys that cascade for deletes, the effect of *Cascades* is not specified in the same way, because there can be no cycles in the use of schemas as predicates. In the general case, the specifications merely state that, after the deletion of the set of keys selected, all relations after the operation are the maximal subsets of the original relations to satisfy the database constraints.

The mapping of this case in general is likely to be very difficult and is not going to be investigated. The problem is that the information on the relevant foreign keys is not available in this part of the specification. It is probably still possible to work out what are the foreign keys which need to be considered, but it would involve checking the foreign key constraints and, maybe, even the specification of the other delete schemas.

However, if the DBMS and/or the query/host language supports recursion, the only difficulty would be identifying the relevant foreign keys, since the target implementation ought to be basically the same as presented in case (A), except that an explicit stop condition for the recursion must be provided within the procedure that implements deletes by the primary keys in *one* of the relations involved. Even so, it is important to point out that the existence of such a cycle of *Cascades* is not common and should be avoided whenever possible, because it can potentially destroy the database.

(C) In the particular case of delete *Cascades* where the foreign key is targeted at the same relation (loop), the operator *CASC_DELETE* is used in the specification and the relevant foreign key is explicitly mentioned.

Hence, if recursion is supported, the implementation code is similar to that presented in case (A), the difference being the delete procedure call, which is recursive and is only activated if there are foreign key references to deleted tuples. So, the mapping is again relatively simple.

Given that a foreign key specified as *FOR_KEY depts SupENum empls ENum* (supervisor of employees) is to cascade for deletes, the predicate of the corresponding delete schema should be: $empls' = CASC\_DELETE\ SupENum\ empls\ ENum\ sekey?$.

In DBPL, the body of the corresponding implementation procedure `Delete_empls`, introduced below, should include the standard delete command, as presented in the general case (rule B7), followed by the code which implements the cascade deletes option using recursion. Notice that the name of the auxiliary variable, `sekey_aux`, as well as its type, `REL_EMPL_K`, are mapped from variable `sekey`, the set of primary keys of employees to be deleted, which is a parameter of *CASC_DELETE*.

```
empls     :-  REL_EMPL   {  ...  sekey  ...  };

sekey_aux  :=  REL_EMPL_K  {  e.ENum  OF
                               EACH  e  IN  empls  :
                                     e.SupENum  IN  sekey  };

IF  sekey_aux  <>  REL_EMPL_K { }  THEN
    Delete_empls  ( sekey_aux );
END;
```

In addition, it is worth mentioning that, although the target code is basically the same as that of case (A), the mapping is completely different and a lot simpler than that of case (A) because most of the generated code is embedded in the mapping and pasted into the implementation whenever the *CASC_DELETE* operator is used.

However, when recursion is not supported, it is easy to simulate it with an explicit iteration of deletions and, thus, the mapping is still simple. Basically, the implementation code would use an intermediate variable, initialized with the set of keys to be deleted, and a `WHILE` loop containing the appropriate delete command which would iterate while there are foreign key references to deleted tuples.

In DBPL, the implementation code without using recursion is presented below, where the name and type of the auxiliary variable `sekey_aux` are mapped as before, and "..." stands for omitted details which are unchanged from the previous case.

```
sekey_aux  :=  sekey;

WHILE  sekey_aux <> REL_EMPL_K { }  DO
       empls  :-  REL_EMPL  {  ...  sekey_aux  ...  };
       sekey_aux  :=  REL_EMPL_K  {  ...  sekey_aux  ...  };
END;
```

B7.3 - Deletes nullify

In the case of *Nullifies*, as in the general delete (rule B7), the ideal implementation would be to write a generic higher order function to implement the operator *UPDATE* used in the specifications, and this would make the mapping very simple.

Assuming these are *not* supported, the solution is then to embed the knowledge of *UPDATE* in the mapping and use the appropriate command or sequence of commands of the chosen DBMS/query language to implement it.[2] Therefore, the aim here is to generate the code that changes to *null* all foreign key references to deleted tuples.

As far as the implementation is concerned, there is in general no need to explicitly say that some tuples are not going to be changed. Basically, the tuples that will be changed are to be selected, and the change to be made and then effected, i.e. written to the database. Changing to *null* the value of the relevant attribute in the selected tuples should map easily to an assignment in the implementation language.

For example, if the foreign key attribute *ManENum* (manager of departments) were to be nullified whenever managers were deleted, the predicate of the delete schema would include the equation: $depts' = UPDATE\ depts\ ManENum\ sekey?\ NULLNAT$.

The corresponding DBPL implementation code is given below, where `NULLNAT` is the appropriate null value.

```
FOR EACH t IN depts : t.ManENum IN sekey DO
    t.ManENum := NULLNAT;
END;
```

In the specifications, when *Nullifies* is chosen and the foreign key refers to the same relation, a **let** expression and a local variable are used to join the equation above with the one that specifies the deletions (B7) because the result of one operation must be the input to the other. Still, this distinction is not needed in the implementation level.

For instance, if the *Nullifies* option were chosen for the supervisor of employees foreign key attribute *SupENum*, this effect would be specified using a **let** expression. Even so, the corresponding implementation in DBPL should be:

```
FOR EACH t IN empls : t.SupENum IN sekey DO
    t.SupENum := NULLNAT;
END;
```

Observe that this mapping strategy should also be the base for the translation of the operator *UPDATE* used in other contexts, with the possible exception of the updates of primary keys, discussed in rule B10. Moreover, the implementation code embedded in the mapping of *UPDATE* should be very similar to the code to be generated for the updates of attributes (rule B9).

[2] In fact, this strategy is to be used whenever an operator is used in the specification, unless explicitly stated otherwise. From now on this is not going to be mentioned anymore.

B7.4 - Special case (recursive cascade deletes)

Regarding the special case of the specification method which prescribes the use of the expression $\{\ t1 : (rel1 \setminus rel1') \bullet t1.Pk1\ \}$ in all places where $sk1?$ appears in rules B7.1, B7.2 (A) and B7.3, if relation $rel1$ is part of a cycle of foreign keys that cascade for deletes, there is in general no need to do the same at the implementation level.

In theory, the implementation should still use `sk1` in those places where the expression above was used, but the code corresponding to foreign key compensating actions of relations outside the cycle of *Cascades* should be placed within the recursion or iteration which simulates it.

The mapping for this case has not been investigated in detail though. So, there might be overlooked details which could cause difficulties in its implementation but its investigation is not going to be pursued further.

B8 - Other deletes

At the specification level, any other deletes are written in terms of combined select and project operations, which return the set of primary keys of the relevant relations, and the schemas that specify deletes based on the primary keys of the relations.

This is exactly how the general delete *Cascades* option is specified – rule B7.2 (A) of the method. Therefore, the corresponding implementation code, omitted here, should be mapped similarly.

Notice that, essentially, this kind of delete adds nothing to the expressivity and complexity of the specification method since the same results can be achieved by writing an extra subtransaction that selects the primary keys of the tuples to be deleted and passes the result to the appropriate delete subtransaction. Additionally, the generated code is in both cases inefficient since the relation will in general be read twice.

B9 - Updates

In theory, the implementation code for the updates of attributes should be similar to the code generated in the mapping of the operator *UPDATE*, described in rule B7.3. The mapping however is rather different because the specifications of updates are written in terms of a select condition and an update rule, instead of using *UPDATE*.

The general rule for the specification of updates is given below, where <condition> is a boolean expression based on attributes of relation *rel* and <result> is an expression that, applied to any tuple t, returns the corresponding updated tuple.

$$rel' = \{\ t : rel \bullet \textbf{if}\ \langle\text{condition}\rangle\ \textbf{then}\ \langle\text{result}\rangle\ \textbf{else}\ t\ \}$$

Even though <result> may be any expression of type *REL*, it usually is an expression which uses the *But* ($\setminus$) operator to define the value of the the modified attributes, and this is the only case that is going to be investigated in detail.

As mentioned in rule B7.3, there is, in general, no need to explicitly say that some tuples are not going to be changed. Moreover, changing the value of attributes should map easily to assignments in the implementation language and, indeed, this is the case of *But* ($\setminus$) expressions of the form $t \setminus (Att1 = v1, Att2 = v2,\ ...)$.

Therefore, mapping the above specification to the appropriate implementation code should be fairly straightforward. In DBPL, updates using *But* (\) expressions should be implemented by the following code:

```
FOR  EACH  t  IN  rel :  <condition>  DO
     t.Att1 := v1;  t.Att2 := v2;  ...
END;
```

Another possibility is to specify <result> as a tuple of type *REL* such that, for each of its attributes, there is a corresponding expression which delivers the new value of the attribute in the relation tuples, usually in terms of its previous values. In other words, the attributes of <result> are expressions which usually involve computations using the previous values of the attributes in the relation tuples ($t.Att1, t.Att2$, etc.), even though constants can also be used and some attributes might be unchanged. Although mapping this case to an appropriate implementation should still be simple, it is not discussed further.

B10 - Update of keys

In the relational model, the update of the primary keys of one or more tuples of a relation is usually identical to the update of any other attribute of the relation.

Therefore, the corresponding implementation should be strictly similar to the one described for the case of general updates (rule B9) and also in the *Nullifies* option of deletes based on the primary keys (rule B7.3).

Depending on how these updates are specified, i.e. whether the *UPDATE* operator is used or not, the mapping is the same as that of one of the two aforementioned rules and is not going to be repeated.

Nevertheless, some DBMSs (e.g. DBPL) do *not* permit changes to the primary keys of the tuples and these can only be achieved by a combination of deletes and inserts. Although this case is not going to be investigated completely, the specific case where the operator *UPDATE* is used to specify the update of the primary key of a single tuple is going to be explored.

The general approach is to use an auxiliary variable to save the old tuple, delete it from the relation, change the primary key attribute of the tuple variable, and then reinsert the tuple. Therefore, the mapping of *UPDATE* in this context will be different from the general mapping presented in rule B7.3.

For instance, changing the value of the primary key `ENum` of relation `empls` of the company database example from `old` to `new`, which according to the method is specified as $empls' = UPDATE\ empls\ ENum\ \{old?\}\ new?$, should be mapped to the following implementation (in DBPL):

```
empls_aux  :=  empls[old];
empls      :-  REL_EMPL  { empls_aux };
empl_aux.ENum  :=  new;
empls      :+  REL_EMPL  { empls_aux };
```

When the primary key of the relation is the target of one or more foreign keys, either in other relations or in the same relation, the foreign key compensating actions should also be written as part of the updates of the primary key operations, similarly to the case of deletes based on the primary keys (B7). These are presented in rules B10.1, B10.2 and B10.3 below.

B10.1 - Updates restricted

As in the case of deletes (B7.1), the specification of *Restricted* is normally done by default and, so, no implementation is needed. Thus, explicit redundant specification of *Restricted* should also be ignored in the translation.

B10.2 - Updates cascade

In this context, the specification of *Cascades* uses the operator *UPDATE* and is basically equivalent to the case of delete *Nullifies*, presented in rule B7.3, including the case where the foreign key is in the same relation. Hence, apart from using different values, the mapping is strictly the same and is not going to be presented again.

B10.3 - Updates nullify

The specification and mapping of the *Nullifies* option of primary key updates are also strictly the same as the *Nullifies* option of deletes by the primary keys (rule B7.3) and are not presented again.

7.3 Mapping the advanced features

This section refers to the mapping of the more advanced features, which include transactions (only the correct behaviour), sorting of results, aggregate functions, composite attributes, and views. The extended transactions including error handling are presented in the next section.

A1 - Transactions

Transactions are more complicated operations, potentially involving a number of simpler operations, which must execute as a whole or fail completely. Most RDBMSs support the definition of transactions and the appropriate syntax should be the target code.

The most common way of supporting transactions is by delimiting their scope with two special commands provided to the user: one to start a transaction and the other to end it successfully. Some DBMSs implicitly insert these delimiters in the beginning and at the end of application programs so that, by default, each program is a transaction. In others, these delimiters are written as a special kind of procedure.

A third command usually allows the user to abort the transaction at any time and will normally undo all the database updates already done. The component operations are simply written within the transaction scope using the normal syntax.

Regardless of these implementation details, the mapping should be simple for most DBMSs. If procedures are supported by the query/host language, they should be used to separate the correct behaviour of the transactions from the error handling code.

In DBPL, a transaction is just a special kind of procedure, the difference being it starts with the keyword `TRANSACTION`. Also, there is no automatic undo facility for unsuccessful transactions.

The approach to the mapping of the correct behaviour of the transactions to DBPL is to write them and their subtransactions (basic operations) as procedures, named after the corresponding specifications. Input and output variables are to be passed as value and variable parameters, respectively. Parameters of the subtransactions which are not parameters of the transaction should be mapped to local variables.

For example, transaction *Salary_dept* of the company database example returns the sum of the salaries of all employees hired by department *d?*. It was specified using two subtransactions: *Empls_of_dept*, that returns all employees *sempl!* hired by department *d?*, and *Sum_Salary_empls*, that receives a set of employees *sempl?* and returns the sum of their salaries *tot_sal!*. The corresponding DBPL implementation code, excluding error handling, is presented below.

```
TRANSACTION  Salary_dept  ( d: DNUM;  VAR tot_sal: SALARY;
                                      VAR result:  STRING );

    PROCEDURE  Salary_dept_Ok;      (**  Correct Behaviour  **)
    VAR  sempl:  REL_EMPL;
    BEGIN
        Empls_of_dept     ( d,  sempl );
        Sum_salary_empls  ( sempl,  tot_sal );
    END Salary_dept_Ok;

BEGIN
    ... error handling code ...
    Salary_dept_Ok;
    result  :=  "Success";
END Salary_dept;
```

A2 - Sorting of results

Most RDBMSs support sorting of results. In most cases, an external sort procedure is provided by the system and can be accessed from the query and/or host language. The specific syntax varies from one system to another but is usually straightforward and should be the target code for the translation.

In SQL systems, the command `SELECT` accepts an extra clause, `ORDER BY`, which takes the attribute(s) used to sort the results. In the case of descending order, each relevant attribute is followed by the keyword `DESC`.

A3 - Aggregate functions

If the DBMS supports aggregate functions, this should again involve a simple translation to the appropriate syntax. The details are omitted once again.

If aggregate functions are not supported, they should be simulated based on the specification of the relevant operators. The mapping should also be fairly simple.

In the specific case of DBPL, only the number of tuples of a relation (#) is directly supported (as the function `CARD`).

The next one, $COUNT$, which returns the number of tuples of a relation such that a given attribute is not null, is implemented using the function `CARD` and a selection. For example, the number of employees such that attribute Age is not null, specified as $COUNT\ sempl?\ Age$, should be translated to:

```
CARD  ( Rel_Empl  { EACH  t  IN  sempl :  t.Age <> NULLNAT } );
```

The functions MAX and MIN return the maximum and minimum value of a given attribute of a relation, respectively. For example, the maximum salary of an employee is specified as: $max = MAX\ sempl?\ Salary$.

The corresponding implementation code is presented below. Notice that tuples where the value of attribute $Salary$ is null (`NULLREAL`) are ignored.

```
max  :=  ...Minimum value of the domain...
FOR  EACH  t  IN  sempl :  t.Salary <> NULLREAL  DO
     IF  t.Salary  >  max  THEN
         max  :=  t.Salary;
     END;
END;
```

The implementation code for function MIN is omitted because it is very similar.

Similarly, the specification equation $tot\_sal! = SUM\ sempl?\ Salary$ returns the sum of the salaries of all employees and should be translated to the following code:

```
tot_sal  :=  0.0;
FOR  EACH  t  IN  sempl :  t.Salary <> NULLREAL  DO
     tot_sal  :=  tot_sal  +  t.Salary;
END;
```

Finally, function $AVER$ returns the average value of the attribute in the relation. Basically, it returns the result of SUM divided by the result of $COUNT$, but its result is explicitly defined to be zero if $COUNT$ returns zero, i.e., when the attribute is null in all tuples of the relation – this includes the case of an empty relation. A summary of the implementation code for the average salary is presented below.

```
aver  :=  ...the number of employees with salary...
IF  aver  <>  0  THEN
    sum   :=  ...the sum of the salaries...
    aver  :=  sum  /  aver;
END;
```

A4 - Composite attributes

In the specification level, composite attributes are needed to make possible the specification of composite candidate and foreign keys, as well as sorting of results based on more than one attribute.

In most RDBMSs, composite primary keys and sorting of results based on more than one attribute are defined in exactly the same way as their single attribute equivalents, except that all the attributes are listed. Sometimes they are separated by a specific delimiter, for example commas. In most cases the mapping is trivial.

In the specific case of DBPL, the equation $(CA2\ Att1\ Att2)$ in the specification of primary keys should be mapped to: `Att1, Att2`.

If foreign key constraints are supported, the composite foreign keys ought to be written in a similar style, i.e., by simply listing all the relevant attributes. The mapping should, therefore, be similar.

However, since foreign key constraints are usually not supported, they have to be enforced in the error handling code. In this case, composite foreign keys are not going to present any difficulty because occurrences of the operators $CA2$, $CA3$, etc. disappear in the simplification of the precondition of the transactions.

A5 - Views

Views are used to restrict the data visible to or updatable by a specific user or group of users. They may be composed of a number of base relations and virtual relations derived from base relations. Database systems which support views should provide the means for hiding base relations and, thus, prevent the users from accessing the relations they are not authorized to use.

As mentioned in Chapter 4, the updatability of views is not addressed in this work since this is, in its own right, a whole area of research [54]. In general, it is not even possible to decide whether some views are updatable or not [55]. Moreover, it is not always clear what the semantics of updates of specific views should be.

The mapping of views was not investigated in detail. Even so, it is argued briefly below that, in general, the mapping ought to be simple and comparable to those of other parts of the specifications.

Firstly, for most RDBMSs, the definition of view relations (intention and extension) is comparable to the definition of base relations. Thus, the mapping should be very similar to ones presented in rules D2 and D3.

Secondly, just like in the specifications, view relations based on operations are very similar to the basic operations on base relations (rules B1-B7). Hence, it is possible the corresponding mapping is also similar.

In addition, the mapping of the constraints associated with the view schemas *View* and $\Delta View$ should be exactly the same as the mapping of static and dynamic attribute constraints, respectively, since the specifications of the former are particular cases of the latter.

Finally, the operations involving view relations are specified very similarly to any other operations. Also, the implementation syntax for any given RDBMS should be exactly the same as the ones using only base relations. Therefore, the mappings should also be very similar (if not the same) to the ones presented in rules B1-B10.

7.4 The extended operations for error handling

According to the specification method, the predicates of the error schemas associated with the transactions are written as sequences of expressions of the form presented below, connected to each other by logical disjunctions ($\vee$).

$$(\text{<condition>} \quad \wedge \quad result! \ = \ \text{“}\mathit{error\ message}\text{”} \ )$$

In the expression above, <condition> stands for generic predicates representing the logical conditions which violate the precondition of the transaction. In general, each of them involves a combination of predicates connected by logical conjunctions ($\wedge$), disjunctions ($\vee$), and/or negations ($\neg$), as well as existential quantifiers ($\exists$).[3] Sometimes, it also includes set comprehensions.

The approach here is to map each of these generic predicates to the appropriate piece of implementation code that evaluates it and verifies, using a conditional statement, whether the precondition is violated. Whenever one of these predicates is *true*, the transaction must fail. This means that all changes which might have already been made must be undone, so that the consistency of the database is guaranteed. If an undo facility is supported it should be used whenever appropriate. Otherwise, no change should be made to the database before all such predicates are checked.

For most DBMSs, it should be simple to generate conditional statements of the implementation language from the aforementioned predicates, except for the existential quantifiers. When the DBMS supports existential quantifiers, the appropriate syntax should be used and the translation ought to be simple. Otherwise, the result of the evaluation of the existential quantifiers should be assigned to auxiliary boolean variables. These variables should then be used in the conditional statement.

These existential quantifiers are always based on relations and attributes, since they are derived from the precondition of the transactions. Therefore, the evaluation of these expressions can be implemented by checking whether the relevant read-only operations (i.e. select, join, and/or project) implicitly specified by their predicates actually return any data. If they do, the result is *true*, else the result is *false*.

For example, suppose *Salary_dept* is a transaction that returns the total salary of employees working for a given department $d?$. The predicate of the corresponding error schema *Salary_dept_Error* is given by:

$$\begin{array}{l} (\ \neg \ (\exists \ dp : depts \ \bullet \ dp.DNum = d?) \ \wedge \\ \qquad\qquad result! = \text{“}\mathit{Invalid\ department\ number}\text{”} \ ) \end{array}$$

[3] Expressions involving the universal quantifier ($\forall$) can be rewritten using the existential quantifier, because $\forall x : T \bullet p$ is equivalent to $\neg \ (\exists x : T \bullet \neg \ p)$ for any predicate p.

In DBPL, the universal and existential quantifiers can be mapped directly to the `FORALL` and `SOME` commands, which simplifies the problem. Thus, the corresponding DBPL error handling implementation code should be:

```
IF  NOT  (  SOME  dp  IN  depts  ( dp.DNum = d )  )  THEN
    result  :=  "Invalid Department Number";
    RETURN;
END;
```

If DBPL did not support the existential quantifier, it would be simulated by testing whether the select operation $\{dp : depts \mid dp.DNum = d?\}$ returns any tuples, and the result would be assigned to an auxiliary boolean variable as follows:

```
IF  REL_DEPT  {  EACH  dp  IN  depts :  dp.DNum = d  }
                                <>  REL_DEPT  { }  THEN
    exist_aux  :=  TRUE;
ELSE
    exist_aux  :=  FALSE;
END;
```

The error handling implementation code presented before would then follow, but the auxiliary variable `exist_aux` would substitute the existential quantifier (`SOME ...`) in the conditional statement. The resulting code is presented below.

```
IF  NOT  exist_aux  THEN.
    result  :=  "Invalid Department Number";
    RETURN;
END;
```

7.5 Conclusion

This chapter described the generic mapping of specifications written according to the method to a generic relational database system, even though the DBPL system was extensively used in the examples.

It is well known that the application of translation techniques between different query languages is a tractable problem [125].

The following chapter then proceeds with a detailed report on the experiment that was carried out and involved the construction of a prototype tool to partially implement the mapping for a specific RDBMS, namely the DBPL system.

Chapter 8

The prototype

This chapter describes the prototype that supports the method, which was presented in Chapter 4. It also partially implements the mapping presented in Chapter 7.

The prototype was implemented for a number of reasons: to show that the method is usable in the context of relational applications generation, specially in the presence of tool support; to show that it is possible to automate the mapping process for at least one target RDBMS; and to provide some evidence that the whole process is not particularly arduous and is, therefore, likely to be applicable to other target database systems.

Specifically, the prototype is meant to automatically generate relational database applications to be run on the DBPL [9] system. Nevertheless, it is worth emphasizing that DBPL is just the chosen example of a target database system/language.

In particular, this chapter summarizes the functionality and implementation details of the prototype tool that was built to support the method. In addition, it discusses the specific DBPL implementation problems that occurred during the development of the prototype, presenting the corresponding solutions; discusses the current status of the implementation; and provides a quick introduction to the *Synthesizer Generator* [10, 11], the tool used to build the prototype.

8.1 Design and implementation strategy

This section summarizes the strategy used in the design and implementation of the prototype tool. Since the developed tool is only a prototype, it was necessary to decide what should be included in its functionality.

Basically, the prototype provides a syntactic editor for the method and a partial implementation of the mapping described in Chapter 7. Its outputs are specifications written in Z (using the syntax provided by the `zed.sty` [126] style option for LaTeX) and relational database applications written in DBPL, respectively.

Regarding the implementation of the mapping, the general objective was to produce syntactically correct DBPL code from at least a subset of the operations, advanced features, and error handling schemas written according to the method, in addition to the complete structure of DBPL databases. Even so, the aforementioned subsets should be large enough to permit the automatic generation of syntactically correct programs, at least for some transactions.

8.1.1 Design decisions

In addition to the general objective, i.e. generating syntactically correct programs for at least some transactions, there were a number of other design decisions which are more implementation oriented.

Firstly, it was decided the syntactic editor should accept a large subset of all possible specifications which are correct according to the method, even though checking all the syntax details to enforce the method and reject ill-formed specifications would have to be given a low priority.

The main justification for this decision is the fact that these specifications are the inputs for the prototype implementation of the mapping process and, thus, restricting the correct specifications accepted by the editor would also mean restricting the automatic generation of the database programs.

As already mentioned, enforcing the correct syntax of the method was given a low priority in the implementation of the tool. Nevertheless, it is important to make it clear that this activity is *not* inherently complicated and that several different kinds of syntax checks were implemented as examples, mainly in the structure part of the specifications.

Another design decision was to try to generate the specifications automatically as much as possible, so that the actual typing done by the user would be restricted to a minimum. To achieve this, it was necessary to embed part of the semantics of the specification method in the syntactic editor.

More specifically, the idea was to automatically generate those parts of the syntax of the specifications that are unchanged among different instantiations of the same feature of the method, as well as all those parts that can be derived from other parts of the specification. Those repetitions of identifier names that are imposed by the method should also be automatically generated.

For instance, most of the syntax of one insert operation is exactly the same as that of any other insert operation and, so, these parts are automatically generated whenever the user says he will specify an insertion. In addition, the type of the auxiliary variable representing the tuples being inserted in a relation is always the same as that of the relation itself and, so, the former is derived as well. As a result, the user only needs to inform those parts which are missing, namely the name of the relation and the name of the auxiliary variable. Moreover, these are only informed once.

The last design decision made was not to use formal methods techniques in the implementation of the tool, mainly because it is not a production tool, only a prototype. However, since the prototype tool was built using the synthesizer generator and the inputs to this system are formal specifications, it is still correct to assert that the tool was formally specified.

8.1.2 Customizing the mapping process

After choosing a target database system to be used in an implementation of the mapping, regardless of whether it is a prototype or a production tool, it is necessary to adapt the generic mapping to the particular restrictions imposed by the chosen system and corresponding query/host language.

In the implementation of the prototype tool for generating DBPL programs, there was a single design strategy which proved to be very successful and, thus, it should be used in future implementations. This refers to writing by hand, in advance, the so-called ideal implementation programs corresponding to a reasonably large example specification, e.g. the company database example presented in Chapter 6.

The main benefit of writing the target implementation code beforehand is that it provides a concrete output for the mapping process. The existence of a concrete output proved to be very important because it helped to identify all the pieces of information which were relevant for the mapping and, consequently, to provide a better understanding of the mapping.

Another benefit provided by the hand-written code was to uncover a number of errors, omissions, and ambiguities in the previous descriptions of the mapping. Although the mapping is believed to be free from major errors, the feedback provided by using it to generate implementations targeted at other relational database systems and languages would certainly help to improve it further.

In addition, it is important to compile these hand-written programs and run the transactions using some test data to make sure they are syntactically and semantically correct, before trying to carry out the mapping. In the specific case of DBPL, a number of implementation problems were uncovered. These are discussed in Section 8.2

Finally, it is also possible that the ideal implementation code proves to be too difficult to derive (this has not occurred in the implementation of the mapping for DBPL though). In these cases, an alternative solution to the problem should be provided, i.e., the target programs should be changed so that the mapping can be as smooth as possible.

8.1.3 Tool support

The tool used to build the prototype is the *Synthesizer Generator*, which is a system for automating the implementation of language-based syntactic editors. Basically, the user provides a specification written in the Synthesizer Specification Language (SSL) and the system creates a syntactic editor for the language.

More specifically, the synthesizer generator supports the definition of views for the display of different information, possibly using different levels of abstraction, which are automatically updated by the system. If used together with the X Windows system, different views can be presented at the same time in separate windows.

In particular, this facility was exploited to automatically generate DBPL programs as a result of using the prototype syntactic editor that supports the method and the results were very encouraging indeed. As a matter of fact, exploiting the view facility to generate code written in a different language happens to be very easy, as long as the mapping is well defined.

A more detailed explanation on how to use the view facility is presented in Section 8.4 together with a quick introduction to the synthesizer Generator system.

To conclude, it is important to point out that the choice of which tool(s) to employ in the construction of an implementation can influence the results tremendously. Moreover, the synthesizer generator proved to be a very fitting tool for the kind of implementation carried out in this research.

8.2 Implementation problems

This section discusses the specific DBPL implementation problems that occurred during the development of the prototype tool and presents the corresponding solutions. Some of these problems regard limitations of the DBPL system whereas others regard features of the theoretical DBPL language which were not implemented.

8.2.1 Relation inclusion

The first problem that arose regards the DBPL operator for relation inclusion (`IN`) which is not currently implemented as an independent command of the language but only as part of the `EACH`, `SOME`, and `ALL` commands. For this reason, expressions such as `tuple IN relation`, which should be valid, are not accepted by the DBPL compiler.

Notice that the `IN` operator was extensively used in the examples presented in the general mapping (Chapter 7) and would be part of the derived code corresponding to several parts of the specifications written according to the method. In particular, these include the implementation of derived attributes, deletes based on the primary key, deletes cascade, and deletes nullify.

The adopted solution to this problem is then to use an existential quantifier to express the set inclusion. This means that such expressions (`tuple IN relation`) must be written as follows:

```
SOME  t  IN  relation  (tuple = t)
```

According to the group who implemented DBPL in the University of Hamburg, this operator was not implemented simply because there is an equivalent solution to represent it using an existential quantifier and, so, they decided to introduce this implementation restriction for simplicity.

However, personally, I think this was *not* a good implementation decision. Although the equation using an existential quantifier is only slightly more extensive, the original equation using the `IN` operator is much more intuitive and, thus, easier to understand. Also, in theory, it should be fairly simple to generate the same object code for both cases, probably just as simple as (or even simpler than) rejecting one of them.

8.2.2 Extended project operations

In general, the mapping of the project operations to DBPL is simple. The specification equations $res! = \{\ t : rel \bullet <\text{result}>\ \}$ are translated to the code presented below, where `REL_RES` is the type of the tuple of relation extension variable `res`, and `<result>` is the list of attributes of `rel` on which the project operation is based.

```
res  :=  REL_RES  {  { <result> }  OF  EACH  t  IN  rel :  TRUE  };
```

In theory, this mapping should cover all possible cases and these include the more general (extended) project operation which permits the use of any computations based on attributes of the relation in the projection list.

It is important to point out that, according to the DBPL user and system manual [9], these computations should be valid in DBPL. Even so, they have not been implemented and, therefore, are not accepted by the compiler.

Notice however that it is still possible to write such projections in DBPL but the mapping is a bit more complicated. The code to implement them involves an explicit `FOR EACH` loop over the relation, and the insertion of a tuple at a time through an auxiliary tuple variable. In addition, these operations do not merge with selects and theta-joins as easily as before.

For instance, consider the transaction *Weighted_salary_proj* of the company database example, presented in Subsection 6.6.5. Its first subtransaction, *Empls_salary_hours*, selects the tuples of relation *works* that refer to project $p?$ and joins them with relation *empls* based on the employee number *ENum*. The result is then projected to build an intermediate relation containing the employee number, the salary, and the number of hours worked by each employee.

Since the salary and the number of hours of each tuple of the intermediate relation will be multiplied later by subtransaction *Weighted_salary*, an equally valid alternative design would be to perform this multiplication in subtransaction *Empls_salary_hours*. The result would then be included in the intermediate relation (*sempl_work*!), instead of the salary of the employees. To conclude, subtransaction *Weighted_salary* would then merely divide the sum of this "multiplication attribute" by the total number of hours worked.

The main equation of the predicate of subtransaction *Empls_salary_hours* would then be specified as:

$$\begin{aligned} sempl\_work! \ = \ \{\ & e : empls;\ w : works \ \mid \\ & w.PNum = p? \ \wedge \ e.ENum = w.ENum \\ & \bullet \ (e.ENum, e.Salary * w.Hours, w.Hours)\ \} \end{aligned}$$

According to the general mapping, the specification above would be translated to the following DBPL code:

```
sempl_work  :=
   REL_EWORK  {  {e.ENum, (e.Salary * FLOAT (w.Hours)), w.Hours}  OF
                         EACH  e  IN  empls,  EACH  w  IN  works  :
                                (w.PNum = p)  AND  (e.ENum = w.ENum)  };
```

However, the DBPL compiler rejects computations (in this case the multiplication) in the projection list of relation expressions.

As already mentioned, the solution involves the declaration of an auxiliary tuple variable such as the one presented below. Notice that the name and type of this variable are mapped from the name and type of the corresponding relation, respectively.

```
VAR  empl_work : EMPL_WORK;   (***  Auxiliary tuple variable  ***)
```

The rest of the DBPL implementation code corresponding to the aforementioned specification is presented below.

```
sempl_work  :=  REL_EWORK  { };
FOR  EACH  e  IN  empls  :  TRUE  DO
     FOR  EACH  w  IN  works  :  (w.PNum = p)  AND
                                 (e.ENum = w.ENum)  DO
          empl_work.ENum    :=  e.ENum;
          empl_work.Salary  :=  e.Salary * FLOAT (w.Hours);
          empl_work.Hours   :=  w.Hours;
          sempl_work  :+  REL_EWORK  { empl_work };
     END;
END;
```

Finally, notice that even though the mapping is obviously more complicated than that of the general case, it is still possible to generate the above code automatically.

8.2.3 Sorting of results

The DBPL system does not specifically support sorting of results. However, I have developed a strategy to allow results to be sorted by an external call to the unix sort command.

The general approach used is to (1) copy the relation tuples to a temporary unix file; (2) call the unix sort with the appropriate parameters, which creates a second temporary file; and (3) list the sorted file.

The implementation code is written using low-level commands and, thus, it is not very short. However, the mapping *per se* is not too complicated because most of the target code is simply pasted into the translation.

Before discussing the mapping of the *SORT* operator to the appropriate DBPL implementation code, a summary of the modifications which were needed in the setup of the DBPL system is presented below.

- The standard modula-2 module `InOut`, which implements the low-level input and output procedures, was modified to accept a filename as a parameter in procedures `OpenInput` and `OpenOutput`, in addition to receiving it from a prompt to the user. This was done to make the automatic generation and manipulation of the temporary unix files used to sort relations transparent to the user running the transactions. In addition, the modified module was called `MyInOut` to make it clear that it is *not* the standard module.

- A module called `Aux_Procs` was written to make a number of auxiliary tasks more straightforward. One of its procedures, `UnixCommand`, simplifies the pipe to execute a unix command from within a DBPL program, which is reduced to a procedure call with the unix command being passed as a parameter of type `STRING`. The other procedure, `ListUnixFile`, receives a file name parameter and lists the contents of the corresponding unix file.

- The DBPL makefile `dbpl.mk` of the DBPL library was modified to include the new modules `MyInOut` and `Aux_Procs` in the list of modules which are automatically included in the linking stage of DBPL modules, so that these new modules can also be used without an explicit import declaration, just like the standard modula-2 modules.

Now, the relevant implementation code for sorting relations is discussed. Since the code is extensive and written in low-level language, only the main parts of the mapping and corresponding code are presented.

The first part of the mapping involve copying, to a temporary unix file, the tuples of the relation that must be sorted. The main issues involved in its mapping are discussed below.

- The first information needed is a file name for this intermediate file. The adopted convention is to name it `/usr/temp/rel.i`, where `rel` is the original name of the relation to be sorted.

- The second key issue is the actual writing of each tuple. In DBPL, it is not possible to write the complete tuple at once. Thus, it was necessary to write each of the attributes separatedly and to explicitly write the *newline* character at the end of the tuple. Also, the actual output commands require the length of the attribute being written. So, it was necessary to embed an appropriate number as default for each possible type of attribute. For example, the command used to print attribute `ENum` of type `CARDINAL` using 5 digits is: `MyInOut.WriteCard (t.ENum, 5);`.

After the intermediate unix file is written, it is necessary to call the unix sort with the appropriate parameters, which sorts the first file and creates a second temporary file. The name of this second file is, by convention, `/usr/temp/rel`, i.e. the name of the first file without the extension "`.i`".

Since an auxiliary procedure (`UnixCommand`) was written to simplify the execution of unix commands from within DBPL, the only issue is to generate the appropriate parameters for the unix sort command. In particular, the only difficulty here refers to the mapping of the sort key parameter. Basically, each key attribute must be mapped to the string "`+i.0 -j.0`", where `j` is the relative position of the attribute within the tuple (1 for first, 2 for second, etc.) and `i` is `j` minus one. In addition, the suffix "` -r`" is added to the above string for each attribute that must be sorted in descending order.

Finally, the sorted file must be printed, which is done by a simple call to the auxiliary procedure `ListUnixFile`. In addition, the two temporary unix files created are deleted by another call to procedure `UnixCommand` passing the appropriate unix remove command as a parameter.

8.2.4 Type of the primary key attributes

Another problem that occurred regards the type of the attributes in the primary key of relations. In the actual implementation of the DBPL system, attributes of type `REAL` cannot be (part of) the primary key of relations.

The proposed solution to this problem is to adopt an artificial surrogate primary key for the relation, already in the specification level.

Another possible solution would be to store the values of the attribute as integers. These would then be converted to the corresponding real numbers by an appropriate division operation, every time their values were needed for use in computations.

For instance, suppose an attribute `Salary` of type `REAL` (with two decimal digits) were part of the primary key of a relation. Then, it would have to be stored as an integer. Whenever its value in a given tuple `t` were needed for a computation, the expression `FLOAT (t.Salary) / 100` would be written instead of the usual `t.Salary`.

8.3 Current status of the implementation

This section describes the current status of the implementation of the prototype tool. It summarizes what has already been implemented and what remains to be done in order to generate the DBPL code corresponding to the complete example specification. It also addresses the effort that would be required to generate code for another database system and language, and presents a number of snapshots of the windows of the prototype.

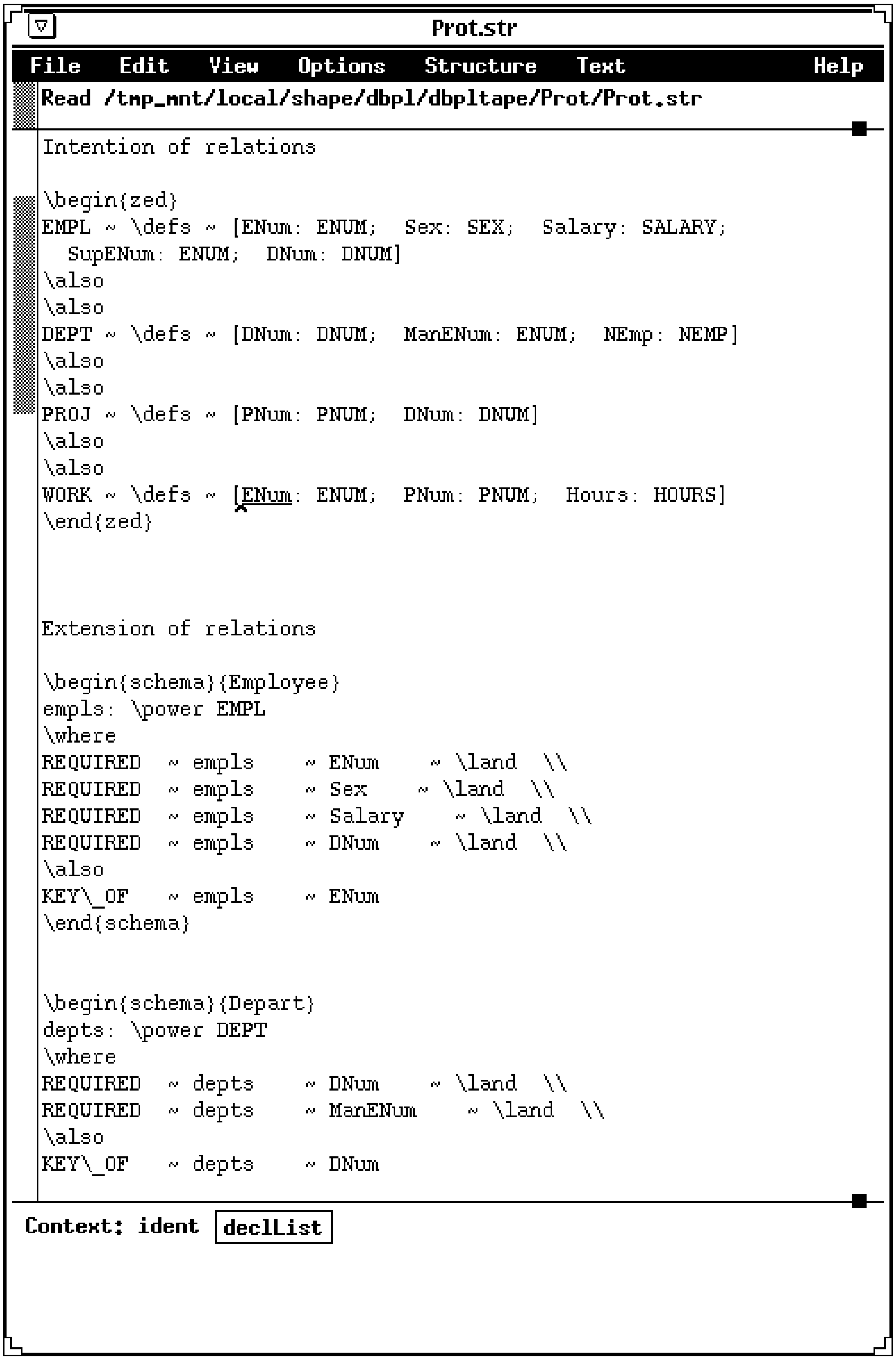
Regarding the status of the prototype tool, the syntactic editor currently accepts all the correct specifications referring to the structure part of the database and to most of the simple operations. In particular, the foreign key compensating actions for deletes have also been implemented. In addition, the tool enforces the correct syntax of the method at selected parts of the specifications. The definitions of the views for generating the DBPL code corresponding to all these features have also been written. Nevertheless, the specification of more complicated transactions and the error handling schemas have not been written yet.

In respect of a possible implementation of the mapping to generate programs to be run in another database system, the effort required would depend on a number of points. If the aim were to implement another prototype, the effort should be reasonably small because the syntactic editor is already built. Hence, the only difficulty would be to adapt the mapping to consider the limitations imposed by the chosen system, since writing the SSL code for the views is fairy simple.

On the other hand, to construct a production tool for DBPL, or even for some other system, would require a bigger effort because the syntactic editor would have to go through a major revision to be able to enforce all the features of the method.

Now, to give a better idea of what the prototype tool looks like, snapshots of a number of screens are provided. For instance, Figures 8.1 and 8.2 present consecutive snapshots of the specification window. These include the intention and extension of the relations as well as part of the database state schema of the company database example introduced in Chapter 6.

The generated DBPL implementation code corresponding to these specifications is presented in Figure 8.3, which is a snapshot of the `TYPES_D` view. This view allows for the generation of the definition module that contains all the global types. The snapshot shows the types of the intentions and extensions of all the relations, and this includes the primary keys of the relations.




```
Intention of relations

\begin{zed}
EMPL ~ \defs ~ [ENum: ENUM;  Sex: SEX;  Salary: SALARY;
  SupENum: ENUM;  DNum: DNUM]
\also
\also
DEPT ~ \defs ~ [DNum: DNUM;  ManENum: ENUM;  NEmp: NEMP]
\also
\also
PROJ ~ \defs ~ [PNum: PNUM;  DNum: DNUM]
\also
\also
WORK ~ \defs ~ [ENum: ENUM;  PNum: PNUM;  Hours: HOURS]
\end{zed}



Extension of relations

\begin{schema}{Employee}
empls: \power EMPL
\where
REQUIRED  ~ empls   ~ ENum    ~ \land  \\
REQUIRED  ~ empls   ~ Sex    ~ \land  \\
REQUIRED  ~ empls   ~ Salary    ~ \land  \\
REQUIRED  ~ empls   ~ DNum    ~ \land  \\
\also
KEY\_OF   ~ empls   ~ ENum
\end{schema}


\begin{schema}{Depart}
depts: \power DEPT
\where
REQUIRED  ~ depts   ~ DNum    ~ \land  \\
REQUIRED  ~ depts   ~ ManENum    ~ \land  \\
\also
KEY\_OF   ~ depts   ~ DNum
```

Figure 8.1: The prototype - specification window - part 1

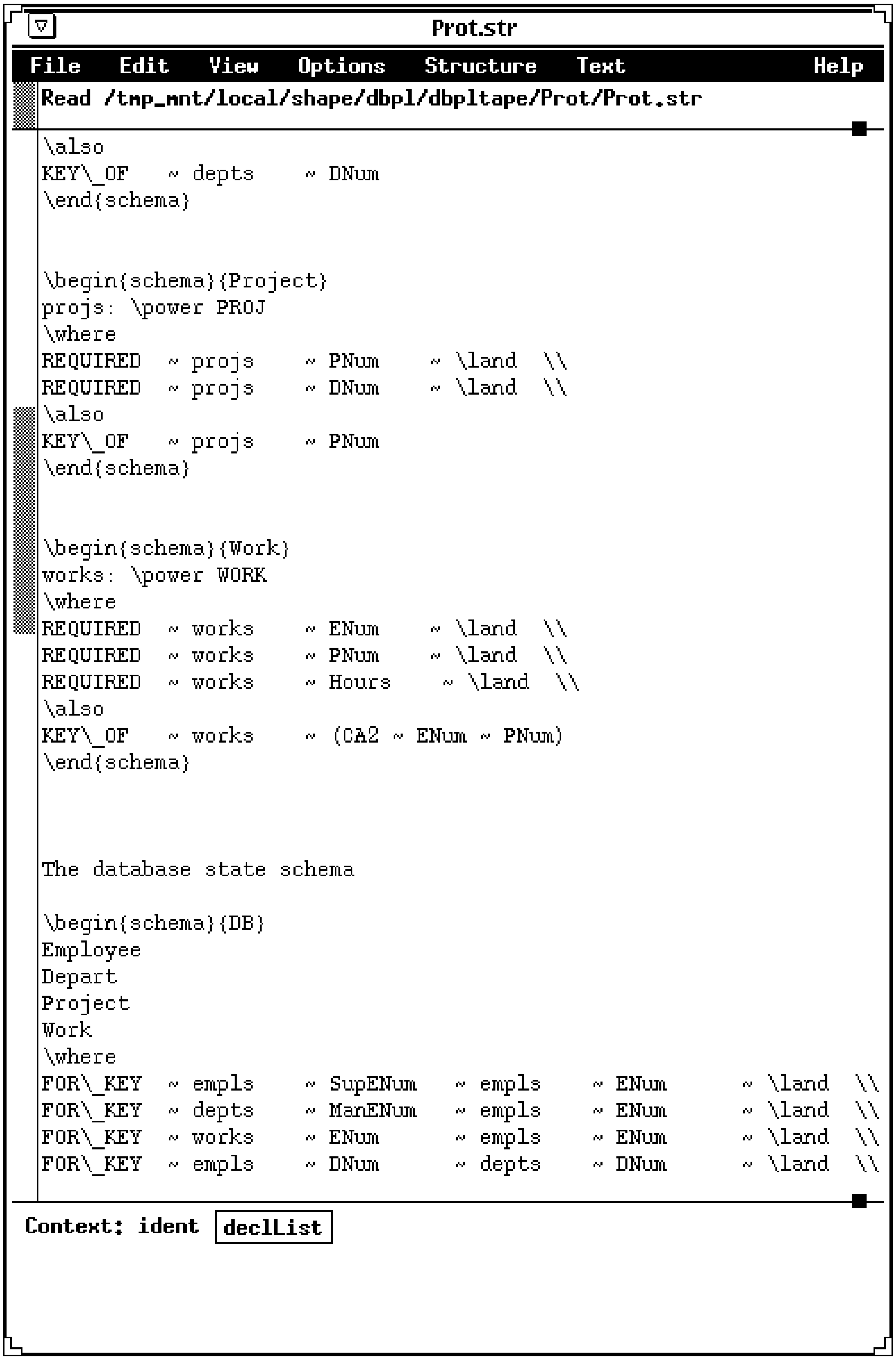


Figure 8.2: The prototype - specification window - part 2

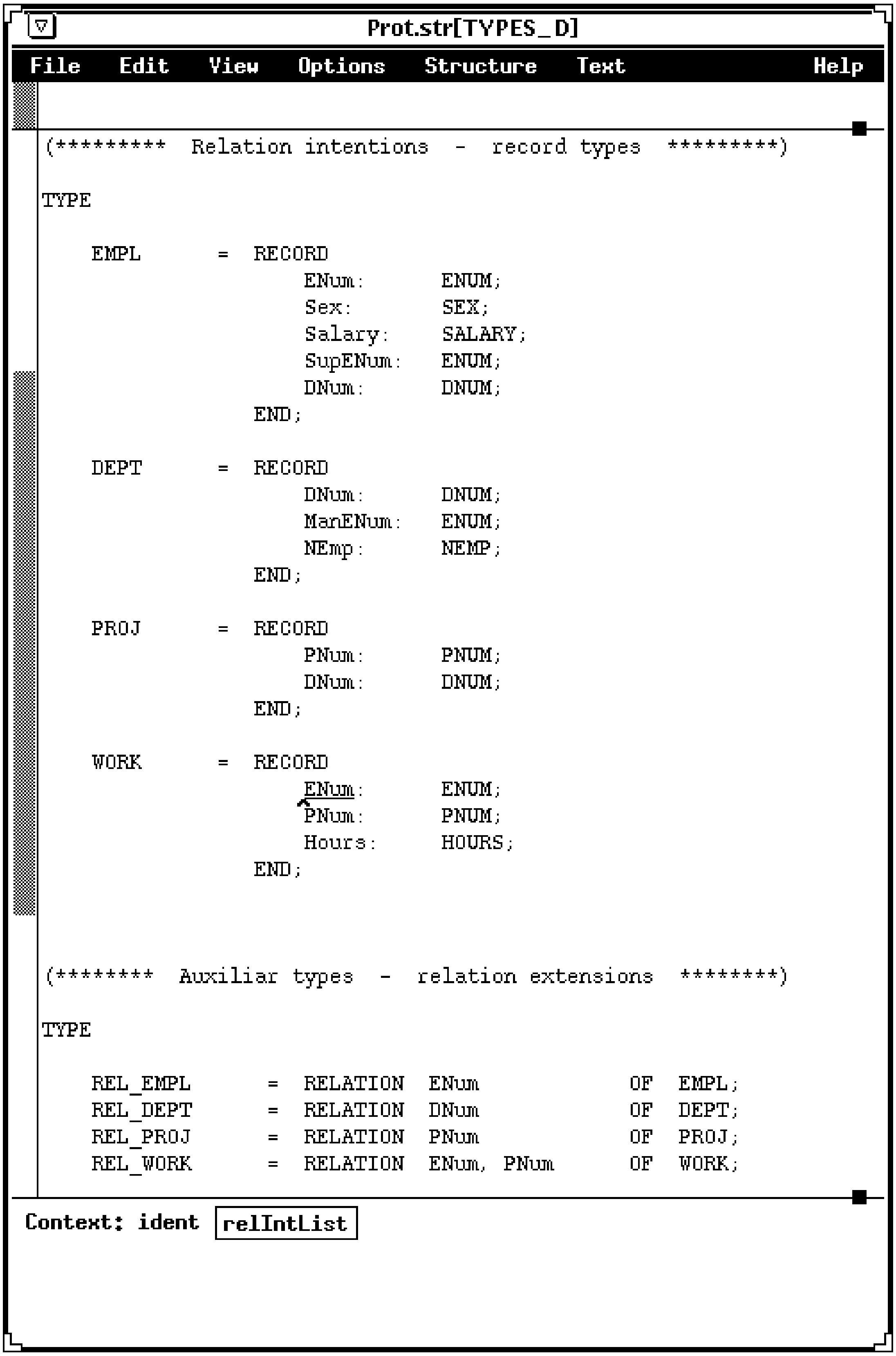


Figure 8.3: The prototype - DBPL database structure window

8.4 Prototyping with the synthesizer generator

This section provides a quick introduction to the *Synthesizer Generator*. Emphasis is given on the basic features needed to generate syntactic editors and on the capabilities of the system.

Some detail on the syntax of the *Synthesizer Specification Language* (SSL) is also provided. For specific details on the syntax of SSL it is better to look at the selected SSL code presented in Appendix B.

Basically, the user provides a specification written in SSL describing a language and the system creates a syntactic editor for the language.

In a sense, SSL is in fact a set of specification languages integrated in a single language. These sublanguages provide constructs to define the abstract syntax, context-sensitive relationships, and the input and output display syntax of the target language in a single specification language.

8.4.1 The abstract syntax

The abstract syntax is the most important part of a specification to generate a syntactic editor using the synthesizer generator. Even though it is not the most extensive part, it is probably the one that requires most attention (design) because most other parts of the specification depend on the abstract syntax.

The abstract syntax specification is essentially a context free specification of the target language (or method) underlying structure which will be used by the synthesizer generator to implement a syntactic editor.

The abstract syntax specification is formed by a root declaration, which gives the phylum name of the root of the tree (`root phylum;`), and a number of productions (equations) following the pattern

```
phylum :  production  ( sub-phyla );
```

where `phylum` is a node in the tree, `production` is the name of a possible production of the language syntax and `sub-phyla` is a list of phylum names separated by spaces. For each non-terminal phylum in the right hand side of a production there must be at least one equation describing it further.

Also, several productions of a given phylum can be written together, as shown below, which should improve the readability of the specifications.

```
phylum :  production  ( sub-phyla ),
       |  production  ( sub-phyla ),
       |  ...
       |  production  ( sub-phyla );
```

Finally, there are three special declarations which declare phyla as being optional (`optional phylum;`), list (`list phylum;`) and optional list (`optional list phylum;`) respectively and should precede their equations. For more details on these please refer to the selected SSL code given in Appendix B.

8.4.2 Unparsing rules

The unparsing rules are the part of the specification used to define how "programs" written using the syntactic editor are to be displayed on the screen.

For each production of the abstract syntax the user specifies the display format by adding constants and tabulations (indentation and newlines) to the values of its sub-phyla, in addition to whether the phyla can be text edited or not, and also which sub-phyla can be selected and which cannot.

The specification of an unparsing rule follows the pattern below, where `select-flag` specifies whether the phylum can be selected (`@`) or not (`^`), `text-edit-flag` specifies whether the phylum can be text edited (`::=`) or not (`:`), and `display` defines how the phylum is displayed.

```
phylum :  production  [ select-flag  text-edit-flag  display ];
```

The `display` part of the unparsing rules combines constants, the tabulation characters for newline (`%n`) and indentation (`%t` and `%b`), and the value of attributes together with a `select-flag` for each sub-phylum. Starting from the left, each `select-flag` is associated with a `phylum` and defines where it will be displayed.

Finally, since there is usually one unparsing rule for each production of the abstract syntax, they can be written together as follows:

```
phylum :  production  ( sub-phyla )
                      [ select-flag  text-edit-flag  display ];
```

8.4.3 Template transformations

A template transformation is a facility for creating commands that change the structure of documents and allow navigation downwards in the syntactic tree. In other words, these commands transform a given node in the tree into one of its syntactic subtrees.

When used together with the X Windows system, the synthesizer Generator will create a button for each template transformation. This facility enables the activation of these transformations to be done by clicking the buttons with the mouse.

The syntax for defining a template transformation is usually given by:

```
transform  phylum  on  "temp-name"  pattern :  production;
```

where `phylum` is the name of the node to be transformed, `temp-name` is the string naming the corresponding button, `pattern` is usually `<phylum>`, and `production` is the name of a production of the phylum followed, when appropriate, by its parameters (sub-phyla) within parentheses. The parameters are usually listed like patterns i.e., they are written as `<sub-phylum>`.

Optionally, the template transformations can also include a `when-clause` with a boolean condition to restrict the application of certain transformations. In these cases, the `pattern` is usually a pattern variable which is used within the `when-clause` to access the phylum attributes.

8.4.4 The use of attributes

Attributes are variables used to pass information to other nodes of the tree. The use of attributes allows for the implementation of context-sensitive checks, i.e., checks which depend on more than one node.

More generally, attributes may contain arbitrary auxiliary information and the collection of all attributes constitutes a database that can be used to present information on the screen and to control the editing process.

The values given to the attributes are defined by attribute equations which are usually written for each production of the phyla. Before an attribute is used it is often necessary to declare it. An attribute declaration is written as:

```
kind  phylum  name;
```

where `kind` is the kind of attribute being defined, `phylum` is the "type" of information it stores, and `name` is its name.

The synthesizer generator supports four kinds of attributes: synthesized (`syn`), inherited (`inh`), local (`local`), and non-terminal attributes. Their main characteristics are presented below.

Synthesized attributes

- They are used to pass information to nodes *up* in the syntactic tree, i.e., they pass information from the node to the parent node.

- Their values are specified in the productions of the node where they are defined and are accessible in the parent nodes, i.e., the nodes one level up in the tree.

- Equations defining values for synthesized attributes always refer to the phylum (node) in the *left* hand side of the production.

- References to a synthesized attribute `a` of a phylum `p` are written as `p.a`. The phylum on the left hand side of the production can also be referred to by `$$`. If there are two or more occurrences of a given phylum `p` in the production, then `p$1`, `p$2`, etc. are used to refer to them and are taken from left to right.

Inherited attributes

- They are used to pass information to nodes *down* in the syntactic tree, i.e., they pass information from the parent node to the node.

- Their values are specified in the productions of the parent node (one level up) and are accessible in the nodes where they are defined.

- Equations defining values for inherited attributes always refer to one of the phyla (nodes) in the *right* hand side of the production.

- References to inherited attributes are similar to the case of synthesized attributes.

Local attributes

- Their definitions and values are local to one the productions of the phylum only, rather than the phylum. Their names are, therefore, enough to refer to them inside the production.
- Local attributes are the best choice to implement error messages since these tend to be different from one production to another.
- A useful way of implementing global checks is to define a local attribute at the root production of the tree and use a facility called *upward remote attribute set* to access it from all nodes of the tree. Such an attribute is accessed by the expression `{p.a}`, where `p` is the production name[1] and `a` is the attribute name.

Non-terminal attributes

- Every non-terminal in a production may be seen as an attribute and, thus, it may be used in attribute equations. No explicit declaration is needed.
- References to non-terminal attributes are similar to the cases of synthesized and inherited attributes except for no attribute name is given.
- If an equation is written giving a value to a non-terminal phylum attribute, i.e., its value is derived from the values of other attributes, the effect is to make the node read only.

8.4.5 The concrete syntax for text editing

The synthesizer generator also supports a *text* editing facility, in addition to the *structure* editing. The main reason is that structure editing can be too slow sometimes, but this facility is also useful to read and syntax-check existing documents using the editor.

In order to allow the text editing and re-editing of certain nodes (phyla) the user needs to follow a number of extra steps.

Firstly, for each phylum where text editing will be allowed there must be a corresponding phylum declaration for its *concrete* syntax. This phylum must have at least one attribute, usually synthesized, which is used to specify how the concrete syntax is translated to the abstract syntax. Their definitions are presented below:

```
concrete-phylum {syn abstract-phylum name; };
```

Then, for each concrete phylum created, an explicit entry declaration is written. This declaration allows for the association of the attribute with the corresponding phylum of the abstract syntax and is given by:

```
abstract-phylum ~ concrete-phylum.name;
```

[1] In fact synthesized and inherited attributes can also be accessed by an upward remote attribute set and in this case `p` is the phylum name.

Finally, for each different pattern of text editing allowed in each concrete phylum there must be an attribute equation giving the value of the attribute, i.e., the actual translation to terms of the abstract syntax, as given below.

```
concrete-phylum  ::=  (pattern)  { $$.name = abstract-equation; };
```

Notice that, in the above equation, `pattern` can be any combination of concrete phyla, lexicals, and/or single characters (e.g. `'c'`); `$$` is a short-hand for the left-hand side phylum in attribute equations (i.e. `concrete-phylum`); and `abstract-equation` is any equation of the abstract syntax such that it has the `abstract-phylum` type.

As in other parts of the specification, more than one pattern for text editing of the same phylum can be written together as follows:

```
concrete-phylum  ::=  (pattern)  { $$.name = abstract-equation; },
                   |  (pattern)  { $$.name = abstract-equation; },
                   |  ...
                   |  (pattern)  { $$.name = abstract-equation; };
```

8.4.6 Using views to generate code

One of the most distinct features of the synthesizer generator is to support the definition of different views for the display of different information. When used together with the X Windows system, different views can be presented at the same time using separate windows.

One of the advantages of this facility is to allow the users to create views which displays the information using different levels of abstraction. For example, in addition to the main view, users might have a view that omit the comments, another that omit the errors, and so on.

However, the main advantage of the view facility is to permit the generation of alternative display schemes which are automatically updated by the system. These make it very easy to generate code written in a different language as long as the mapping is well defined.

Basically, the user gives a different name to each different view by means of a view declaration (`view name;`) and defines a new set of unparsing rules for each view, where the name of the view is written before the `select-flag` as follows:

```
phylum :  production [ view  select-flag  text-edit-flag  display ];
```

Naturally, it is still necessary to design the mapping between the languages and, thus, a number of different types of information will usually be needed in order to make the translation possible, i.e., a number of extra attributes are required.

The most obvious application of this facility is to generate the equivalent object code from source code written in a high level programming language. It is obviously also possible to generate source code in another high-level language.

As already mentioned, the view facility of the Synthesizer Generator was exploited to automatically generate relational database programs written in DBPL.

8.4.7 Other features

The Synthesizer Generator also provides a number of other features which are not going to be described here. These include conditional, let, and binding expressions; comparison and logical operators; conversion of terms from and to strings; glyphs; etc. Others like for example Lexical declarations and functions are briefly described below.

Lexical declarations

The synthesizer generator system also allows the user to define patterns for the name of identifiers by means of regular expressions called lexical declarations. In fact, there must be one lexical declaration for each keyword or multi-character token of the language.

Lexical declarations must be declared before they can be used. Thus, they are usually the first part of every SSL specification. Also, the order of the declarations is important since names will be matched by the first lexical declaration to be satisfied.

Finally, lexical declarations can also be used to give names to text editing commands. This contributes to improving the readability of the patterns in the specification of the concrete input syntax.

Functions

The system also allows the user to define functions similarly to the way it is done in high level programming languages like for example Pascal.

Just like in the programming world, the main reasons to use this facility are (1) to reuse parts of the specifications which are basically the same, and (2) to structure the specifications and make them easier to read.

8.5 Conclusion

This Chapter provided an outline of the experiment that was carried out and involved the implementation of a prototype tool to support the method (Chapter 4) and instantiate the mapping (Chapter 7) for the DBPL system. This concludes the principal part of this thesis.

Next is Chapter 9, which closes the main body of the thesis and presents the overall conclusions reached by the research.

Chapter 9

Conclusion

In this thesis, an extension aimed at enhancing the traditional database design process was proposed. It adds a number of phases to the traditional process and aims to formalize the development of (relational) database applications (transactions).

In the perfect world, applications should be formally specified and modularization techniques used, when necessary, to make the specifications easier to understand. Also, reasoning and/or refinement techniques could be applied before the implementation is actually developed.

This work has addressed the problems of specifying relational database applications and of deriving relational database programs directly from the specifications. However, the use of modularization and formal reasoning techniques have only been investigated superficially and are not included in the thesis. Furthermore, the use of refinement techniques has not been addressed at all.

Specifically, most of the thesis was devoted to the presentation of the following:

- A well defined set of rules for the formal specification of relational databases and their applications using Z. Throughout the thesis, this set of rules is referred to as "the method".

- A partial set of rules for the generation of database applications directly from formal specifications written in Z according to the method. These rules are called the mapping.

- The description of a prototype syntactic tool which aims to support and enforce a reasonably large subset of the method. In addition, the prototype instantiates and partially implements the mapping for a particular RDBMS.

9.1 The method

An important first part of this research was the development of a method for the formal specification of relational database applications. The method provided a formal starting point for the investigation of all other aspects of the work. Therefore, it was vital to improve it as much as possible before proceeding to investigate the other parts because a weak method would probably make the whole work fail.

The complete description of the method (last version) was introduced in Chapter 4, whereas Chapter 5 dealt with the formal definition of the generic operators used in the specifications prescribed by the method. A lengthy example specification using the method was presented in Chapter 6.

The method is aimed at formalizing the design of real relational database transactions and, so, it should help practitioners in the development of real world applications. In addition, the method is generic and may be the first step in the direction of the formal development of database applications and of specification standardization in this context. Moreover, it should improve the system documentation and the quality of the application programs which should contain fewer errors.

It is believed the method achieves the proposed objectives. Firstly, it provides a simple way of specifying relational database applications formally. Secondly, it is generic and may be the first step in the direction of the formal development of database applications and of specification standardization in this context. Thirdly, it deals not only with the correct behaviour of the operations, but also with the specification of errors. Finally, because of its ease-of-use, it may be applied to the specification and documentation of relational database systems.

Also, the method is specifically intended to be used in the formal specification of relational database applications. Thus, it should lead to specifications which are amenable to implementation using RDBMSs. Even so, specifications written according to the method do not address issues of system performance or difficulty of implementation for any particular RDBMS. On the contrary, it could possibly be used to specify systems which are to be implemented using either DBMSs based on other approaches, e.g. the inverted list approach [5, pp. 737–751], or even a file-based approach.

Note that the choice of Z in this work does not preclude using other model-oriented languages. This means that the method is generic and that different users may use different specification languages to specify their applications. In particular, a previous paper on this method [127] was written in Zc [128, 129], a strongly-typed Z-like language, with minor modifications only.

9.2 The mapping

The (semi-)automatic generation of relational database applications through a simple translation process directly from formal specifications that result from using the method (reification) was also subject of investigation.

More specifically, this thesis addressed the problems involved in the derivation of appropriate relational database programs directly from specifications written according to the method.

The mapping introduced in Chapter 7 described, for a comprehensive subset of the method, what the target implementation code should look like, without binding it to any particular database system or language. However, most examples were written in DBPL [7, 8, 9], a RDBMS built in the University of Hamburg, because this was the target system used to build the prototype.

It was claimed most previous approaches to the derivation of databases programs had not properly addressed the problem, because the problem was either kept too general, without being restricted to any particular database model, or greatly simplified, by not addressing the specification of the database constraints and/or more complicated transactions. The work described here is restricted to the relational database model and addresses all possible constraints as well as generic transactions.

9.3 The prototype

An intrinsic part of this thesis is a prototype tool which was built to support the method and implement the mapping. Its components are a syntactic editor for the method and a built-in tool which translates the specifications to database commands.

Since the tool is only a prototype, it does not support the full method. Nevertheless, a comprehensive subset of all correct specifications is accepted by the syntactic editor. The implementation of the mapping, which generates relational database applications to be run in the DBPL system, is also partial.

The prototype was developed using the Synthesizer Generator [10, 11], which is a powerful tool for implementing language-based editors. It allows for the generation of syntactic editors fairly quickly, as long as the syntax and semantics of the target language are well defined.

The effort to learn the basic features of the system was also fairly small. It took about two weeks to get the first specification running and another two weeks to experiment with most of the features of the system.

The Synthesizer Generator helped to create appropriate support to using the method for the specification of relational database applications as well as to deriving relational database programs from the specifications.

In particular, the view facility of the Synthesizer Generator was used to automatically generate parts of relational database programs written for a given RDBMS and 4GL (or query/host language), namely the DBPL system. The syntactic editor that supports and enforces the method is a bonus. Eventually, the process could be instantiated for a RDBMS offering SQL [21] as its data-sublanguage, e.g. DB2 [124].

As far as I can see, the main challenge was to come up with a good design for instantiating the general mapping to the particular RDBMS chosen (DBPL) within the time available. Writing the SSL specification for the syntactic editor and using the view facility to generate database programs per se were reasonably straightforward.

9.4 The specification of database transactions in Z

One of the conclusions of this thesis is that the choice of Z as the formal language for the specification of relational database transactions was an appropriate decision.

In the main, the specification method uses only standard Z [3]. Still, most aspects of the method are clear and simple and are defined using a suitable level of abstraction.

The extensions to Z used or suggested in this thesis were kept to a minimum. These are presented below.

The tuple type: This extension was proposed by van Diepen and van Hee [116] and is aimed at preventing the users from specifying predicates as part of a type definition. The method uses it for defining the intention of relations (rule D2).

The expression using ΞDB: This refers to an artifice aimed at achieving a more clear way of saying what variables of the state schema are going to be changed by update operations. In standard Z, these expressions are syntactically correct but they are only accepted in the predicate part of the schemas. The method uses such expressions in the specification of all update operations (rules B5–B10).

The *But* operator: This extension makes possible the modification of one or more attributes of variables of a tuple type, preserving the values of the other attributes of the tuple. This extension was first needed in the specification of update operations (rule B9) and it is particularly useful because the resulting update operations need not be changed if new attributes are added to the corresponding relations. However, the method also uses this operator for defining views based on updates (rule A5.4) and in the formal specification of the *UPDATE* operator (Section 5.4).

The schema piping ($>>$): The version of the schema piping used in this thesis allows for the output *and* primed state variables (*all results*) of the first schema to be matched against the input *and* unprimed state variables of the second schema, respectively. In standard Z, the schema piping does *not* match the state variables. The method used the schema piping for the specification of the correct behaviour of transactions (rule A1).

9.5 Further work

One natural extension to the method refers to the modularization of the specifications that result from the application of the method. This can be achieved by using the modularization structures *Document* and *Chapter* of Zc, also proposed for incorporation in Z [41]. These structures have been used to modularize the specification of real life systems, such as a Student Records Control System [1] and the Interface of a Hypertext System [130], with good results.

The idea is to split the specification of systems (documents) into several modules (chapters) based on the connections between objects. Specifically, the specification of complex relational databases should be split into several Chapters based on the connections between the relations, i.e. the foreign keys. This would result in a specification which is modular and, therefore, easier to understand. The problems that may arise from such a separation and a detailed explanation of what is needed to avoid these problems are planned to be investigated in the near future.

The full treatment for error handling could also be subject of future work. The main objective would be to try to identify, for each of the operations prescribed by the method, all the possible kinds of constraints that might be violated. Moreover, it might even be possible to identify specific equations in the simplified precondition of the transactions that correspond to certain operation and constraint pairs. The results of such investigation could then lead to a more straight-forward way of developing the precondition of database transactions written according to the method. The automatic generation of parts of the predicate of the error schemas associated with the transactions might also be feasible.

The application of reasoning techniques to specifications written according to the method could also be investigated. The approach would be to try and come up with standard theorems about common properties of such specifications and prove them so that the users would be discharged from proving them again. The main benefit would be to formally prove that the method is sound and that transactions specified according to method do indeed maintain the consistency of the database.

One possible approach to prove the theorems about relational specifications written according to the method could be to generate, using another view in the prototype tool, a version of the specifications written in the formal specification language adopted by some other system supporting theorem proving, e.g. ADABTPL [101]. The theorems could then be checked by the theorem prover, either by translating the theorems written in Z or by writing them already in the appropriate language.

Another possibility is to adapt the generic mapping to generate code for another relational database system, possibly a system providing SQL as the query language. In particular, it would be really interesting if such a project could be developed in partnership with an existing company and were targeted at a DBMS that is actually used in the development of real, large-scale relational database applications.

There are a number of other directions in which this research could advance. One of them would be to work on guidelines aimed at maximizing the reuse of specifications of sub-transactions. Even if a complete investigation of this problem in not carried out, it should be possible to write a number of guidelines based on the experience gained with the specification of the company database example.

Another way to proceed would be to use a controlled experiment to compare the specification of simple relational database applications written using the method against others written without the method. To be meaningful, such an experiment would have to be carried out using several groups of people with different backgrounds. The results of the experiment would probably enable an easier identification of the strengths and weaknesses of the method and, thus, help to improve it.

Finally, it is possible that the method can be adapted for developing object-oriented database [131] applications and this could also be subject to investigation.

9.6 Final remark

To conclude, it is believed the mapping process described in this thesis, as well as its actual prototype implementation for DBPL (and indeed for most RDBMSs), are neither too easy nor too difficult. Moreover, it is claimed this work provides evidence that the application of formal techniques in the development of real life software is feasible. Even though there is no formal proof that the mapping retains all the properties of the method, the well-defined semantics of the relational model together with extensive testing of the prototype suggests this is indeed the case.

Appendix A

Simplification of a precondition

In this appendix, the precondition of transaction $Move\_empls\_proj\_Ok$ is to be written and simplified, since only the simplified precondition was introduced in the specification of the example (Chapter 6, Subsection 6.6.2, page 78).

Before this, the simplified specification of the transaction is presented again in order to make the simplification process easier to follow.

$$
\begin{array}{l}
\textit{Move\_empls\_proj\_Ok} \\
\hline
\Delta DB \\
p1?, p2? : PNUM \\
proj? : PROJ \\
\Xi DB \setminus (projs, works) \\
\hline
proj?.PNum = p2? \;\wedge \\
\exists\, pj : projs \bullet pj.PNum = p1? \;\wedge \\
projs' = \textbf{if } \neg\,(\exists\, pj : projs \bullet pj.PNum = p2?) \\
\qquad\qquad \textbf{then } DELETE\ (projs \cup proj?)\ PNum\ p1? \\
\qquad\qquad \textbf{else } DELETE\ projs\ PNum\ p1? \;\wedge \\
works' = \{\, w : works \bullet \textbf{if } w.PNum = p1? \\
\qquad\qquad\qquad\qquad \textbf{then } w \setminus (PNum = p2?) \\
\qquad\qquad\qquad\qquad \textbf{else } w \,\} \\
\hline
\end{array}
$$

As usual, the predicate of schema $pre\ Move\_empls\_proj\_Ok$ is given by:

$$\exists\, DB' \bullet Move\_empls\_proj\_Ok$$

Expanding the above schema expression will result in:

$$
\begin{array}{l}
\exists\, DB' \bullet \\
\quad \Delta DB \;\wedge \\
\quad \Xi DB \setminus (projs, works) \;\wedge
\end{array}
$$

$$proj?.PNum = p2? \ \wedge$$

$$\exists\, pj : projs \ \bullet \ pj.PNum = p1? \ \wedge$$

$$\begin{aligned} projs' \ = \ & \textbf{if} \ \neg\, (\exists\, pj : projs \ \bullet \ pj.PNum = p2?) \\ & \textbf{then} \ DELETE\ (projs \cup proj?)\ PNum\ p1? \\ & \textbf{else} \ DELETE\ projs\ PNum\ p1? \ \wedge \end{aligned}$$

$$\begin{aligned} works' \ = \ \{\, w : works \ \bullet \ & \textbf{if} \ w.PNum = p1? \\ & \textbf{then} \ w \setminus (PNum = p2?) \\ & \textbf{else} \ w \,\} \end{aligned}$$

Notice that the third and fourth equations are independent of DB' and, thus, can be moved outside the existential quantifier. Although schema ΔDB includes all database constraints, only the ones involving the changed relations ($projs'$ and $works'$) are to be considered, because the database is always in a valid state before any operation. Also, the unchanged relations $empls'$ and $depts'$ are taken out of the outer existential quantifier and references to these are changed to $empls$ and $depts$, respectively.

After rearranging the predicate to group equations by relation, in order to simplify its understanding, the above precondition expands to:

$$proj?.PNum = p2? \ \wedge$$

$$\exists\, pj : projs \ \bullet \ pj.PNum = p1? \ \wedge$$

$$\begin{aligned} & \exists\, projs' : \mathbb{P}\, PROJ;\ works' : \mathbb{P}\, WORK \ \bullet \\ & \quad REQUIRED\ projs'\ PNum \ \wedge \\ & \quad REQUIRED\ projs'\ DNum \ \wedge \\ & \quad KEY\_OF\ projs'\ PNum \ \wedge \\ & \quad FOR\_KEY\ projs'\ DNum \ \ depts\ DNum \ \wedge \\ & \quad projs' \ = \ \textbf{if} \ \neg\, (\exists\, pj : projs \ \bullet \ pj.PNum = p2?) \\ & \qquad\qquad \textbf{then} \ DELETE\ (projs \cup proj?)\ PNum\ p1? \\ & \qquad\qquad \textbf{else} \ DELETE\ projs\ PNum\ p1? \ \wedge \\ & \quad REQUIRED\ works'\ ENum \ \wedge \\ & \quad REQUIRED\ works'\ PNum \ \wedge \\ & \quad REQUIRED\ works'\ Hours \ \wedge \\ & \quad KEY\_OF\ works'\ (CA2\ ENum\ PNum) \ \wedge \\ & \quad FOR\_KEY\ works'\ ENum \ \ empls\ ENum \ \wedge \\ & \quad FOR\_KEY\ works'\ PNum \ \ projs'\ PNum \ \wedge \\ & \quad \forall\, w : works' \ \bullet \ w.Hours \geq 4 \ \wedge \\ & \quad \forall\, e : empls \ \bullet \ (\exists\, w : works' \ \bullet \ w.ENum = e.ENum) \ \wedge \\ & \quad works' \ = \ \{\, w : works \ \bullet \ \textbf{if} \ w.PNum = p1? \\ & \qquad\qquad\qquad\qquad \textbf{then} \ w \setminus (PNum = p2?) \\ & \qquad\qquad\qquad\qquad \textbf{else} \ w \,\} \end{aligned}$$

Regarding the constraints involving relation $projs'$, required attribute constraints, primary key constraints, and foreign key constraints are affected only by insertions and updates. Therefore, these can only be affected if project $p2?$ doesn't exist. For this reason, an appropriate implication is added, even though it is not strictly needed at this point, since it will make the next step easier. However, because $p2?$ is guaranteed to be the primary key $PNum$ of the inserted tuple $proj?$, the primary key constraint cannot be violated and is omitted.

Regarding the constraints involving relation $works'$, the following assertions are true: no tuple is inserted or deleted, and attributes $ENum$ and $Hours$ are not changed. So, the corresponding required attribute constraints cannot be violated and are omitted. For the same reason, the first of the foreign key constraints and the other two attribute constraints (universal quantifiers) cannot be violated either and are omitted as well.

All these changes simplify the precondition to the following:

$$
\begin{array}{l}
proj?.PNum = p2? \;\wedge \\[4pt]
\exists\, pj : projs \;\bullet\; pj.PNum = p1? \;\wedge \\[4pt]
\exists\, projs' : \mathbb{P}\, PROJ;\; works' : \mathbb{P}\, WORK \;\bullet \\
\quad \neg\, (\exists\, pj : projs \;\bullet\; pj.PNum = p2?) \;\Rightarrow \\
\qquad\qquad (\; REQUIRED\ projs'\ PNum \;\wedge \\
\qquad\qquad\;\; REQUIRED\ projs'\ DNum \;\wedge \\
\qquad\qquad\;\; FOR\_KEY\ projs'\ DNum\ \ depts\ DNum\ ) \;\wedge \\[4pt]
\quad projs' \;=\; \textbf{if}\;\; \neg\, (\exists\, pj : projs \;\bullet\; pj.PNum = p2?) \\
\qquad\qquad\;\; \textbf{then}\;\; DELETE\ (projs \cup proj?)\ PNum\ p1? \\
\qquad\qquad\;\; \textbf{else}\;\; DELETE\ projs\ PNum\ p1? \;\;\wedge \\[4pt]
\quad REQUIRED\ works'\ PNum \;\wedge \\
\quad KEY\_OF\ works'\ (CA2\ ENum\ PNum) \;\wedge \\
\quad FOR\_KEY\ works'\ PNum\ \ projs'\ PNum \;\wedge \\[4pt]
\quad works' \;=\; \{\; w : works \;\bullet\; \textbf{if}\;\; w.PNum = p1? \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\;\; w \setminus (PNum = p2?) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\;\; w\; \}
\end{array}
$$

Now, since $\exists\, x \bullet (x = y \wedge P(x))$ is equivalent to $P(y)$, $projs'$ and all related equations can be removed from the existential quantifier. Notice that the first conditional statement (**if-then-else**) is regarded as an equation of the form $projs' = y$. This is possible because the constraints that can only be affected if project $p2?$ doesn't exist have already been enforced by an explicit implication in the previous step.

In addition, because the database is always in a valid state before any operation, the required and foreign key constraints can be simplified to check only the inserted or changed tuples. Also, the primary key operator is substituted for its corresponding predicates. The precondition resulting from these changes is presented below.

$$
\begin{array}{l}
proj?.PNum = p2? \;\wedge \\[4pt]
\exists\, pj : projs \;\bullet\; pj.PNum = p1? \;\wedge
\end{array}
$$

$$
\begin{array}{l}
\neg\,(\exists\, pj : projs \;\bullet\; pj.PNum = p2?) \;\Rightarrow \\
\qquad\quad (\; proj?.PNum \neq NULLNAT \;\;\wedge \\
\qquad\quad\;\; proj?.DNum \neq NULLNAT \;\;\wedge \\
\qquad\quad\;\; \exists\, dp : depts \;\bullet\; dp.DNum = proj?.DNum \;) \;\;\wedge \\
\exists\, works' : \mathbb{P}\, WORK \;\bullet \\
\quad p2? \neq NULLNAT \;\;\wedge \\
\quad \neg\,(\;\exists\, w1, w2 : works' \;\bullet\; w1.ENum = w2.ENum \;\;\wedge \\
\qquad\qquad\qquad\qquad\qquad\quad\;\; w1.PNum = w2.PNum \;) \;\;\wedge \\
\quad works' \;=\; \{\; w : works \;\bullet\; \mathbf{if}\;\; w.PNum = p1? \\
\qquad\qquad\qquad\qquad\qquad\quad\; \mathbf{then}\; w \setminus (PNum = p2?) \\
\qquad\qquad\qquad\qquad\qquad\quad\; \mathbf{else}\;\; w \;\}
\end{array}
$$

Again, because $\exists\, x \;\bullet\; (x \;=\; y \;\wedge\; P(x))$ is equivalent to $P(y)$, the quantifier over $projs'$ can be removed. Also, the predicate $proj?.PNum \neq NULLNAT$ is equivalent to $p2? \neq NULLNAT$, because $proj?.PNum = p2?$. Since the former must hold if project $p2?$ does not exist and the latter must hold at all times, the former can be omitted.

$$
\begin{array}{l}
p2? \neq NULLNAT \;\;\wedge \\
proj?.PNum = p2? \;\;\wedge \\
\exists\, pj : projs \;\bullet\; pj.PNum = p1? \;\;\wedge \\
\neg\,(\exists\, pj : projs \;\bullet\; pj.PNum = p2?) \;\Rightarrow \\
\qquad\quad (\; proj?.DNum \neq NULLNAT \;\;\wedge \\
\qquad\quad\;\; \exists\, dp : depts \;\bullet\; dp.DNum = proj?.DNum \;) \;\;\wedge \\
\neg\,(\;\exists\, w1, w2 : \{\; w : works \;\bullet\; \mathbf{if}\;\; w.PNum = p1? \\
\qquad\qquad\qquad\qquad\qquad\qquad\;\; \mathbf{then}\; w \setminus (PNum = p2?) \\
\qquad\qquad\qquad\qquad\qquad\qquad\;\; \mathbf{else}\;\; w \;\} \;\bullet\; w1.ENum = w2.ENum \;\;\wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad w2.PNum = w2.PNum \;)
\end{array}
$$

Finally, because the database is always in a valid state before any operation, there is only one way the last constraint can be violated, which is when one of the changed tuples and one of the unchanged tuples have the same primary key. Assuming $w1$ stands for the changed tuple, the above predicate is equivalent to the one given below, which is exactly the simplified precondition of transaction *Move_empls_proj_Ok* presented in page 78.

$$
\begin{array}{l}
p2? \neq NULLNAT \;\;\wedge \\
proj?.PNum = p2? \;\;\wedge \\
\exists\, pj : projs \;\bullet\; pj.PNum = p1? \;\;\wedge \\
\neg\,(\exists\, pj : projs \;\bullet\; pj.PNum = p2?) \;\Rightarrow \\
\qquad\quad (\; proj?.DNum \neq NULLNAT \;\;\wedge \\
\qquad\quad\;\; \exists\, dp : depts \;\bullet\; dp.DNum = proj?.DNum \;) \;\;\wedge \\
\neg\,(\;\exists\, w1, w2 : works \;\bullet\; w1.ENum = w2.ENum \;\;\wedge \\
\qquad\qquad\qquad\qquad\qquad\quad w1.PNum = p1? \;\;\wedge\;\; w2.PNum = p2? \;)
\end{array}
$$

Appendix B

Selected SSL code

This appendix presents selected parts of the SSL code written to generate the prototype (discussed in Chapter 8). The aim here is to give a concrete idea of the structure of SSL specifications by introducing an example which contains all the syntax details.

Each of the following sections includes specifications that use a different feature of SSL. Most of the included specifications refer to the domains and the relations of the specification method in order to make it easier to understand the relationships among distinct parts of the SSL code.

Finally, no extra explanation is provided in this appendix. Thus, the reader may find it helpful to refer to the introduction to the synthesizer generator (Section 8.4) while reading the rest of this appendix.

B.1 Abstract syntax

```
root specification;

specification :  Spec  ( domDefList     relIntList     relExtList
                         state          basicOperList  transList );

list domDefList;                        /* List of domain definitions */
domDefList :  DomDefListNil   ( )
           |  DomDefListPair  ( domDef  domDefList );

domDef :  DomDef  ( ident  domExp );           /* Domain definition */

domExp :  EmptyDom  ( )                        /* Domain expression */
       |  IntDom, NatDom, RealDom, BoolDom, StrDom  ( )
       |  EnumDomExp   ( identList );

list relIntList;                      /* List of relation intentions */
relIntList :  RelIntListNil   ( )
           |  RelIntListPair  ( relInt  relIntList );

relInt :  RelInt  ( ident  declList );         /* Relation intention */
```

```
list relExtList;                        /* List of relation extensions */
relExtList :  RelExtListNil   ( )
           |  RelExtListPair  ( relExt  relExtList );

                                                /* Relation extension */
relExt :  RelExt  ( ident  ident  ident  identList  ident3  optConstr );


                  /* The state schemas: DB, DeltaDB, XiDB, and InitDB */
state :  State  ( ident      identList  forKeyList
                  optConstr  optConstr  identList  );


list forKeyList;                    /* List of foreign key constraints */
forKeyList :  ForKeyListNil   ( )
           |  ForKeyListPair  ( forKey  forKeyList );

forKey :  ForKey  ( ident  ident3  ident ); /* Foreign key constraint */


list declList;                               /* List of declarations */
declList :  DeclListNil   ( )
         |  DeclListPair  ( decl  declList );

decl :  Decl  ( ident  declExp );                      /* Declaration */


declExp :  EmptyDecl  ( )                   /* Declaration expression */
        |  IntDecl, NatDecl, RealDecl, BoolDecl, StrDecl  ( )
        |  IdentDeclExp  ( ident )
        |  PowerDeclExp  ( declExp )
        |  SeqDeclExp    ( declExp )
        |  CartProdExp   ( declExp  declExp )
        |  SetUnionExp   ( declExp  declExp );


list identList;                               /* List of identifiers */
identList :  IdentListNil   ( )
          |  IdentListPair  ( ident  identList );

ident :  IdentNull  ( )                         /* Normal identifiers */
      |  Ident  ( IDENT );

ident3 :  Ident3  ( ident )       /* Identifiers allowing composition */
       |  CA2     ( ident ident );
```

B.2 Attribute definitions

```
domDefList  { syn  identList  domNames; };        /* All domain names */

domDef  { syn  ident  name; };                    /* Name of the domain */

domExp  { syn  STR  equal; };      /* Equality sign for domains defs. */

                           /* Schema names of all relation intentions */
relIntList  { syn  identList  intNames; };

relInt  { syn  ident  name; };   /* Schema name of relation intention */

          /* Attributes of relExt aggregating all relation extensions */
relExtList  { syn  identList  schNames;
              syn  identList  relNames;
              syn  STR        relPKeys; };

relExt  { syn  ident  schName;   /* Schema name of relation extension */
          syn  ident  relName;      /* Relat. extension variable name */
          syn  STR    relPKey; };   /* Relation of Primary key type   */
                                    /*   formatted for TYPES-I view.  */

state  { syn  ident  name; };             /* Name of the state schema */

identList  { inh  STR  listSep; };      /* identifiers list separator */


specification :  Spec
    { local  identList  domNames;
      domNames = domDefList.domNames;

      local  relIntList  relInts;
      relInts = relIntList;

      local  identList  intNames;
      intNames = relIntList.intNames;

      local  relExtList  relExts;
      relExts = relExtList;

      local  identList  REschNames;
      REschNames = relExtList.schNames;

      local  identList  relNames;
      relNames = relExtList.relNames;
```

```
      local  ident  DBname;
      DBname = state.name;  };


domDefList :  DomDefListNil
    { $$.domNames = IdentListNil; };

domDefList :  DomDefListPair
    { $$.domNames = (domDef.name :: domDefList$2.domNames); };

domDef :  DomDef
    { local  STR  domError;
      domError = ( ident$1 == IdentNull  ||
                   NumDecl (ident$1, {Spec.domNames}) < 2 )  ?
                        "" : " *** Duplicate domain name ***";
      $$.name = ident$1; };

domExp :  EmptyDom  { $$.equal = " ~ == ~ "; };
domExp :  IntDom    { $$.equal = " ~ == ~ "; };
domExp :  NatDom    { $$.equal = " ~ == ~ "; };
domExp :  RealDom   { $$.equal = " ~ == ~ "; };
domExp :  BoolDom   { $$.equal = " ~ == ~ "; };
domExp :  StrDom    { $$.equal = " ~ == ~ "; };

domExp :  EnumDomExp
    { $$.equal = " ~ ::= ~ ";
      identList.listSep = " ~|~ "; };

relIntList :  RelIntListNil
    { $$.intNames = IdentListNil; };

relIntList :  RelIntListPair
    { $$.intNames = (relInt.name :: relIntList$2.intNames); };

relInt :  RelInt
    { local  STR  intError;
      intError = ( ident$1 == IdentNull  ||
                   NumDecl (ident$1, {Spec.intNames}) < 2 )  ?
                   "" : " *** Duplicate relation intention name ***";
      $$.name = ident$1; };

relExtList :  RelExtListNil
    { $$.schNames = IdentListNil;
      $$.relNames = IdentListNil;
      $$.relPKeys = ""; };
```

```
relExtList :  RelExtListPair
    { $$.schNames = (relExt.schName :: relExtList$2.schNames);
      $$.relNames = (relExt.relName :: relExtList$2.relNames);
      $$.relPKeys = (relExt.relPKey  #  relExtList$2.relPKeys); };

relExt :  RelExt
    { local  STR  schError;
      schError = ( ident$1 == IdentNull  ||
                   NumDecl (ident$1, {Spec.REschNames}) < 2 )  ?
                   "" : " *** Duplicate relation extension schema ***";

      local  STR  relError;
      relError = ( ident$2 == IdentNull  ||
                   NumDecl (ident$2, {Spec.relNames}) < 2 )  ?
                      "" : " *** Duplicate relation variable name ***";

      local  STR  intError;
      intError = ( ident$3 == IdentNull  ||
                   IsDecl (ident$3, {Spec.intNames}) )  ?
                     "" : " *** Relation intention not declared ***";

      local STR  constrSep;
      constrSep = optConstr.notNull ? "\\also  \n" : "";

      $$.schName = ident$1;
      $$.relName = ident$2;
      $$.relPKey = RelOfPKey (ident$2, {Spec.relInts}, {Spec.relExts});

      identList.listSep = "    ~ \\land  \\\\"  #  "\n"
                               #  "REQUIRED  ~ "
                               #  unparse (ident$2)
                               #  "    ~ "; };

state :  State
    { local STR  constrSep1;
      constrSep1 = optConstr$1.notNull ? "\\also  \n" : "";

      local STR  constrSep2;
      constrSep2 = optConstr$2.notNull ? "\\where  \n" : "";

      identList$1 = {Spec.REschNames};
      identList$1.listSep = "\n";

      identList$2 = {Spec.relNames};
      identList$2.listSep = "'  = \\{ \\}  ~ \\land  \\\\  \n";

      $$.name = ident$1; };
```

```
forKey :  ForKey
    { local  ident3  PKeyAtt;
      PKeyAtt = PKeyOf (ident$2, {Spec.relExts}); };

xiExp :  XiExp
    { local  ident  DBname;
      DBname = {Spec.DBname};
      identList.listSep = ", "; };

identList :  IdentListPair  { identList$2.listSep = $$.listSep; };
```

B.3 Unparsing rules

```
specification :  Spec  [ ^ : "This is a new specification"  "%n%n%n%n"
                             "Domains"  "%n%n"
                                 "\\begin{zed}"  "%n"
                                 ^
                                 "\\end{zed}"  "%n%n%n%n"
                             "Intention of relations"  "%n%n"
                                 "\\begin{zed}"  "%n"
                                 ^
                                 "\\end{zed}"  "%n%n%n%n"
                             "Extension of relations"  "%n%n"
                                 ^  "%n%n%n%n"
                             "The database state schema"  "%n%n"
                                 ^  "%n%n%n%n"
                             "The basic operations"  "%n%n"
                                 ^  "%n%n%n%n"
                             "The transactions"  "%n%n"
                                 ^  "%n%n%n"
                             "End of the specification" ];

domDefList :  DomDefListNil   [ ^ : ]   /* List of domain definitions */
           |  DomDefListPair  [ @ :  ^  ["\\also"  "%n"]  @ ];

                                                    /* Domain definition */
domDef :  DomDef  [ ^ ::=  @  domError  domExp.equal  ^  "%n" ];

domExp :  EmptyDom     [ @ ::=  "<domain>" ]     /* Domain expression */
       |  IntDom       [ @ ::=  "\\num" ]
       |  NatDom       [ @ ::=  "\\nat" ]
       |  RealDom      [ @ ::=  "REAL" ]
       |  BoolDom      [ @ ::=  "BOOL" ]
       |  StrDom       [ @ ::=  "STRING" ]
       |  EnumDomExp   [ ^ :    @ ];
```

```
relIntList :  RelIntListNil   [ ^ : ]  /* List of relation intentions */
           |  RelIntListPair  [ @ :  ^  ["\\also"  "%n"
                                          "\\also"  "%n"]  @ ];

                                                 /* Relation intention */
relInt :  RelInt  [ ^ ::=  @  intError  " ~ \\defs ~ ["  @  "]"  "%n" ];

relExtList :  RelExtListNil   [ ^ : ]  /* List of relation extensions */
           |  RelExtListPair  [ @ :  ^  ["%n%n%n"]  @ ];

                                                 /* Relation extension */
relExt :  RelExt  [ ^ :  "\\begin{schema}{"  @  "}"  schError  "%n"
                         @  relError  ": \\power "  @  intError  "%n"
                         "\\where  %n"
                         "REQUIRED  ~ "  ident$2
                             "    ~ "  @
                             "    ~ \\land  \\\\"  "%n"
                         "\\also"  "%n"
                         "KEY\\_OF   ~ "  ident$2
                             "%M(21)"  "~ "  @  "%n"
                         constrSep  @
                         "\\end{schema}" ];

                                                  /* The state schemas */
state :  State  [ ^ ::=  "\\begin{schema}{"  @  "}"  "%n"        /* DB */
                             ^  "%n"
                             "\\where"  "%n"
                             ^  constrSep1
                             @
                             "\\end{schema}"  "%n%n%n"

                         "The DeltaDB schema"  "%n%n"      /* DeltaDB */
                             "\\begin{schema}{\\Delta "  ident$1
                                 "}"  "%n"
                             ident$1  "%n"
                             ident$1  "'"  "%n"
                             constrSep2
                             @
                             "\\end{schema}"  "%n%n%n"

                         "The XiDB schema"  "%n%n"            /* XiDB */
                             "\\begin{zed} %n"
                             "\\Xi "  ident$1  "  \\defs  [ \\Delta "
                                 ident$1  "  |  \\theta "  ident$1
                                     " = \\theta "  ident$1  "' ]"  "%n"
                             "\\end{zed}"  "%n%n%n"
```

```
                              "The initialization schema" "%n%n" /* InitDB */
                                  "\\begin{schema}{\\Init\\_"  ident$1
                                      "}"  "%n"
                                  ident$1  "'"  "%n"
                                  "\\where"  "%n"
                                  ^  "'  = \\{ \\}"  "%n"
                                  "\\end{schema}" ];

                                        /* List of foreign key constraints */
forKeyList :  ForKeyListNil   [ ^ : ]
           |  ForKeyListPair  [ @ :  ^  @ ];

                                                        /* Basic operation */
forKey :  ForKey  [ ^ :  "FOR\\_KEY  ~ "  @  "%M(21)"  "~ "  @
                                  "%M(33)"  "~ "  @
                                  "%M(44)"  "~ "  PKeyAtt
                                  "%M(56)"  "~ \\land  \\\\"  "%n" ];

declList :  DeclListNil   [ ^ : ]            /* List of Declarations */
         |  DeclListPair  [ @ :  ^  ["; "  "%t%o%b"]  @ ];

decl :  Decl  [ ^ ::=  @  ": "  ^ ];                        /* Declaration */

                                                 /* Declaration Expression */
declExp :  EmptyDecl     [ @ ::=  "<domain>" ]
        |  IntDecl       [ @ ::=  "\\num" ]
        |  NatDecl       [ @ ::=  "\\nat" ]
        |  RealDecl      [ @ ::=  "REAL" ]
        |  BoolDecl      [ @ ::=  "BOOL" ]
        |  StrDecl       [ @ ::=  "STRING" ]
        |  IdentDeclExp  [ @ ::=  ^ ]
        |  PowerDeclExp  [ ^ ::=  "\\power "  @ ]
        |  SeqDeclExp    [ ^ ::=  "\\seq "    @ ]
        |  CartProdExp   [ ^ ::=  @  " \\cross "  @ ]
        |  SetUnionExp   [ ^ ::=  @  " \\cup "  @ ];

identList :  IdentListNil   [ ^ : ]            /* List of Identifiers */
          |  IdentListPair  [ @ :  ^  [$$.listSep]  @ ];

ident :  IdentNull  [ ^ ::=  "<ident>" ]         /* Normal identifiers */
      |  Ident      [ ^ ::=  ^ ];

                                   /* Identifiers allowing composition */
ident3 :  Ident3      [ ^ ::=  ^ ]
       |  CA2         [ ^ ::=  "(CA2 ~ "  @  " ~ "  @  ")" ];
```

B.4 Concrete input syntax for text editing

```
domDefList_C     { syn  domDefList     t; };
domDef_C         { syn  domDef         t; };
domExp_C         { syn  domExp         t; };
relIntList_C     { syn  relIntList     t; };
relInt_C         { syn  relInt         t; };
declList_C       { syn  declList       t; };
decl_C           { syn  decl           t; };
declExp_C        { syn  declExp        t; };
identList_C      { syn  identList      t; };
ident3_C         { syn  ident3         t; };
ident_C          { syn  ident          t; };

domDefList     ~  domDefList_C.t;
domDef         ~  domDef_C.t;
domExp         ~  domExp_C.t;
relIntList     ~  relIntList_C.t;
relInt         ~  relInt_C.t;
declList       ~  declList_C.t;
decl           ~  decl_C.t;
declExp        ~  declExp_C.t;
identList      ~  identList_C.t;
ident3         ~  ident3_C.t;
ident          ~  ident_C.t;

domDefList_C  ::=  (domDef_C)  { $$.t = (domDef_C.t :: DomDefListNil); }
   |        (domDef_C  domDefList_C)
            { $$.t  =  (domDef_C.t :: domDefList_C$2.t); };

domDef_C  ::=  (ident_C)     { $$.t = DomDef (ident_C.t, EmptyDom); }
   |     (ident_C  domExp_C) { $$.t = DomDef (ident_C.t, domExp_C.t); };

domExp_C  ::=  (NUM)         { $$.t  =  IntDom; }
   |           (NAT)         { $$.t  =  NatDom; }
   |          (REALN)        { $$.t  =  RealDom; }
   |         (BOOLEAN)       { $$.t  =  BoolDom; }
   |          (STRING)       { $$.t  =  StrDom; }
   |       (identList_C)     { $$.t  =  EnumDomExp (identList_C.t); };

relIntList_C  ::=  (relInt_C) { $$.t = (relInt_C.t :: RelIntListNil); };

relInt_C ::= (ident_C)
             { $$.t = RelInt (ident_C.t,
                              DeclListPair (Decl (IdentNull, EmptyDecl),
                                                        DeclListNil) ); }
```

```
   |     (ident_C  declList_C)
         { $$.t  =  RelInt (ident_C.t, declList_C.t); };

declList_C  ::=  (decl_C)      { $$.t = (decl_C.t :: DeclListNil); }
   |     (decl_C  declList_C)  { $$.t = (decl_C.t :: declList_C$2.t); };

decl_C  ::=   (ident_C)       { $$.t = Decl (ident_C.t, EmptyDecl); }
   |     (ident_C  declExp_C) { $$.t = Decl (ident_C.t, declExp_C.t); };

declExp_C  ::=  (NUM)         { $$.t  =  IntDecl; }
   |            (NAT)         { $$.t  =  NatDecl; }
   |           (REALN)        { $$.t  =  RealDecl; }
   |          (BOOLEAN)       { $$.t  =  BoolDecl; }
   |           (STRING)       { $$.t  =  StrDecl; }
   |          (ident_C)       { $$.t  =  IdentDeclExp (ident_C.t); }
   |           (POWER)        { $$.t  =  PowerDeclExp (EmptyDecl); }
   |      (POWER  declExp_C)  { $$.t  =  PowerDeclExp (declExp_C$2.t); }
   |            (SEQ)         { $$.t  =  SeqDeclExp (EmptyDecl); }
   |       (SEQ  declExp_C)   { $$.t  =  SeqDeclExp (declExp_C$2.t); }
   |           (CARTP)        { $$.t  =  CartProdExp (EmptyDecl,
                                                       EmptyDecl); }
   |      (declExp_C  CARTP)  { $$.t  =  CartProdExp (declExp_C$2.t,
                                                       EmptyDecl); }
   |  (declExp_C  CARTP  declExp_C)
                              { $$.t  =  CartProdExp (declExp_C$2.t,
                                                     declExp_C$3.t); }
   |           (UNION)        { $$.t  =  SetUnionExp (EmptyDecl,
                                                       EmptyDecl); }
   |      (declExp_C  UNION)  { $$.t  =  SetUnionExp (declExp_C$2.t,
                                                       EmptyDecl); }
   |  (declExp_C  UNION  declExp_C)
                              { $$.t  =  SetUnionExp (declExp_C$2.t,
                                                    declExp_C$3.t); };

identList_C  ::=  (ident_C)  { $$.t = (ident_C.t :: IdentListNil); }
   |  (ident_C  identList_C) { $$.t = (ident_C.t :: identList_C$2.t); };

ident_C  ::=  (IDENT)       { $$.t  =  Ident (IDENT); };

ident3_C  ::=  (IDENT)      { $$.t  =  Ident3 (Ident (IDENT)); }
   |           (CATT)       { $$.t  =  CA2 (IdentNull, IdentNull); }
   |      (CATT  IDENT)     { $$.t  =  CA2 (Ident (IDENT), IdentNull); }
   |  (CATT  IDENT  IDENT)  { $$.t  =  CA2 (Ident (IDENT$1),
                                            Ident (IDENT$2)); };
```

B.5 Template transformation rules

```
transform  domExp                                         /* Domain Expression */
       on  "Int - .I"       <domExp> :  IntDom,
       on  "Nat - .N"       <domExp> :  NatDom,
       on  "Real - .R"      <domExp> :  RealDom,
       on  "Bool - .B"      <domExp> :  BoolDom,
       on  "String - .S"    <domExp> :  StrDom,
       on  "Enumeration"    <domExp> :  EnumDomExp  ( <identList> );

transform  declExp                                   /* Declaration Expression */
       on  "Int - .I"        <declExp> :  IntDecl,
       on  "Nat - .N"        <declExp> :  NatDecl,
       on  "Real - .R"       <declExp> :  RealDecl,
       on  "Bool - .B"       <declExp> :  BoolDecl,
       on  "String - .S"     <declExp> :  StrDecl,
       on  "PowerSet - .P"   <declExp> :  PowerDeclExp  ( <declExp> ),
       on  "Sequence"        <declExp> :  SeqDeclExp  ( <declExp> ),
       on  "CartProd"        <declExp> :
                CartProdExp  ( <declExp>,  <declExp> ),
       on  "Set Union"       <declExp> :
                SetUnionExp  ( <declExp>,  <declExp> );

transform  ident3                                        /* Composite Attribute */
       on  "CA2"             <ident3> :  CA2 ( <ident>, <ident> );
```

B.6 Lexical syntax declarations

```
WHITESPACE :  < [\ \t\n] >;

NUM      :  < "."[IZ]|"\\num" >;        /* Integer domain */
NAT      :  < ".N"|"\\nat" >;           /* Natural domain */
REALN    :  < ".R"|"REAL" >;            /* Real domain */
BOOLEAN  :  < ".B"|"BOOL" >;            /* Boolean domain */
STRING   :  < ".S"|"STRING" >;          /* String  domain */
POWER    :  < ".P"|"\\power" >;         /* Power   domain */
SEQ      :  < "\\seq" >;                /* Sequence domain */
CARTP    :  < ".CP"|"\\cross" >;        /* Cartesian product */
UNION    :  < ".U"|"\\cup" >;           /* Set Union */

TRUE     :  < "T"|"true" >;             /* Boolean true */
FALSE    :  < "F"|"false" >;            /* Boolean false */

CATT     :  < "CA2" >;                  /* Composite Attribute */

IDENT    :  < [a-zA-Z]("\\_"?[a-zA-Z0-9]+)*['?!]? >;   /* Identifiers */
```

B.7 View definitions

```
view TYPES_D;                        /* View Declaration of TYPES_D */

specification :  Spec
   [ TYPES_D  ^ : "DEFINITION MODULE "  DBname  "_Types;"

                  "%M(32)"  "(***  Database types "
                                      "and constants  ***)"  "%n%n%n%n"

                  "(********  The STRING type  -  not "
                                    "basic In DBPL  *********)"  "%n%n"
                  "TYPE"  "%n%n"
                  "%M(4)"  "STRING    =  ARRAY  [0..100]  OF  CHAR;"
                                                          "%n%n%n%n"

                  "(***************  Domains  -  simple "
                                      "types  ***************)"  "%n%n"
                  "TYPE"  "%n%n"
                  ^  "%n%n%n"

                  "(*********  Relation intentions  "
                                  "-  record types  *********)"  "%n%n"
                  "TYPE"  "%n%n"
                  ^  "%n%n%n"

                  "(********  Auxiliary types  -  "
                                "relation extensions  *******)"  "%n%n"
                  "TYPE"  "%n%n"
                  ^  "%n%n%n"

                  "(*******  Auxiliary types  -  sets of "
                                       "primary keys  *******)"  "%n%n"
                  "TYPE"  "%n%n"
                  relExtList.relPKeys  "%n%n%n"

                  "(*****************  Null value "
                                "constants  *****************)"  "%n%n"
                  "VAR"  "%n%n"
                  "%M(4)"   "NULLNAT   :  CARDINAL;"  "%n"
                  "%M(4)"   "NULLINT   :  CARDINAL;"  "%n"
                  "%M(4)"   "NULLREAL  :  REAL;"      "%n"
                  "%M(4)"   "NULLSTR   :  STRING;"    "%n"

                  ..  ..  ..  "%n"
                  "END "  DBname  "_Types."  "%n" ];
```

```
                                        /* List of domain definitions */
domDefList :  DomDefListNil  [ TYPES_D  ^ : ]
           |  DomDefListPair [ TYPES_D  @ :  ^  @ ];

                                                 /* Domain definition */
domDef :  DomDef  [ TYPES_D  ^ :  "%M(4)"   @
                                  "%M(14)"  "=  "  ^  ";"  "%n" ];

                                                 /* Domain expression */
domExp :  EmptyDom     [ TYPES_D  @ :  "<domain>" ]
       |  IntDom       [ TYPES_D  @ :  "INTEGER" ]
       |  NatDom       [ TYPES_D  @ :  "CARDINAL" ]
       |  RealDom      [ TYPES_D  @ :  "REAL" ]
       |  BoolDom      [ TYPES_D  @ :  "BOOLEAN" ]
       |  StrDom       [ TYPES_D  @ :  "STRING" ]
       |  EnumDomExp   [ TYPES_D  ^ :  "( "  @  " )" ];


                                       /* List of relation intentions */
relIntList :  RelIntListNil   [ TYPES_D  ^ : ]
           |  RelIntListPair  [ TYPES_D  @ :  ^  ["%n"]  @ ];

                                                /* Relation intention */
relInt :  RelInt  [ TYPES_D  ^ :  "%M(4)"  @
                                  "%M(14)"  "=  RECORD"  "%n"
                                  @
                                  "%M(17)"  "END;"  "%n" ];



                                       /* List of relation extensions */
relExtList :  RelExtListNil   [ TYPES_D  ^ : ]
           |  RelExtListPair  [ TYPES_D  @ :  ^  @ ];

                                                /* Relation extension */
relExt :  RelExt  [ TYPES_D  ^ :  ..  ..
                                  "%M(4)"  "REL_"  @  ..
                                  "%M(18)"  "=  RELATION  "  @
                                  "%M(47)"  "OF  "
                                  ..  ident$3  ";"  "%n" ];


declList :  DeclListNil   [ TYPES_D  ^ : ]    /* List of Declarations */
         |  DeclListPair  [ @ :  ^  @ ];

decl :  Decl  [ TYPES_D  ^ :  "%M(21)"  @  ":"         /* Declaration */
                              "%M(32)"  ^   ";"  "%n" ];
```

```
declExp :  EmptyDecl     [ TYPES_D  @ :  "<domain>" ]
        |  IntDecl       [ TYPES_D  @ :  "INTEGER" ]
        |  NatDecl       [ TYPES_D  @ :  "CARDINAL" ]
        |  RealDecl      [ TYPES_D  @ :  "REAL" ]
        |  BoolDecl      [ TYPES_D  @ :  "BOOLEAN" ]
        |  StrDecl       [ TYPES_D  @ :  "STRING" ]
        |  IdentDeclExp  [ TYPES_D  @ :  ^ ]
        |  PowerDeclExp  [ TYPES_D  ^ :  @ ]
        |  SeqDeclExp    [ TYPES_D  ^ :  @ ]
        |  CartProdExp   [ TYPES_D  ^ :  @  .. ]
        |  SetUnionExp   [ TYPES_D  ^ :  @  .. ];

identList :  IdentListNil   [ TYPES_D  ^ : ]   /* List of Identifiers */
          |  IdentListPair  [ TYPES_D  @ :  ^  [", "]  @  ];

ident :  IdentNull  [ TYPES_D  ^ :  "<ident>" ] /* Normal identifiers */
      |  Ident      [ TYPES_D  ^ :  ^ ];

                                  /* Identifiers allowing composition */
ident3 :  Ident3      [ TYPES_D  ^ :  ^ ]
       |  CA2         [ TYPES_D  ^ :  @  ", "  @  ];
```

Bibliography

[1] Barros R. S. M. "Formal Specification of Very Large Software: a Real Example". Master's thesis, Federal University of Pernambuco (UFPE), Departamento de Informática, Recife, Brazil, October 1988. In Portuguese.

[2] Elmasri R. and Navathe S. B. *Fundamentals of Database Systems*, chapter 14, pages 447–477. World Student Series. The Benjamin/Cummings Publishing Company Inc., second edition, 1994.

[3] Spivey J. M. *The Z Notation: A Reference Manual.* Prentice Hall International (UK) Ltd., Hemel Hempstead, UK, second edition, 1992.

[4] Potter B. F., Sinclair J. E., and Till D. *An Introduction to Formal Specification and Z.* Prentice Hall International (UK) Ltd., Hemel Hempstead, UK, 1991.

[5] Date C. J. *An Introduction to Database Systems,* volume 1. World Student Series. Addison-Wesley Publishing Company Inc., Reading, Massachusetts, USA, fifth edition, 1990.

[6] Codd E. F. "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM*, 13(6):377–387, June 1970.

[7] Schmidt J. W. and Matthes F. "The DBPL Project: Advances in Modular Database Programming". *Information Systems*, 19(2):121–140, 1994.

[8] Schmidt J. W. and Matthes F. "The Database Programming Language DBPL: Rationale and Report". FIDE Technical Report FIDE/92/46, Universität Hamburg, Germany, 1992.

[9] Matthes F., Rudloff A., Schmidt J. W., and Subieta K. "The Database Programming Language DBPL: User and System Manual". FIDE Technical Report FIDE/92/47, Universität Hamburg, Germany, 1992.

[10] Reps T. W. and Teitelbaum T. *The Synthesizer Generator: a system for constructing language-based editors.* Texts and Monographs in Computer Science. Springer-Verlag, New York, USA, 1989.

[11] Grammatech, Inc., Ithaca, NY, USA. *The Synthesizer Generator Reference Manual*, fourth edition, 1992.

[12] Ashworth C. and Goodland M. *SSADM: a Practical Approach.* McGraw-Hill, London, 1990.

[13] Dickover M., McGowan C., and Ross D. "Software Design using SADT". In *Structured Analysis and Design*, volume 2, pages 99–114. Infotech, Maidenhead, England, 1978.

[14] Ross D. and Schoman K. "Structured Analysis for Requirements Definition". *IEEE Transactions on Software Engineering*, SE-3(1):6–15, 1977.

[15] Lamb S. "SAMM: A Modelling Tool for Requirements and Design Specification". In *Second IEEE International Computer Software and Applications Conference (Compsac'78)*, pages 48–53, Silver Spring, Maryland, USA, 1978.

[16] Stay J. F. "HIPO and Integrated Program Design". *IBM Systems Journal*, 15(2):143–154, 1976.

[17] Gane C. and Sarson T. *Structured Systems Analyisis: Tools and Techniques*. Prentice Hall Inc., Englewood Cliffs, NJ, USA, 1979.

[18] Chen P. P. S. "The Entity-Relationship Model - Toward a Unified View of Data". *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[19] Elmasri R. and Navathe S. B. *Fundamentals of Database Systems*, chapter 21, pages 611–661. World Student Series. The Benjamin/Cummings Publishing Company Inc., second edition, 1994.

[20] Codd E. F. "Extending the Database Relational Model to Capture More Meaning". *ACM Transactions on Database Systems*, 4(4):397–434, December 1979.

[21] American National Standards Institute. "The Database Language SQL". Document ANSI X3.135, 1986.

[22] Batini C., Ceri S., and Navathe S. B. *Conceptual Database Design: an Entity-Relationship Approach*. Benjamin/Cummings Series on Database Systems and Applications. The Benjamin/Cummings Publishing Company Inc., Redwood City, California, USA, 1992.

[23] Bagchi T. and Chaudhri V. *Interactive Relational Database Design*, volume 402 of *Lecture Notes in Computing Science*. Springer-Verlag, 1989.

[24] Ceri S. and Gottlob G. "Normalization of Relations and PROLOG". *Communications of the ACM*, 29(6):524–544, 1986.

[25] Wing J. M. "A Specifier's Introduction to Formal Methods". *IEEE Computer*, 23(9):8–24, September 1990.

[26] Ince D. C. *An Introduction to Discrete Mathematics and Formal System Specification*. Oxford Applied Mathematics and Computing Science Series. Oxford University Press, UK, 1988.

[27] Hall A. "Seven Myths of Formal Methods". *IEEE Software*, 7(5):11–19, September 1990.

[28] Duce D. A. and Fielding E. V. C. "Formal Specification - A Comparison of Two Techniques". *The Computer Journal*, 30(4):316–327, 1987.

[29] Jones C. B. *Systematic Software Development Using VDM.* Prentice Hall International (UK) Ltd., Hemel Hempstead, UK, second edition, 1990.

[30] Jones C. B. and Shaw R. C. F. (eds.). *Case Studies in Systematic Software Development.* Prentice Hall International (UK) Ltd., Hemel Hempstead, UK, 1990.

[31] The RAISE Language Group. *The RAISE Specification Language.* The BCS Practitioners Series. Prentice Hall International (UK) Ltd., 1992.

[32] Gordon M. J. C. and Melham T. J. (eds.). *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, UK, 1993.

[33] Hoare C. A. R. *Communicating Sequential Processes.* Prentice Hall International (UK) Ltd., Hemel Hempstead, UK, 1985.

[34] Milner R. *Communication and Concurrency.* Prentice Hall International (UK) Ltd., Hemel Hempstead, UK, 1989.

[35] Goguen J. A. and Winkler T. "Introducing OBJ3". Technical Report SRI-CSL-88-9, SRL, Menlo Park, USA, 1988.

[36] Guttag J. V. and Horning J. J. (eds.). *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[37] Burstall R. M. and Goguen J. A. *The Semantics of Clear, a Specification Language*, volume 86 of *Lecture Notes in Computing Science.* Springer-Verlag, 1981.

[38] Luckham D. C. *Programming with Specifications: An Introduction to ANNA - A Specification Language for ADA.* Springer-Verlag, 1990.

[39] Ehrig H. and Mahr B. *Fundamentals of Algebraic Specification 1 - Equations and Initial Semantics.* Number 6 in EATCS Monographs on Theoretical Computing Science. Springer-Verlag, 1985.

[40] Samson W. B. and Wakelin A. "Algebraic Specification of Databases: A Survey from a Database Perspective". In Harper D. J. and Norrie M. C. (eds.), *Specification of Database Systems, Glasgow 1991*, Workshops in Computing Series, pages 246–254. Springer-Verlag, 1992.

[41] Sampaio A. C. and Meira S. L. "Modular Extensions to Z". In Bjørner D., Hoare C. A. R., and Langmaack H. (eds.), *VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computing Science*, pages 211–232. Springer-Verlag, 1990.

[42] Fitzgerald J. *Modularity in Model-Oriented Formal Specifications and its Interaction with Formal Reasoning.* PhD thesis, University of Manchester, Department of Computing Science, 1991.

[43] Hoare C. A. R. "Preface". In Bjørner D., Hoare C. A. R., and Langmaack H. (eds.), *VDM'90 - VDM and Z!*, volume 428 of *Lecture Notes in Computing Science*, pages vii–x. Springer-Verlag, 1990.

[44] Hayes I. J., Jones C. B., and Nicholls J. E. “Understanding the Differences between VDM and Z”. Technical Report UMCS-93-8-1, Department of Computing Science, University of Manchester, UK, August 1993.

[45] Plat N. *Experiments with Formal Methods in Software Engineering.* PhD thesis, Delft University of Technology, Faculty of Technical Mathematics and Informatics, The Netherlands, 1993.

[46] McParland P. J. *Software tools to Support Formal Methods.* PhD thesis, The Queen’s University of Belfast, Northern Ireland, October 1989.

[47] Bloomfield R. E. and Froome P. K. D. “The Application of Formal Methods to the Assessment of High Integrity Software”. *IEEE Transactions on Software Engineering*, 20(9):988–993, September 1986.

[48] Diller A. *Z: An Introduction to Formal Methods.* John Wiley & Sons Ltd., Chichester, UK, second edition, 1994.

[49] Wordsworth J. B. *Software Development with Z.* Addison-Wesley Publishing Company Inc., Wokingham, England, 1992.

[50] Spivey J. M. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, UK, 1988.

[51] Hayes I. J. (ed.). *Specification Case Studies.* Prentice Hall International (UK) Ltd., Hemel Hempstead, UK, second edition, 1993.

[52] Barden R., Stepney S., and Cooper D. *Z in Practice.* The BCS Practitioners Series. Prentice Hall International (UK) Ltd., Hemel Hempstead, UK, 1994.

[53] Barros R. S. M. and Meira S. L. “A Method for the Specification of Relational Databases using Zc”. In *proceedings of XVI SEMISH*, SBC Brazilian Congress, Uberlândia, Brazil, July 1989. In Portuguese.

[54] Furtado A. L. and Casanova M. A. “Updating Relational Views”. In Kim W., Reiner D., and Batory D. (eds.), *Query Processing in Database systems*, pages 127–142. Springer-Verlag, 1985.

[55] Buff H. W. “Why Codd’s Rule No. 6 Must be Reformulated”. *ACM SIGMOD Record*, 15(4), December 1988.

[56] Wirth N. *Programming in Modula-2.* Texts and Monographs in Computer Science. Springer-Verlag, Berlin, Germany, third edition, 1985.

[57] Welsh J. and Elder J. *Introduction to Modula-2.* Prentice Hall International (UK) Ltd., Hemel Hempstead, UK, 1987.

[58] King S. “The Use of Z in the Restructure of IBM CICS”. In Hayes I. J. (ed.), *Specification Case Studies*, chapter 14, pages 202–213. Prentice Hall International (UK) Ltd., 1993.

[59] Houston I. and King S. "CICS Project Report: Experiences and Results from the Use of Z in IBM". In Prehn S. and Toetenel W. J. (eds.), *VDM'91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computing Science*, pages 588–596. Springer-Verlag, October 1991.

[60] Collins B. P., Nicholls J. E., and Sørensen I. H. "Introducing Formal Methods: The CICS Experience with Z". Technical Report TR12.277, IBM (UK) Laboratories, Hursley, UK, December 1987.

[61] King S. "Specifying the IBM CICS Application Programming Interface". In Hayes I. J. (ed.), *Specification Case Studies*, chapter 15, pages 214–225. Prentice Hall International (UK) Ltd., 1993.

[62] Wordsworth J. B. "The CICS Aplication Programming Interface Definition". In Nicholls J. E. (ed.), *Z User Workshop, Oxford 1990*, Workshops in Computing Series, pages 285–294. Springer-Verlag, 1991.

[63] Hayes I. J. "Applying Formal Specification to the Development of Software in Industry". In Hayes I. J. (ed.), *Specification Case Studies*, chapter 13, pages 181–201. Prentice Hall International (UK) Ltd., 1993.

[64] Hayes I. J. "CICS Temporary Storage". In Hayes I. J. (ed.), *Specification Case Studies*, chapter 16, pages 226–237. Prentice Hall International (UK) Ltd., 1993.

[65] Hayes I. J. "CICS Message System". In Hayes I. J. (ed.), *Specification Case Studies*, chapter 17, pages 238–243. Prentice Hall International (UK) Ltd., 1993.

[66] Jacky J. "Specifying a Safety-Critical Control System in Z". In Woodcock J. C. P. and Larsen P. G. (eds.), *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computing Science*, pages 388–402. Springer-Verlag, April 1993.

[67] Boswell T. "Specification and Validation of a Security Policy Model". In Woodcock J. C. P. and Larsen P. G. (eds.), *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computing Science*, pages 42–51. Springer-Verlag, April 1993. Industrial Usage Report.

[68] King T. "Formalizing British Rail's Signalling Rules". In Naftalin M., Denvir T., and Bertran M. (eds.), *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computing Science*, pages 45–54. Springer-Verlag, October 1994. Industrial Usage Report.

[69] Wood A. W. "The SWORD Model of Multilevel Secure Databases". RSRE Report 90008, Defence Research Agency (DRA), Electronics Division, UK, June 1990.

[70] Smith P. and Keighley R. "The Formal Development of a Secure Transaction Mechanism". In Prehn S. and Toetenel W. J. (eds.), *VDM'91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computing Science*, pages 457–476. Springer-Verlag, October 1991.

[71] Craigen D., Gerhart S., and Ralston T. "Formal Methods Reality Check: Industrial Usage". In Woodcock J. C. P. and Larsen P. G. (eds.), *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computing Science*, pages 250–267. Springer-Verlag, April 1993.

[72] Craigen D., Gerhart S., and Ralston T. "An International Survey of Industrial Applications of Formal Methods". Technical Report NIST GCR 93/626-V1 & 2, Atomic Energy Control Board of Canada, US National Institute of Standards and Technology, and US Naval Research Laboratories, 1993.

[73] Austin S. and Parkin G. I. "Formal Methods: A Survey". Technical report, Division of Information Technology and Computing, National Physical Laboratory, Teddington, Middlesex, UK, March 1993.

[74] Bowen J. P. and Hinchey M. G. "Seven More Myths of Formal Methods: Dispelling Industrial Prejudices". In Naftalin M., Denvir T., and Bertran M. (eds.), *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computing Science*, pages 105–117. Springer-Verlag, October 1994.

[75] Weber-Wulff D. "Selling Formal Methods to Industry". In Woodcock J. C. P. and Larsen P. G. (eds.), *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computing Science*, pages 671–678. Springer-Verlag, April 1993.

[76] Morris J. M. "A Theoretical Basis for Stepwise Refinement and The Programming Calculus". *Science of Computer Programming*, 9(3):287–306, December 1987.

[77] Morgan C. C. "The Specification Statement". *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[78] Morgan C. *Programming from Specifications.* Prentice Hall International (UK) Ltd., Hemel Hempstead, UK, second edition, 1994.

[79] Back R.-J. R. "A Calculus of Refinements for Program Derivations". *Acta Informatica*, 25(6):593–624, August 1988.

[80] Dijkstra E. W. *A Discipline of Programming.* Prentice Hall Inc., Englewood Cliffs, NJ, USA, 1988.

[81] O'Neill G. "Automatic Translation of VDM Specifications into Standard ML Programs". *The Computer Journal*, 35(6):623–624, 1992. Short note.

[82] O'Neill G. "Automatic Translation of VDM Specifications into Standard ML Programs". NPL Report DITC 196/92, National Physical Laboratory, Teddington, Middlesex, UK, February 1992.

[83] Milner R., Tofte M., and Harper R. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts, USA, 1990.

[84] Date C. J. *An Introduction to Database Systems,* volume 1, chapter 13, pages 295–333. World Student Series. Addison-Wesley Publishing Company Inc., Reading, Massachusetts, USA, fifth edition, 1990.

[85] Wong E. Y. and Samson W. B. "The Specification of a Relational Database (PRECI) and its Realization in Hope". *The Computer Journal*, 29(3):261–268, 1986.

[86] Burstall R. M., MacQueen D. B., and Sannela D. T. "Hope: an Experimental Applicative Language". Internal Report CSR-62-80, Department of Computing Science, University of Edinburgh, UK, 1980.

[87] Fitzgerald J. and Jones C. B. "Modularizing the Formal Description of a Database System". In Bjørner D., Hoare C. A. R., and Langmaack H. (eds.), *VDM'90 - VDM and Z!*, volume 428 of *Lecture Notes in Computing Science*, pages 189–210. Springer-Verlag, 1990.

[88] Winterbottom N. and Sharman G. C. H. "NDB: Non-Programmer Database Facility". Technical Report TR.12.179, IBM (UK) Laboratories, Hursley, UK, September 1979.

[89] Turner R. and Lowden B. G. T. "An Introduction to the Formal Specification of Relational Query Languages". *The Computer Journal*, 28(2):162–169, 1985.

[90] Relational Technology Inc., Alameda, California, USA. *INGRES/QUEL Reference Manual*, 1988.

[91] Date C. J. *An Introduction to Database Systems,* volume 1, chapter 14, pages 335–368. World Student Series. Addison-Wesley Publishing Company Inc., Reading, Massachusetts, USA, fifth edition, 1990.

[92] Sufrin B. and Hughes J. "A Tutorial Introduction to Relational Algebra". Programming Research Group, Oxford University Computing Laboratory, July 1985. Unpublished draft.

[93] Khosla S., Maibaum T., and Sadler M. "Database Specification". In Steel Jr. T. B. and Meersman R. (eds.), *Database Semantics (DS-1)*, pages 141–158. Elsevier Science Publishers B. V. (North Holland), 1986.

[94] Fiadeiro J. and Sernadas A. "Specification and Verification of Database Dynamics". *Acta Informatica*, 25(6):625–661, 1988.

[95] Abiteboul S. and Vianu V. "A Transaction-based Approach to Relational Database Specification". *Journal of the ACM*, 36(4):758–789, October 1989.

[96] Harper D. J. and Norrie M. C. (eds.). *Specification of Database Systems, Glasgow 1991*, Workshops in Computing Series. Springer-Verlag, 1992.

[97] Pastor J. A. and Olivé A. "An Approach to the Synthesis of Update Transactions in Deductive Databases". In *Proceedings of CISMOD'94 (Fifth International Conference on Information Systems and Management of Data)*, Madras, India, October 1994.

[98] Minker J. *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 1988.

[99] Gallaire H., Minker J., and Nicolas J.-M. "Logic and Databases: A Deductive Approach". *ACM Computing Surveys*, 16(2):153–185, June 1984.

[100] Pastor J. A. "Deriving Consistency-Preserving Transaction Specifications for View Updates in Relational Databases". In *Proceedings of III International Workshop on the Deductive Approach to Information Systems and Databases*, pages 275–300, Roses, Catalonia, Spain, 1992.

[101] Sheard T. and Stemple D. "Automatic Verification of Database Transaction Safety". *ACM Transactions on Database Systems*, 14(3):322–368, September 1989.

[102] Boyer R. S. and Moore J. S. *A Computational Logic*. Academic Press, New York, USA, 1979.

[103] Steinberg G., Faley R., and Chinn S. "Automatic Database Generation by Novice End-Users Using English Sentences". *Journal of Systems Management*, 45(3):10–15, March 1994.

[104] Qian X. "The Deductive Synthesis of Database Transactions". *ACM Transactions on Database Systems*, 18(4):626–677, December 1993.

[105] Qian X. *The Deductive Synthesis of Database Transactions*. PhD dissertation, Stanford University, Computer Science Department, USA, November 1989.

[106] Manna Z. and Waldinger R. *The Logical Basis for Computer Programming,* volume 2. Addison-Wesley Publishing Company Inc., Reading, Massachusetts, USA, 1990.

[107] Schewe K.-D., Schmidt J., and Wetzel I. "Specification and Refinement in an Integrated Database Application Environment". In Prehn S. and Toetenel W. J. (eds.), *VDM'91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computing Science*, pages 496–510. Springer-Verlag, October 1991.

[108] Borgida A., Mylopoulos J., and Schmidt J. W. "Final Version on TDL Design". DAIDA Deliverable DES1.2, ESPRIT Project 892, September 1987.

[109] Mylopoulos J., Bernstein P. A., and Wong H. K. T. "A Language Facility for Designing Interactive Database-Intensive Applications". *ACM Transactions on Database Systems*, 5(2):185–207, June 1980.

[110] Abrial J.-R. "Abstract Machines, Parts I, II, and III". 26 Rue des Plantes, Paris 75014, France, 1990. Unpublished.

[111] Abrial J.-R. et al. "The B Method". In Prehn S. and Toetenel W. J. (eds.), *VDM'91: Formal Software Development Methods*, volume 552 of *Lecture Notes in Computing Science*, pages 398–405. Springer-Verlag, October 1991. Tutorial.

[112] Edinburgh Portable Compilers Ltd., UK. *B-Tool Reference Manual*, 1991.

[113] Günther T., Schewe K.-D., and Wetzel I. "On the Derivation of Executable Database Programs from Formal Specifications". In Woodcock J. C. P. and Larsen P. G. (eds.), *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computing Science*, pages 351–366. Springer-Verlag, April 1993.

[114] Schewe K.-D., Wetzel I., and Schmidt J. "Towards a Structured Specification Language for Database Applications". In Harper D. J. and Norrie M. C. (eds.), *Specification of Database Systems, Glasgow 1991*, Workshops in Computing Series, pages 255–274. Springer-Verlag, 1992.

[115] Barros R. S. M. "Formal Specification of Relational Database Applications: A Method and an Example". Research Report GE-93-02, Department of Computing Science, The University of Glasgow, UK, September 1993.

[116] van Diepen M. J. and van Hee K. M. "A Formal Semantics for Z and the Link between Z and the Relational Algebra". In Bjørner D., Hoare C. A. R., and Langmaack H. (eds.), *VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computing Science*, pages 526–551. Springer-Verlag, 1990.

[117] Barros R. S. M. "Company Database Example". Department of Computing Science, The University of Glasgow, 1993. First step specification.

[118] Barros R. S. M. "Company Database Example". Department of Computing Science, The University of Glasgow, 1993. Second step specification.

[119] Barros R. S. M. "Company Database Example". Department of Computing Science, The University of Glasgow, 1993. Third step specification.

[120] Date C. J. "Null Values in Database Management". In *Relational Database: Selected Writings*, pages 313–334. Addison-Wesley Publishing Company Inc., Reading, Massachusetts, USA, 1986.

[121] Codd E. F. *The Relational Model for Database Management - Version 2*. Addison-Wesley Publishing Company Inc., Reading, Massachusetts, USA, 1990.

[122] Barros R. S. M. "Deriving Relational Database Programs from Formal Specifications". In Naftalin M., Denvir T., and Bertran M. (eds.), *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computing Science*, pages 703–723. Springer-Verlag, October 1994.

[123] Senn J. A. *The Student Edition of dBASE IV, version 1.1, User's Manual*. Addison-Wesley Publishing Company Inc., Reading, Massachusetts, USA, 1991.

[124] Date C. J. and White C. J. *A Guide to DB2*. Addison-Wesley Publishing Company Inc., Reading, Massachusetts, USA, third edition, 1989.

[125] Fiddian N. J., Gray W. A., and Howells D. I. "Query Language Inter-translation in Heterogeneous Distributed Database Systems". *Database Technology*, 2:3–8, 1990.

[126] Spivey J. M. "A Guide to the `zed` Style Option". Oxford University Computing Laboratory, Oxford, UK, December 1990. Unpublished.

[127] Barros R. S. M. and Harper D. J. "Formal Development of Relational Database Applications". In Harper D. J. and Norrie M. C. (eds.), *Specification of Database Systems, Glasgow 1991*, Workshops in Computing Series, pages 21–43. Springer-Verlag, 1992.

[128] Sampaio A. C. and Meira S. L. "Zc: A Notation for Complex Systems Specification". In *proceedings of XV SEMISH*, SBC Brazilian Congress, Rio de Janeiro, Brazil, 1988. In Portuguese.

[129] Sampaio A. C. "Zc: A Notation for Complex Systems Specification". Master's thesis, Federal University of Pernambuco (UFPE), Departamento de Informática, Recife, Brazil, November 1988. In Portuguese.

[130] Vasconcelos A. M. "Specifying the Interface of a Hypertext System". Master's thesis, Federal University of Pernambuco (UFPE), Departamento de Informática, Recife, Brazil, August 1989. In Portuguese.

[131] Zdonik S. B. and Maier D. (eds.). *Readings in Object-Oriented Database Systems.* Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 1990.

Index