A Formal Specification in Z of the Relational Data Model, Version 2, of E. F. Codd

Dorel D. Băluţă

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfilment of the Requirements for
the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

March 1995

ISBN 0-612-01361-8

ABSTRACT

A Formal Specification in Z of the Relational Data Model, Version 2, by E. F. Codd

Dorel Băluţă

Formal specifications are becoming more widely used in developing computer systems. They are gaining acceptance as an important component of methods for developing high-quality software. Data modelling at the conceptual level has always been a major part of database design of information systems in general. A data model is a set of conceptual tools for describing relevant properties of the system under consideration, and it consists of three distinct, yet closely interrelated parts: the data structure description, a set of operations for data update and retrieval and a set of constraints that data values must satisfy in order to be considered valid. The relational model, first introduced by Codd in 1970, has undergone a certain amount of revision and refinement since its original definition. The last version of it, RM/V2, was defined by Codd in 1990 by preserving all existing features of the previous versions, and adding some new features intended to improve the understanding of this data model and to enhance its power. This thesis presents a formal definition of the basic concepts of the relational data model, including some additional features introduced by RM/V2. The formal notation that we use is Z, a model-oriented specification language which is widely used in industry as part of the software development process.

Acknowledgments

I would like to express my fully gratitude to my thesis supervisor Dr. Greg Butler for his continued support and guidance during the period of research that has lead to the present thesis. The many stimulating and varied discussions which we have had were very helpful for my work.

I am grateful to Concordia University for its financial support during my graduate studies and research.

In the memory of my mother

Contents

Chapter 1

Introduction

Formal methods are becoming more widely used in developing computer systems. They provide frameworks within which people can specify, develop and verify systems in a systematic, rather than ad hoc, manner. The use of mathematically based techniques for the description of system structure and behaviour is necessitated by the increasing size and complexity of modern software systems, which can not be handled by classical structured programming and design techniques.

The ability to specify system properties in a precise and rigorous way has been invaluable in the design and implementation of many computer systems, regardless of their size [Hal90, Haye92]. Formal specifications are gaining acceptance as an important component of methods for developing high-quality software [But94]. A formal specification has many roles in the development process. Some examples are:

- as a mean of precise communication between the customer and the software supplier,
- as the standard document against which the final software product is verified,
- as a model from which it is possible to prove properties of the system and as an aid to validating the formal specification against the informal requirements.

Growing awareness of the need for formal specification has led to the development of various specification languages. Some of these take as their starting point a large body of standard mathematical concepts and notation, including set theory and predicate logic, and use these to build models of systems. Among these, the most popular are Z [Sp88] and VDM [Jon86], which has even been recommended to become an official standard for software system specification.

Data modelling at the conceptual level has always been a major part of database design of information systems in general. A data model is a set of conceptual tools for describing relevant properties of the system under consideration, and it consists of three distinct, yet closely interrelated parts: the data structure description, a set of operations for data update and retrieval and a set of constraints that data values must satisfy in order to be considered valid. One of the most popular data models which enjoys widespread acceptance is the relational data model.

The relational model, first introduced by Codd in 1970 [Cod70], has undergone a certain amount of revision and refinement since its original definition [Dat86]. The last version of it, RM/V2, was defined by Codd in 1990 [Cod90] by preserving all existing features of the previous versions (RM/V1 [Cod70], RM/T [Cod79]), and adding some new features intended to improve the understanding of this data model and to enhance its power. The most important additional features in RM/V2 are as follows:

- a new treatment of items of data missing because they represent properties that happen to be inapplicable to certain object instances for example, the name of the spouse of an employee when that employee happens to be unmarried;

- new features supporting all kinds of integrity constraints, especially the user-defined type;
- a more detailed account of view updatability;
- a more detailed account of what should be in the catalog;
- some of the fundamental laws on which the relational model is based.

In RM/V2, the author attempts to emphasize the numerous semantic features in the relational model. The semantic features include the following:

- domains, primary keys, and foreign keys;
- duplicate values are permitted within columns of a relation, *but duplicate rows are prohibited*;
- systematic handling of missing information independent of the type of datum that is missing.

The motivations for Version 2 of the relational data model included the following:

1. all of the motivations for Version 1;

2. the errors in implementing RM/V1, such as:

   a. duplicate rows permitted by the language SQL;

   b. primary keys have either been omitted altogether, or they have been made optional on base relations;

   c. major omissions, especially of all features supporting the meaning of the data (including domains);

   d. omission of almost all the features concerned with preserving the integrity of the database.

3. the need to assemble all features of the relational model in one document for DBMS vendors, users, and inventors of new data models who seem to be unaware of the scope of the relational data model;

4. the need for extensions such as new kinds of joins, user-defined integrity and view updatability;

5. the need for users to realize what they are missing in present relational DBMS products because only partial support of the relational model is built into these

products [McGo94].

It is true, and indeed significant, that the changes introduced by RM/V2 have been evolutionary, not revolutionary, in nature. Nevertheless, the situation is that (to our knowledge) there does not exist any formal definition of the RM/V2. The previous work on specifying the relational data model includes only some semi-formal definitions of this data model which do not specify all the concepts involved. It therefore seems worthwhile to attempt such a definition, to provide a convenient source of reference and understanding on the relational data model and on the recent work on extending the model to incorporate additional meaning. This thesis presents a formal definition of the basic concepts of the relational data model, including some of the new features introduced by RM/V2. The formal notation that we use is Z, a model-oriented specification language which is widely used in industry as part of the software development process.

Chapter 2 of this thesis has three parts. First we present the benefits of the formal specification during the software development process. The second part is a short description of existing representative specification languages. The last part presents briefly the Z notation, the mathematical entities used by it, and how systems can be modelled using Z schemas.

Chapter 3 presents a survey of previous work on the specification of databases, related specification case studies using Z or VDM, and formalizations of different data models.

The main part of the thesis is presented in *Chapter 4* where a formal specification of the RM/V2 is defined. The Z specification language is used throughout, mainly because of its elegance and simplicity, its mathematical foundation and because of the link that exists between Z and relational algebra. The specification includes concepts like domains, rows and relations, retrieval and manipulative operators, and integrity rules such as entity integrity and referential integrity. The definitions consider the case of composite keys and make use of the naming technique defined in RM/V2. Some additional features defined in RM/V2 such as the new treatment of missing data are also specified. The specification does not include concepts like view and view updatability, and the definition of the advanced operators.

Chapter 5 presents how different properties of the relational data model can be proved using the Z specification.

Chapter 6 presents the conclusions of this project, several remarks about Z and FUZZ, and proposed further work.

Chapter 2

Formal Specifications and Formal Specification Languages

This chapter is intended to present the importance of formal specifications in the software development process, to briefly describe the common specification languages and to introduce the Z language, its properties and its mathematical toolkit.

2.1 The need for formality and abstraction

System specification plays a central role in the software life-cycle for several reasons: It serves as a contract, a valuable piece of documentation, and a means of communication among a client, a specifier, and an implementor; it represents in a systematic fashion the current state of the real world, its problems and its future requirements;

it enables the system developer to transform real world problems into other forms which are more manageable in terms of size, complexity, human understanding and computer processability. A formal software specification is a specification expressed in a language whose vocabulary, syntax and semantics are formally defined. It describes in a precise way the properties which a software system must have, without constraining the way in which these properties are implemented, in other words it describes *what* the system must do without saying *how* it is to be done. The need for formality and abstraction means that the specification languages cannot be based on natural language, due to its lack of precision, but must be based on mathematics. The advantages of using such a formal language for precise specifications [Som89] are as follows:

1. Formal specification provides insights into and better understanding of the software requirements and software design;

2. Having a formal system specification written using a formal specification language, it may be possible to prove that a program conforms to its specification;

3. Formal specifications may be automatically processed using software tools built to help with their development and debugging;

4. A formal specification may serve as the basis for the development of a prototype system;

5. The mathematical basis of the formal specifications provides the means of precisely defining notions like consistency, completeness and correctness. Formal specifications can be studied and analysed using mathematical methods;

6. Formal specifications may be used in identifying appropriate test cases in formal program verification.

These advantages of formal specifications fully justify their use in the software development process. Their use should be seen as a way of achieving a high degree of confidence that a system will conform to its specification.

The role of the *language* in connection with formal specifications is very important. We can say that, in many respects, the practical progress in software engineering is language-driven: it is hard to introduce methodological concepts unless these are concretely available as constructs in the language in use. This has been a major motivation behind the introduction of formal specification languages. In practice it is not enough to have a good methodological frame or writing formal specifications: we need also a language as a vehicle. In order for a requirement specification to be useful in systems development, the specification language must exhibit various features which contribute to the ease and user-friendliness of the specification process [Tse91].

2.2 Common Families of Formal Specification Languages

The most commonly encountered families of formal specification languages are model-oriented and property-oriented techniques [Wi90]. Using a model-oriented method, a specifier defines a system's behavior directly by constructing a model of the system in terms of mathematical structures. Using a property-oriented method, a specifier defines the system's behaviour indirectly by stating a set of properties, usually in the form of a set of axioms that the system and its behaviour must satisfy.

2.2.1 Model-oriented Techniques

Model-oriented specification languages encourage the construction of a model of a system in terms of sets, maps, sequences and predicates, such that the exhibited behavioral properties are those desired for the system. The behavior of the system is studied through a sequence of *states* of the modeled object where each state is an instance of the variables, data types and operations causing a transition of the state. This approach is also called *operational*. These are the techniques most widespread in industrial use, and the most readily grasped. Model-oriented methods for specifying the behaviour of sequential programs and abstract data types include Parnas' state-machines [Tsa95]; VDM [Jon86]; and Z [Sp88, Sp92].

Methods for specifying the behaviour of concurrent and distributed systems include Petri nets; Milner's Calculus of Communicating Systems (CSS); and Hoare's Communicating Sequential Systems (CSP) [Hoa85]. The process algebras exemplified by CSP allow a system to be modelled by a collection of processes which communicate with one another. The mathematical basis of the CSP has much in common with that of Z and this facilitates the complementary use of the two techniques.

2.2.2 Property-oriented Techniques

Property-oriented methods can be broken into two categories, sometimes referred to as axiomatic and algebraic.

Axiomatic methods stem from Hoare's work on proofs of correctness of implementations of abstract data types, where first-order predicate logic preconditions and postconditions are used for the specification of each data type. OBJ and Larch [Wi87] are example specification languages that support an axiomatic method.

The *algebraic* specification technique has been developed independently by Goguen *et al* [Gog75] and Guttag and Horning [Gu77], among others. It has been used first in the specification of abstract data types but, since then, this technique has been extended into a general-purpose approach to system specification. The name *algebraic* is derived from the fact that the technique's theoretical underpinning is the mathematics of *many sorted* algebras (an algebra is a set on whose members certain operations are defined and which possess certain formal properties; the term "many sorted" refers to the fact that the sets are heterogeneous, i.e. comprise members of various types or *sorts*). In the algebraic approach, a specification has four parts: an introduction part where the sort of the entity being specified is introduced and the names of any other specification which are required are set out; an informal description of the sort and its operations; a signature part containing the definition of the operations on that object and the sorts of their parameters; and an axioms part where the relationships between the sort operations are specified. The axioms are stated in an *equational* style, i.e. as a set of equations between expressions involving the operations. The introduction part of a specification also includes an *import* part which indicates the other required specifications.

2.3 The Z Notation

The Z notation is a model-oriented formal specification language based on set theory and first order logic. It has been developed at the Programming Research Group at the Oxford University Computing Laboratory and elsewhere for well over a decade. Z is now used by industry as part of the software (and hardware) development process: IBM has used Z successfully in specifying parts of its CICS transaction processing system [Haye92]; Z has been used to clarify an IEEE floating point standard; It is currently undergoing BSI standardization in the UK and has been accepted for the ISO standardization process internationally. The aim of this section is to introduce the basic features of Z which will be used throughout the specification of the relational data model. For a complete reference of the syntax and semantic of the Z notation see [Sp88, Sp92], [Pot91] or [Wo92].

A specification written in Z is a mixture of formal, mathematical statements and informal explanatory text. Both have their importance: the formal part gives a precise definition of the system being specified while the informal text makes the specification more comprehensible and readable, linking the abstract definition of the system to the real world.

The set theory on which Z is based on is a typed set theory, that is the objects in its universe may belong to different types, and there is no overlap between distinct types. The notation of predicate logic is used to describe abstractly the effect of each operation of the system in a way that makes possible reasoning about system's behaviour.

A very important feature of Z is the schema, which allows us to decompose a specification into small pieces which can be referred to throughout the specification. Schemas are used to describe both static and dynamic properties of a system. The static aspects refer to the set of states that the system can occupy and the invariant relationships that must be satisfied by each state of the system. The dynamic properties include the set of operations on the system, the relationship between their inputs and outputs, and their effect on the system state. Using Z, different parts of a system can be specified separately, then they can be related and combined using the schema composition language so that large specifications can be built up in stages. Schemas can be used to specify a transformation from one representation of a system to another one containing more details of a concrete design. Using this process called refinement, a specification of a system can start at a very general level and other specifications can be derived from it by introducing more detail.

In Z, a schema has a *name*, a *signature* part and a *predicate* part:

```
┌─ SchemaName ─────────────────────────
│ Signature_part
├──────────────
│ Predicate_part
└──────────────────────────────────────
```

The signature part is a collection of typed variables and the predicate part contains one or more predicates (or axioms) over these variables. Schemas are building blocks which can be combined and used in other schemas. The effects of including a schema A in schema B is that schema B inherits all the variables and predicates defined in schema A. Using this feature, the state space of a system can be split between several schemas which are combined to define the overall state space. This is an important advantage of Z over VDM, which does not have this property of making up the state space of the system combining different modules.

2.3.1 Sets and Predicates

The set theory on which Z is based on is a typed set theory, that is the objects in its universe may belong to different types, and there is no overlap between distinct types. One way to introduce a type into a Z specification is as a *given set*. In the RM/V2 specification, the given set $RNAME$ which denotes all the relation names that can be defined for a relational database will be introduced by putting its name in square brackets thus:

$[RNAME]$

This notation is a declaration introducing the name *RNAME*. A name denotes the thing that it names, in this case a certain set that is to be a type. In Z, every value we speak of must be assigned a type. A name is assigned a type when it is declared. Thus if *RNAME* is a type we can declare a relation name *rel*1 with the following declaration:

$| \ rel1 : RNAME$

The name *rel*1 is called a *variable* and it denotes some undetermined value of type *RNAME*.

A set may be defined by *set enumeration*, that is by listing its members in some order, enclosed in braces. The order of the items is arbitrary, but they must be all of the same type. Two sets of values of the same type are equal if and only if they have the same members. A set with no members is called a *null set* or an *empty set*.

In Z we can introduce a type with a small number of members, and give names to the members of the type. Here is an instance of a declaration to define a boolean type called $BOOL$ which will be used in our specification:

$$BOOL ::= True \mid False$$

The above notation is called a *data type* definition. The sign "::=" is the data type definition symbol, and the sign "|" is the *branch separator.*

The *power set* of a set A is the set of all its subsets and is denoted by $\mathbb{P}\, A$. If A denotes some finite set, then the number of elements in the set is called its *cardinality* or *size*, and is denoted by $\#A$.

One powerful means of defining a set in Z is by stating a property that distinguishes its members from other values of the same type. Suppose D denotes some declarations, P a predicate constraining the values and E an expression denoting a term; then an expression of the form

$$\{D \mid P \bullet E\}$$

is called a *set comprehension* term, and it denotes a set of values consisting of all values of the term E for everything declared in D satisfying the constraint P. The vertical bar separating the declaration from the constraint is called a *constraint bar* and it separates the declaration from the constraint. The *heavy dot* separates the constraint from the term. The whole is enclosed in braces. The set comprehension notation will be used throughout the specification of the RM/V2 to define the result of the relational operations, more precisely to define the set of rows returned by an operation. Reading a set comprehension term aloud may cause a bit of difficulty, especially in the case of long and multiple predicates. For instance, the resulting set of rows of the *selection* operator is defined as follows:

$$\begin{array}{l} SRel!.Rel = \{r : ROW \\ \qquad \mid (r \in (RDM\ REL?).Rel) \\ \qquad \land (\forall i : \mathbb{N} \mid (i \in 1 \mathrel{..} \#AL?) \\ \qquad\qquad \bullet ((r.row\,(AL?(i))),\ VL?(i)) = PL?(i)) \\ \qquad \bullet\ r\} \end{array}$$

This set comprehension might be read: "The set formed of values of the expression r of type ROW, such that r is a row in the set of rows Rel of the input relation $REL?$ having the property that for any natural number i between 1 and the length of the list $AL?$ then the relation between the value of each attribute of that row and the corresponding value from the list $VL?$ should be the same relation as the corresponding relation in the list $PL?$".

This example also shows how we can build compound predicates from simple ones using the logical connectives. The logical connectives Z uses are the ones defined in the set theory: *negation*, *disjunction*, *conjunction*, *implication* and *equivalence*. The general form of a predicate formed by applying the *universal quantifier*

$$\mid \ \forall D \mid P \bullet Q$$

is equivalent to

$$\mid \ \forall D \bullet (P \Rightarrow Q)$$

An existential quantification formed by applying the *existential quantifier* can also be recast in various ways. The general form

$$\mid \ \exists D \mid P \bullet Q$$

is equivalent to

$$\mid \ \exists D \bullet (P \land Q)$$

The notion of there being only one thing of a certain kind is formalized using the *unique existential quantifier*, denoted $\exists_1$. The general form of an *unique existential quantification* is

$$\mid \ \exists_1 \mid P \bullet Q$$

where D represents declarations, P represents a predicate acting as the constraint and Q represents the predicate being quantified.

2.3.2 Relations and Functions

A set of ordered pairs of the same type is called a *relation*. Relations are very useful and occur often in Z specifications. If X and Y are two sets, then the set $X \longleftrightarrow Y$ of all relations from X to Y is defined to be $\mathbb{P}(X \times Y)$. The notation

$$\mid \ X \leftrightarrow Y == \mathbb{P}(X \times Y)$$

is the generic definition for relations, the one that summarizes the whole family of possible definitions.

The *domain* of a relation is the set of first members of the pairs in the relation. Suppose

$$\mid \; R : X \leftrightarrow Y$$

then dom R is the set

$$\mid \; \{x : X \mid (\exists\, y : Y \bullet x \mapsto y \in R\}$$

where $x \mapsto y$ is a graphic way of expressing the ordered pair (x, y).

The *range* of a relation is the set of second members of the pairs in the relation. For the same relation R, ran R is the set

$$\mid \; \{y : Y \mid (\exists\, x : X \bullet x \mapsto y \in R\}$$

A relation having a strict denotation can be defined by a set comprehension notation. A relation can also be defined by enumeration. If R and S are two relations declared as follows:

$$\mid \; R : X \leftrightarrow Y;\, S : Y \leftrightarrow Z$$

then their *composition*, denoted $R; S$ is defined as:

$$\{x : X; z : Z \mid (\exists\, y : Y \bullet (xRy \land ySz)) \bullet x \mapsto z\}$$

Given a relation, we can create a smaller relation by considering only a part of the original domain. The *domain restriction* operator which is used, $\lhd$, appears between the set that we wish to restrict the domain to and the name of the relation being restricted. There is a complementary *range restriction* operation, $\rhd$, which creates a smaller relation by considering only a part of the original range. We can give precise definitions for domain and range restrictions for any relation R of type $X \leftrightarrow Y$. Suppose S is a set of elements of the domain type, X, and T is a set of elements of the range type, Y, then:

$$S \lhd R = \{x : X, y : Y \mid x \in S \land x \mapsto y \in R \bullet x \mapsto y\}$$
$$R \rhd T = \{x : X, y : Y \mid y \in T \land x \mapsto y \in R \bullet x \mapsto y\}$$

If instead of restricting the domain (or range) to a certain set we want to remove a particular set of elements from the domain (or range), we can use the *domain (or range) anti-restriction*. Their definition is as follows:

$$S \ntriangleleft R = \{x : X, y : Y \mid x \notin S \land x \mapsto y \in R \bullet x \mapsto y\}$$
$$R \ntriangleright T = \{x : X, y : Y \mid y \notin T \land x \mapsto y \in R \bullet x \mapsto y\}$$

A *function* between two sets X and Y is a relation between those sets that has the special property that each member of the from-set is related to at most one member of the to-set. The most general sort of functions as defined above is known as a *partial function.* It is called partial since the domain of the function need not to be the whole of the source. Partial functions are represented by a single arrow with a bar across:

$$Row : ANAME \nrightarrow VALUE$$

Partial functions can be defined with a generic definition:

$$\begin{aligned} X \nrightarrow Y == \{&R : X \leftrightarrow Y \\ &\mid (\forall x : X, y, z : Y \bullet x \mapsto y \in R \land x \mapsto z \in R \Rightarrow y = z)\}\end{aligned}$$

One special type of functions are those for which the domain is the whole of the from-set. These are called *total functions* and their definition is:

$$X \longrightarrow Y == \{f : X \nrightarrow Y \mid \operatorname{dom} f = X\}$$

A function in which the second members of the pairs are unique is called an *injection*. The set of all injections from X to Y is defined as follows:

$$\begin{array}{|l} X \rightarrowtail\mapsto Y == \{f : X \twoheadrightarrow\mapsto Y \mid (\forall x1, x2 : X \mid x1 \in \mathrm{dom} f \land x2 \in \mathrm{dom} f \\ \qquad\qquad\qquad \bullet fx1 = fx2 \Rightarrow x1 = x2)\} \end{array}$$

If the domain of an injection is the whole of the from-set, the injection is called a *total injection*:

$$\begin{array}{|l} X \rightarrowtail Y == \{f : X \rightarrowtail\mapsto Y \mid \mathrm{dom} f = X\} \end{array}$$

A *surjection* is a function whose range is the whole of the to-set. As with injectivity, we have partial or total surjective functions. As before the arrows which are used for these type of functions can be defined formally:

$$\begin{array}{|l} X \twoheadrightarrow\mapsto Y == \{f : X \nrightarrow Y \mid \mathrm{ran} f = Y\} \\ X \twoheadrightarrow Y == \{f : X \longrightarrow Y \mid \mathrm{ran} f = Y\} \end{array}$$

Finally, a function which is both injective and surjective is known as a bijection. The bijective arrow has both a tail and an extra head:

$$X \rightarrowtail\!\!\!\!\twoheadrightarrow Y == (X \twoheadrightarrow Y) \cap (X \rightarrowtail Y)$$

An ordered collection of values of the same kind is called a *sequence*. The notion of sequence can be derived fron the notion of partial functions as follows. A sequence of values of type X is a partial function from natural numbers $\mathbb{N}$ to X whose domain is $1..n$, where n is the length of the sequence. Formally we can define the set of all sequences of values of type X as follows:

$$\operatorname{seq} X == \{f : \mathbb{N} \nrightarrow X \mid (\exists n : \mathbb{N} \bullet \operatorname{dom} f = 1..n)\}$$

The type of any sequence of values of type X is $\mathbf{P}(Z \times X)$.

Z allows the specifier to extend the mathematical language in various ways. New functions and relations on values of existing types can be introduced by an *axiomatic description* as follows:

$$\begin{array}{|l}
empty\_rel : RELATION \rightarrow RELATION \\
\hline
\forall R : RELATION \\
\qquad \bullet (empty\_rel\ R).Rel = \varnothing \\
\qquad \wedge (empty\_rel\ R).Att = R.Att
\end{array}$$

The first part of this description declares the variable *empty_rel* to be a total function which takes a relation and returns another relation, that is the empty relation. The second part of the description is the predicate which defines the constraints that an empty relation has an empty set of rows and a set of attribute names.

Since functions are a special type of relations, all the operations which are defined for relations can also be applied to functions. One further operator which is used to update a function with new information and which guarantees that the result is a function is *function overriding.* A *generic* definition of this operator would be like follows:

$$
\begin{array}{|l}
[X, Y] \\
\hline
\_ \oplus \_ : ((X \nrightarrow Y) \times (X \nrightarrow Y)) \longrightarrow (X \nrightarrow Y) \\
\hline
\forall f, g : X \nrightarrow Y \bullet f \oplus g = g \cup ((\operatorname{dom} g) \mathbin{\triangleleft\!\!\!-} f) \\
\hline
\end{array}
$$

This generic definition starts by naming the generic parameters in square brackets. The declaration shows that the symbol being defind is to be used as an infix notation. The function '$\oplus$' is a total function that takes as arguments two functions of the same type, and returns a function of the same type. The predicate defines the value of the result for any two functions f and g of type $\mathbb{P}(X \times Y)$.

2.3.3 Schemas and Specifications

In Z, a *schema* is a piece of mathematical text describing some property of the system being specified. It has a *declaration part* which declares some local variables and a *predicate part* which expresses some requirements on the values of these variables. The name of a schema is used in a specification to refer to the mathematical text. A schema can be written in a vertical or a horizontal format. The former is preferred since usually there are more than a few declarations and predicates.

Schemas can be used to specify the state space of the system or the operations which are defined. In case of a schema used to describe the *abstract states* of the system, the signature and the property together define the *data space.* The property is sometimes called the *data invariant.* Examples of schemas describing properties of the admissable states of the RM/V2 are *RELDATAMODEL*, *RELATION* and *ROW*. The *RELDATAMODEL* schema is used to describe the state on which the requirements for the operations of the RM/V2 are formalized. An operation on an abstract state is called an *abstract operation.* The following conventions about names apply for schemas:

1. Undashed names are used to denote the values of the components of the state before the operation, the *starting state*.

2. Dashed names are used to denote the values of the same components after the operation, the *ending state*.

3. Names ending with a question mark are used to denote the values of inputs to the operation.

4. Names ending with an exclamation mark are used to denote the values of outputs from the operation.

The predicates of a schema concerning only the input value and the starting state define the *preconditions* of the schema. They define the situation that must exist before the operation can behave as it is defined. The other predicates concerning input, output and ending state define the *postconditions* of the schema. They define the situation that must exist after the operation behaved in this way.

Z uses a set of conventions that are intended to reduce the amount of specification we see, to help not repeating the specification of a property and to draw attention to the main points of the behaviour being described. One of these conventions is *schema decoration.* It provides a systematic way of introducing the dashed names of the components of a schema together with the data invariant on them. Another convention is the *schema inclusion* which help us to reuse the name of one schema in the declaration part of another. When a schema is included in a declaration part of another, it brings all its declarations into the declaration part of the new schema. The schema *DIVISION* includes in its declaration part the schema *PROJECTION* and makes use of all declarations of this schema.

Another simplification is achieved by the Δ and Ξ conventions. These are used by schemas defining operations on data types and have two copies of their state variables: undecorated variables corresponding to the state of the data type before the operation and dashed variables corresponding to the state after the operation. The schema $\Delta State$ is defined as the union of State and State':

```
┌─ΔState ──────────────────────────────
│ State
│ State'
└──────────────────────────────────────
```

The Δ is always used to suggest change. In our specification we use this convention for the definition of the manipulative operations which affect the state space of the model. The RM/V2, like all the other data models, has operations which access information in the state without changing the state at all. This fact is specified by using the $\Xi State$ schema which has the following definition:

$$\begin{array}{|l}
\multicolumn{1}{l}{\Xi State} \\
\hline
\Delta State \\
\hline
\theta State = \theta State' \\
\hline
\end{array}$$

In the specification of the RM/V2, the Ξ convention is used for the definition of the retrieval operations which do not modify the state space of the data model.

The schema calculus of Z allows us to build new schemas combining existing schemas. *Schema conjunction, disjunction, negation, implication, quantification, piping, composition* are examples of logical connectives which can be used to create a new signature by merging the signatures of the participant schemas, and to create a new property by combining the properties of the participating schemas.

Chapter 3

Survey of Related Work

3.1 Introduction

The researchers in the database field have been always convinced of the benefits of using formal specification techniques to describe database systems. However, they were faced with a whole host of specification techniques and languages from which to choose. Questions like: "Should we start from basic set theory and first-order predicate logic and devise our own approach ?", "Should we use different techniques for different aspects of database systems ?" or "Which are the relative merits of model-based and algebraic techniques ?" generated many debates on this subject. The decisions that have been made were based partly on functionality, partly on familiarity and partly on politics. This chapter is intended to be a survey of the work done on specification for database systems during the last decade. It presents a set of specifications of different aspects of relational databases, most of them using Z or VDM as the specification language. It also presents specifications of an object-oriented data model and some related Z case studies.

3.2 Formal Specifications for Data Models

One of the first attempts to formalize the relational data model was done by C.J.Date [Dat83, Dat86]. The author summarizes the structure of a relational database by means of a set of production rules. Using an abstract syntax, he defines the basic concepts of the relational data model: domain, attribute, tuple, relation, keys, relational operations, relational rules. The definition of these concepts makes use of plenty of informal explanatory text which completes the definition. For instance, the definition of candidate keys is: $\langle candidate - key \rangle ::= \langle attribute - name - set \rangle$, but the uniqueness and minimality properties of candidate keys are stated informally. The definition of the relational rules and relational retrieval operations is also explained informally. The paper does not include the specification of the manipulative operations and considers the relational data model as defined at that moment. These semi-formal definitions do not make it possible to proove the well-known relational properties.

In [Su85], a brief definition of the algebra which underlies the relational approach to databases is presented. Using the notations of set theory, extended slightly to provide a means of naming mathematical structures, the authors formalize the base concepts of the relational algebra. They also give an account of the simpler normal forms for relational schemes, functional dependencies and decompositions. The paper does not specify the naming technique and its implications in defining the relational model concepts. It also does not define all the relational retrieval operations, the integrity rules and the relational manipulative operations. The authors do define some of the relational properties, but they don't provide proofs for them.

Another formalization of the relational data model is given in [Bjö82]. The authors consider two possible representations of a row (tuple): as a list of values and as a mapping from attribute names to values. They try to illustrate the consequences of choosing one over the other by exemplifying both alternatives in the definition of the relational operations. The paper also presents the definition of a predicate calculus based query language. The preconditions for the relational operations, defined as their syntactic well-formedness, are simplified and do not include all conditions that should be satisfied by the relational operations.The specification of the relational data model does not include concepts like keys, integrity rules and manipulative operations.

A complete formalization of the relational algebra and functional dependencies is presented in [Kan90]. Using predicate logic, the author specifies the syntax and the semantics of relational retrieval operators and of functional dependencies. Different properties and theorems of the relational algebra and its declarative counterpart, the relational calculus query language, are introduced and exemplified. Another part of this work deals with deductive data models and first-order theories. The article does not include the specification of the relational integrity rules and manipulative operators.

In [Miš92], a formal definition of the data dictionary for an extended entity-relationship data model is described using the Z notation. The authors define the basic objects of the model like entities and associations, cycles, cardinality, and relationships like generalisation and specialisation or identification dependence. A schema transformation from the extended entity-relationship data model to the relational data model is presented. Each object (entity or association) is transformed to a relation, attributes of objects are allocated to their appropriate relations, and the key attributes of relations are determined. The paper does not include update and retrieval operations and integrity constraints.

A similar paper which present the entity-relationship model expressed in Z is [Jos91]. The authors take the entity-relationship data model of a CASE tool, and provide a systematic translation of its diagrams into Z. They demonstrate how the expressiveness and precision of structured methods can be enhanced by specifying in Z further constraints on the data model and the effect of the transactions on the system state. The paper provides a style of writing Z specifications that could be easily be adopted by someone familiar with entity-relationship modelling. The same authors present in [Jose91] a library of Z schemas for use in entity-relationship modelling. They also demonstrate how to use the library by instantiating the schemas.

The relationship between Z and the relational algebra is presented in detail in [Die90]. The authors show that the relational algebra can be embedded into Z. A formal semantics for Z based on naive set theory is introduced. Using the notation of a script, which is a set of definitions in which a name is given to expressions, the authors construct a mapping ξ from R-scripts, i.e. scripts in the relational algebra, to Z-scripts, and a partial mapping ϕ the other way around, such that ξ and ϕ are semantics preserving, $\phi \circ \xi$ is the identity on R-script and $\xi \circ \phi$ is a subset of the identity on Z-scripts. The approach opens the possibility of a modular way of specifications, which is important for the reusability of specifications. In this approach the semantics of a schema is a possible infinite table, and there the link with the relational algebra occurs.

A formal specification of an object-oriented data model using the Z specification language is presented in [Cha92]. The model supports all the essential features found in existing object-oriented data models. More important, it simultaneously supports multiple inheritance, method overloading together with static type checking. The specification demonstrates the use of Z as a formal technique in an area where such a definition is greatly needed.

The BROOM data model [Nor92] combines features of the object-oriented, entity-relationship, semantic and relational data model. The author demonstrates the use of an intermediate level of specification by presenting a meta-circular description of the structural part of the data model and indicating how this may then be transformed into a formal specification in the Z language.

3.3 Formal Specifications for Databases

The use of VDM in the formal specification and design of a program to implement simple update operations on a binary relational database called NDB is described in [Wal90]. The author presents first an initial specification and then transforms it in a rigorous way through the careful introduction of design detail in the form of data structures and operations until an implementation is reached. This single level description of NDB is the starting point in [Fit90], where a case study in the modular structuring of this "flat" specification is presented. The goals are the effective separation of concerns within a specification and the module generality. In addition to the flat specification of NDB introduced by [Wal90], the authors present a second specification which makes use of an n-ary relation module, and a third one which uses an n-ary relation module with type and normalization constraints. They demonstrate the reusability of their modules, and also outline specifications for an n-ary relational database with normalization constraints (RDB), and an n-ary relation database with a two-level type hierarchy and no normalization constraints (IS/1). A Z approach in specifying the same database is presented in [Hay92]. The paper provides a comparison of the VDM approach taken in [Fit90] and the approach taken in developing Z specifications for the same systems. After presenting an equivalent Z specification for NDB, the author overviews the differences in the VDM and Z approaches as highlighted by the database examples, and underlines some of the differences between the two specification languages related to this case study.

The design specification of an early version of IST's integrated project support environment, known as the ISTAR database management system, is described in [Sha90]. After describing the informal design requirements that were produced prior to the development of this database system, the author presents an analysis of the requirements using entity relationship modelling and the various database operations are identified. From this a VDM specification structure is derived.

Another aspect of specifications for database systems is related to database applications. A complete description of a general method for the specification of relational database applications using Z is described in [Bar91, Bar93]. 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. The paper also addresses more advanced features such as transactions, sorting of results, aggregate functions, etc. Some features of the relational data model itself are specified as predefined operators which simplify the specifications. The idea of the paper is to formalize the design of database transactions (applications) in a way that can be used by practitioners in the development of real world applications. The use of the method is illustrated by a simple example application. The formal specification of relational database applications and the formal specification of the relational data model are complementary. They both allow for the standardization of the database design process.

Traditionally, the description of database applications consists of two parts, the database schema and a more or less formal description of the application functions. In [Saa91], the authors present an object-oriented approach to integrate both aspects. They describe a formal model that models objects as processes that can be observed through attributes. Based on this model, the language TROLL is introduced, a logical language for the abstract object-oriented description of information systems. The authors propose a mechanism for defining and handling external views and for queries that fit in the framework.

An algebraic approach to the specification of relational database systems is described in [Wor91]. The author shows how specifications may be constructed using axiomatizations of transaction semigroups which generate database instances. A general class of tuple-based transactions is defined, and it is shown how unconstrained insertions and deletions may be modelled and axiomatically characterized using this class. A general class of integrity constraints is defined, and examples are given to show both static and dynamic constraints within this class.

An introduction to the use of formal semantics as a means of specifying relational query languages is presented in [Tur85]. The authors provide a formal semantic framework which can be used in tackling different problems like the well-formedness and correctness of queries and other problems of relational query language analysis. The semantic techniques are demonstrated with respect to a form of the relational calculus and also to two query languages SEQUEL (now SQL) and QUEL. For each language a set of recursively defined functions is provided which map syntactically correct constructs on to elements of certain sets.These domains or sets form a second component of any such formal specification. The framework which is developed can be used to formally define any relational query language.

[Dav91] introduces a denotational semantics approach to object-oriented query language definition. An object-oriented database (OODB) is characterized as a semantic domain, so that query expressions can be mapped to their meaning in terms of the OODB. In this research, the authors propose a high-level view of OODBs and an algebraic query language that support query processing studies. The model is applicable to structurally similar database models, which include most object-oriented and semantic databases. A contribution of the denotational definition of the model and query language is the precise characterization of both the intention and the extention of an OODB and the results of queries on the OODB.

Chapter 4

A Specification of the Relational Data Model Using Z

The main purpose of this chapter is to define in a rigorous and precise manner the basic features of the relational data model version 2 (RM/V2) as defined by Codd [Cod90]. We define first the state space of the system, specifying concepts like domains, rows and relations. The definition of keys and integrity rules completes the state invariant of the relational data model. We define next the basic relational operators intended to retrieve and manipulate information from the database.

The structure and behaviour of the relational data model is defined using the Z specification language. The type correcteness of this specification has been verified by the FUZZ type-checker.

4.1 The Relational Model State Space

4.1.1 Domains, Rows and Relations

The relational data model consists of three basic components:

- a set of domains and a set of relations;
- operations on relations;
- integrity rules.

Basically, domains are sets of values from which one or more attributes draw their values. In the RM/V2, domains are referred to as extended data types which are intended to capture some of the meaning of the data. In this specification, the basic type

$$[DOMAIN]$$

contains all domains which are defined for a relational database. The basic type

$$[VALUE]$$

contains all values that attributes can have. These are proper values drawn from the domains defined above and marked values which can be used whenever a proper value is missing. These marked values are *Amark* values which stands for *missing-but-applicable* values and *Imark* values for *missing-and-inapplicable* values. The *VALUE* domain can be partitioned into three disjoint subsets *Amark*, *Imark* and *ProperValue*:

$$\begin{array}{|l}
Amark : VALUE \\
Imark : VALUE \\
ProperValue : \mathbb{P}\ VALUE \\
\hline
\langle \{Amark\}, \{Imark\}, ProperValue \rangle \ \mathsf{partition}\ VALUE
\end{array}$$

Manipulation of missing information in databases is an important issue which will be discussed in this specification when dealing with concepts like keys and integrity rules.

Other basic types which will be used throughout the specification are

$$[RNAME, ANAME]$$

which contain the names of relations and attributes, respectively.

The function *domain_of_attribute* associates with an attribute name the domain from which that attribute draws its values:

$$| \quad domain\_of\_attribute : ANAME \nrightarrow DOMAIN$$

This axiomatic description introduces the partial function *domain_of_attribute* as a global variable which becomes part of the global signature of the specification. Optionally an axiomatic description has a predicate part which states the relation between the values of the new variables to each other and to the values of variables that has been declared previously. If the predicate part is absent, like in this case, the default is the predicate *true*. The partial function *domain_of_attribute* will be used whenever we need to check whether or not two attributes are based on the same domain.

The function *values_of_domain* returns the set of values corresponding to a domain:

$$| \quad values\_of\_domain : DOMAIN \nrightarrow \mathbb{P}\ VALUE$$

This function will be used in order to verify whether a certain value is in the set of values corresponding to a domain.

A row can be modelled in different ways: as a mapping from attribute names to values; as a sequence of values, as a tuple of values, etc. Each of these representations has advantages and disadvantages. We will use the following representation for rows:

$$
\begin{array}{|l}
\hline
\textit{ROW} \\
\textit{Row} : \textit{ANAME} \nrightarrow \textit{VALUE} \\
\hline
\forall\, \textit{anm} : \textit{ANAME} \\
\qquad \bullet\ \textit{Row anm} \in \textit{values\_of\_domain}\,(\textit{domain\_of\_attribute anm}) \\
\hline
\end{array}
$$

Row is a variable specified as a partial mapping from attribute names to values. This representation has the following advantages:

- it captures both attribute names and their values and, more often, people prefer to refer to individual row elements using a freely chosen attribute name;
- the order among row attributes is of no importance, so we do not need to have an index to specify the "nextness" of row attributes;
- the mapping implies relations to be at least in the first normal form.

The possible problem of duplicate attribute names in intermediate or final results is solved by the naming technique used in the RM/V2. Whenever an attribute name in an intermediate or final result occurs twice, that attribute name is combined with the name of the relation from where that attribute comes. According to this technique, duplicate attribute names will never occur in intermediate or final results as relation names are unique in the database. When an attribute name comes from two different instances of the same relation we consider those two instances of the same relation name to be different, so we don't have to deal with duplicate attribute names.

A relation is a finite set of rows which have the same attributes. The specification of a relation as a set of rows implies that only proper relations which contain no duplicate rows are modelled, as defined by the RM/V2.

$$\begin{array}{|l}
\hline
\textit{RELATION} \\
\textit{Rel} : \mathbb{F}\, \textit{ROW} \\
\textit{Att} : \mathbb{F}\, \textit{ANAME} \\
\hline
\forall r : \textit{ROW} \mid r \in \textit{Rel} \bullet \operatorname{dom} r.\textit{Row} = \textit{Att} \\
\hline
\end{array}$$

The empty relation is defined relative to a type of relations, that is to a set of relations having the same attribute name set:

$$
\begin{array}{|l}
empty\_rel : RELATION \longrightarrow RELATION \\
\hline
\forall R : RELATION \\
\qquad \bullet\ (empty\_rel\ R).Rel = \emptyset \\
\qquad \wedge\ (empty\_rel\ R).Att = R.Att
\end{array}
$$

If R is a relation, then $empty\_rel\ R$ will return the empty relation which has the same set of attribute names as R.

We now introduce an operator which defines the property of a relation $R1$ being a subrelation of a relation $R2$. According to this, $R1$ is a subrelation of $R2$ if it has the same attribute name set and its rows are a subset of those of $R2$:

$$
\begin{array}{|l}
\_subrel\_ : (RELATION \times RELATION) \longrightarrow BOOL \\
\hline
\forall R1, R2 : RELATION \\
\qquad \bullet\ R1\ subrel\ R2 = True \Leftrightarrow \\
\qquad\qquad R1.Att = R2.Att\ \wedge \\
\qquad\qquad R1.Rel \subseteq R2.Rel
\end{array}
$$

The *universal relation* relative to a type of relations is a relation which contains all the rows whose domain is the attribute name set of those relations.

$$\begin{array}{|l}
universal\_rel : RELATION \longrightarrow RELATION \\
\hline
\forall R : RELATION \\
\qquad\qquad \bullet\ (universal\_rel\ R).Att = R.Att \\
\qquad\qquad \land\ (\forall r : ROW \mid \mathrm{dom}\ r.Row = R.Att \\
\qquad\qquad\qquad \bullet\ r \in (universal\_rel\ R).Rel)
\end{array}$$

A relation R is always a subrelation of the universal relation of its type, and it includes the empty relation of its type:

$$\begin{array}{l}
\forall R : RELATION \bullet (empty\_rel\ R)\ subrel\ R = True\ \land \\
\qquad\qquad R\ subrel\ (universal\_rel\ R) = True
\end{array}$$

The *trivial relation* is a relation having no attributes and just a single row, which is the empty mapping. This relation is the only relation having an empty set of attribute names so it is also the *universal relation* of that type.

$$\begin{array}{|l}
trivial\_rel : RELATION \longrightarrow RELATION \\
\hline
\forall R : RELATION \\
\qquad\qquad \bullet\ (trivial\_rel\ R).Att = \varnothing \\
\qquad\qquad \land\ (trivial\_rel\ R).Rel = \varnothing
\end{array}$$

We will use the definitions of the empty relation, universal relation and trivial relation in *Chapter 5*, when some of the properties of the relational model will be proved.

The *size* of a relation is the number of rows it contains:

$$\begin{array}{|l} size : RELATION \longrightarrow \mathbb{N} \\ \hline \forall R : RELATION \bullet size\ R = \#R.Rel \end{array}$$

4.1.2 Keys and Integrity Rules

The concepts of keys and integrity rules will be introduced by a set of functions which return boolean values. We need first to define the boolean type:

$$BOOL ::= False \mid True$$

This free type definition introduces the basic type *BOOL* and two boolean constants, *True* and *False*.

A *candidate key* of a relation is defined as a subset of the attribute set of that relation which has the following time-independent properties:

- *Uniqueness*: In each tuple of the relation, the value of the key uniquely identifies that tuple.

- *Nonredundancy*: No proper subset of the key has the unique identification property.

The *uniqueness* property of a key is defined by an infix function which determines whether a specific set of attribute names uniquely identify the rows of a specific relation.

$$\begin{array}{|l}
\_uniq\_identifies\_rows\_in\_ : (\mathbb{P}\, ANAME \times RELATION) \nrightarrow BOOL \\
\hline
\forall\, atts : \mathbb{P}\, ANAME;\ rel : RELATION \\
\quad \bullet\ atts\ uniq\_identifies\_rows\_in\ rel = True \Leftrightarrow \\
\qquad atts \subseteq rel.Att\ \wedge \\
\qquad (\forall\, r1, r2 : ROW \mid r1 \in rel.Rel \wedge r2 \in rel.Rel \\
\qquad\quad \bullet\ ((atts \lhd r1.Row) = (atts \lhd r2.Row)) \Rightarrow r1.Row = r2.Row)
\end{array}$$

The *domain restriction* $atts \lhd r1$ of the function $r1$ to the set *atts* relates an attribute name to a value if and only if $r1$ relates that attribute name to that value and that attribute name is a member of *atts*. The definition of the *_uniq_identifies_rows_in* is true also for the case of empty relations.

The *_is_key_of_* operator, used to specify that a specific set of attributes of a relation is a candidate key of that relation is defined below:

$$
\begin{array}{|l}
\_is\_key\_of\_ : (\mathbb{P}\, ANAME \times RELATION) \rightarrowtail BOOL \\
\hline
\forall\, atts : \mathbb{P}\, ANAME;\ rel : RELATION \\
\bullet\ atts\ is\_key\_of\ rel = True \Leftrightarrow \\
\qquad\qquad atts\ uniq\_identifies\_rows\_in\ rel = True \land \\
\qquad\qquad \neg\ (\exists\, subsetatts : \mathbb{P}\, ANAME \mid subsetatts \subset atts \\
\qquad\qquad\quad \bullet\ subsetatts\ uniq\_identifies\_rows\_in\ rel = True)
\end{array}
$$

This infix *_is_key_of_* operator takes two parameters: a set of attribute names and a relation. The uniqueness property of the key is specified using the already defined function *_uniq_identify_rows_in_*. The minimality property is specified using a predicate of the form: $\neg\ (\exists S \mid P \bullet Q)$ which is equivalent to $\neg\ (\exists S \mid P \land Q)$. This predicate is true if there are no values of the variables introduced by S so that the property of S is true and both predicates P and Q are true.

For each relation, one and only one candidate key of that relation is designated as the *primary key*. A *foreign key* consists of one or more attributes drawing their values from the domains upon which at least one primary key is defined. A relation can have zero, one or several foreign keys.

Before defining the state space of the RM/V2, we introduce the function *do-*

main_preserving_attribute_map which will be used throught the specification to ensure the fact that corresponding attributes must have the same domain:

$$
\begin{array}{|l}
domain\_preserving\_attribute\_map : (ANAME \rightarrowtail\!\!\!\!\rightarrow ANAME) \longrightarrow BOOL \\
\hline
\forall f : ANAME \rightarrowtail\!\!\!\!\rightarrow ANAME \\
\quad \bullet\ domain\_preserving\_attribute\_map\, f = True \Leftrightarrow \\
\qquad (\forall anm : ANAME \mid anm \in \mathrm{dom}\, f \\
\qquad \bullet\ domain\_of\_attribute\, anm = domain\_of\_attribute\, (f\ anm))
\end{array}
$$

The schema *RELDATAMODEL* defines the major data structure of the relational data model which is a set of uniquely named relations, each relation having a unique primary key and the set of foreign keys defined for the model:

$$
\begin{array}{|l}
\_RELDATAMODEL\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \\
RDM : RNAME \rightarrowtail\!\!\!\!\rightarrow RELATION \\
primarykey : RNAME \mathrel{+\!\!\!\rightarrow} \mathbb{P}\, ANAME \\
foreignkey : (RNAME \times RNAME) \mathrel{+\!\!\!\rightarrow} (ANAME \rightarrowtail\!\!\!\!\rightarrow ANAME) \\
\hline
\mathrm{dom}\, RDM = \mathrm{dom}\, primarykey \\
\forall rnm : RNAME \mid rnm \in \mathrm{dom}\, RDM \\
\qquad \bullet\ (primarykey\ rnm)\ is\_key\_of\ (RDM\ rnm) = True \\
\mathrm{dom}\, foreignkey \subseteq ((\mathrm{dom}\, RDM) \times (\mathrm{dom}\, RDM)) \\
\forall rnm1, rnm2 : RNAME; f : ANAME \rightarrowtail\!\!\!\!\rightarrow ANAME \\
\mid rnm1 \in \mathrm{dom}\, RDM \wedge rnm2 \in \mathrm{dom}\, RDM \wedge \\
\quad f = foreignkey\, (rnm1, rnm2) \\
\bullet\ \mathrm{dom}\, f \subseteq (RDM\ rnm1).Att \wedge \mathrm{ran}\, f = primarykey\, rnm2 \wedge \\
\quad domain\_preserving\_attribute\_map\, f = True \\
\hline
\end{array}
$$

The variable *RDM* of this schema is a partial injective function which associates with a relation name a certain relation in the database. It is partial because there could be relation names which do not designate existing relations and it is injective because two different relation names can not designate the same relation. The *primarykey* variable is a partial function which associates with a relation name a unique (designated) primary key which must be one of the candidate keys of that relation. The variable *foreignkey* is a function which associates with a pair of relation names the mapping between the foreign key of the first relation and the corresponding primary key of the second relation. This mapping should preserve the domain of corresponding attributes.

Another part of the relational data model structure is the *catalog* which holds the database description. This description includes information about domains, relations, integrity rules and views. An important property of the relational data model is that both the database and its description are perceived as a collection of relations. Thus, the same relational language that is used to interrogate and modify the database can be used to interrogate and modify the database description. Hence, we do not explicitly specify the catalog as it has the same structure as any relation and all defined relational operations can be applied on it.

The relational data model includes two general integrity rules which implicitly or explicitly define the set of consistent database states, or changes of state, or both. Other integrity constraints can be specified, for example, in terms of functional dependencies during database design. We restrict ourselves to the integrity rules formulated by Codd.

In order to express these rules we have to deal with "missing values" that attributes can take. The RM/V2 places heavy emphasis on the semantic aspects of missing information. It deals with the inapplicability of certain properties to some objects. In RM/V2, a missing value can be a value that is either not known at the time (*missing-but-applicable*) or does not apply to a given instance of the row (*missing-and-inapplicable*). These missing values are represented by the constants *Amark* and *Imark* defined earlier in this specification. An example of the missing values for the attributes *SALARY* and *SALES_COMMISSION* of a relation *EMPLOYEE* would be as follows:

1. If an employee has a *missing-but-applicable* present salary, his or her record would have an *Amark* in the salary column.

2. If an employee has an *inapplicable* sales commission (such an employee does not sell products at this time), his or her record would have an *Imark* in the commission column.

The first integrity rule, *entity integrity* , states that no component of a primary key is allowed to have a missing value of any type and no component of a foreign key is allowed to have an *Imark* value.

The function *ENTITY_INTEGRITY_RULE* checks the entity integrity rule for all relations of the relational data model:

$$
\begin{array}{|l}
ENTITY\_INTEGRITY\_RULE : RELDATAMODEL \rightarrow BOOL \\
\hline
\forall\, rdm : RELDATAMODEL \bullet \\
\quad ENTITY\_INTEGRITY\_RULE\; rdm = True \Leftrightarrow \\
\qquad (\forall\, rnm1 : RNAME \\
\qquad \mid rnm1 \in \operatorname{dom} rdm.RDM \\
\qquad \bullet (\forall\, r : ROW \\
\qquad\quad \mid r \in (rdm.RDM\; rnm1).Rel \\
\qquad\quad \bullet\; Amark \notin \operatorname{ran}((rdm.primarykey\; rnm1) \lhd r.Row) \wedge \\
\qquad\qquad Imark \notin \operatorname{ran}((rdm.primarykey\; rnm1) \lhd r.Row))) \\
\qquad \wedge \\
\qquad\quad (\forall\, fk : \mathbb{P}\, ANAME \\
\qquad\quad \mid (\exists\, rnm2 : RNAME \mid rnm2 \in \operatorname{dom} rdm.RDM \wedge \\
\qquad\qquad fk = \operatorname{dom}(rdm.foreignkey(rnm1, rnm2))) \\
\qquad\quad \bullet (\forall\, r : ROW \\
\qquad\qquad \mid r \in (rdm.RDM\; rnm1).Rel \\
\qquad\qquad \bullet\; Imark \notin \operatorname{ran}(fk \lhd r.Row)))
\end{array}
$$

The function *ENTITY_INTEGRITY_RULE* returns a *True* value if and only if all the relations which are part of the data model satisfy the two conditions: no missing value for a primary key and no *Imark* value for a foreign key. The value of a primary or foreign key is specified using the domain restriction operator which restricts the set of values of a row to those of the primary or foreign key attributes. The condition for primary keys is that *Amark* and *Imark* should not be in the set of values of a primary key. For foreign keys, the condition is that *Imark* value is not allowed as a value of a foreign key attribute.

The second integrity rule, *referential integrity*, is concerned with foreign keys. The referential integrity rule states that, for each distinct unmarked foreign key value in a relational database, there must exist in the database an equal value of a primary key. If the foreign key is composite, that is it is a set of attribute names, those components that are themselves foreign keys and unmarked must exist in the database as components of at least one primary key value. The function *REFERENTIAL_INTEGRITY_RULE* verifies whether or not a database satisfies this integrity rule:

$$
\begin{array}{|l}
\mathit{REFERENTIAL\_INTEGRITY\_RULE} : \mathit{RELDATAMODEL} \longrightarrow \mathit{BOOL} \\
\hline
\forall\, \mathit{rdm} : \mathit{RELDATAMODEL} \bullet \\
\quad \mathit{REFERENTIAL\_INTEGRITY\_RULE}\; \mathit{rdm} = \mathit{True} \Leftrightarrow \\
\quad\quad (\forall\, \mathit{rnm1}, \mathit{rnm2} : \mathit{RNAME} \mid (\mathit{rnm1}, \mathit{rnm2}) \in \operatorname{dom} \mathit{rdm.foreignkey} \\
\quad\quad \bullet (\forall\, r1 : \mathit{ROW} \mid r1 \in (\mathit{rdm.RDM}\; \mathit{rnm1}).\mathit{Rel} \\
\quad\quad\quad \bullet (\exists\, r2 : \mathit{ROW} \mid r2 \in (\mathit{rdm.RDM}\; \mathit{rnm2}).\mathit{Rel} \\
\quad\quad\quad\quad \bullet (\forall\, \mathit{anm1} : \mathit{ANAME} \\
\quad\quad\quad\quad\quad \mid \mathit{anm1} \in \operatorname{dom}(\mathit{rdm.foreignkey}\,(\mathit{rnm1}, \mathit{rnm2})) \\
\quad\quad\quad\quad\quad \bullet (r1.\mathit{Row}\; \mathit{anm1}) = \\
\quad\quad\quad\quad\quad\quad (r2.\mathit{Row}((\mathit{rdm.foreignkey}\,(\mathit{rnm1}, \mathit{rnm2}))(\mathit{anm1}))) \\
\quad\quad\quad\quad\quad\quad \vee\ (r1.\mathit{Row}\; \mathit{anm1}) = \mathit{Amark}))))
\end{array}
$$

These integrity rules define constraints on the values that primary and foreign keys can take. The *referential integrity* rule also implicitly defines the possible actions that could be taken whenever updates, insertions and deletions are made. All these issues will be discussed when dealing with manipulative operators.

4.2 The Relational Algebra

Relational algebra is a collection of operations to manipulate relations. These operators are the basic operators intended to retrieve information from the database and the manipulative operators concerned with making changes to the contents of the database.

In the specification of these operations we check the syntactic and semantic correcteness against the proper data part of the relational model, that is the set of values that attributes can take. Another possibility is to check against the schema definition of the model, which is a catalog describing all existing relations.

4.2.1 The Basic Operators

The basic operators of the RM/V2 include relational **union, intersection, difference, projection, theta-selection, theta-join, natural-join,** equi-join, and

division. The definition of each of these operators is a schema such that:

1. It includes the $\Xi RELDATAMODEL$ schema which specifies that the state space of the system does not change;

2. It declares the input variables and output variables of the operation. Input variables are those whose names are followed by '?' and output variables are those whose names are followed by '!';

3. It declares the *pre-conditions* of the operation and defines the results of the operation. The resulting relation is of type *RELATION*, but it is not stored explicitly in the system, so it has no name. The variable *Rel* of the resulting relation indicates the resulting set of rows; it is defined using the set comprehension notation.

The relational union, intersection and difference operations require that operand relations be union compatible. Two relations $R1$ and $R2$ are *union compatible* if they have the same arity (degree) and it is possible to establish at least one mapping f between the attributes of $R1$ and those of $R2$ that is one-to-one, and with the property that, for every attribute $A1$ of $R1$, then $A1$ and $f(A1)$ draw their values from a common domain. These operators also require that the attribute alignment for their two operands be in conformity with one of the mappings that guarantees union compatibility. The union compatibility property is specified using the infix function, *domain_compatible* which checks whether two attribute name sets are domain compatible, that is they have the same number of elements, there exists a one-to-one correspondence between all elements of the two attribute name sets and corresponding attributes draw their values from the same domain:

$$
\begin{array}{|l}
\_domain\_compatible\_ : \mathbb{P}\, ANAME \times \mathbb{P}\, ANAME \longrightarrow BOOL \\
\hline
\forall\, atts1, atts2 : \mathbb{P}\, ANAME \\
\quad \bullet\ atts1\ domain\_compatible\ atts2 = True \Leftrightarrow \\
\qquad (\exists f : atts1 \rightarrowtail\!\!\twoheadrightarrow atts2 \\
\qquad\quad \bullet\ domain\_preserving\_attribute\_map\, f = True)
\end{array}
$$

The mapping between corresponding attributes of the two domain compatible attribute sets is specified by the bijective function f which preserves the domain of the corresponding attributes.

The attribute names which occur in the resultant relation can be the attribute names of one of the two relations or they can be indicated by the user. We introduce first the function *rename* which, given a relation and a mapping from one attribute name set to another, returns the identical relation (in terms of values) with the attributes renamed according to the input mapping.

$$
\begin{array}{|l}
rename : RELATION \times (ANAME \rightarrowtail ANAME) \longrightarrow RELATION \\
\hline
\forall rel : RELATION;\ anm : ANAME;\ f : ANAME \rightarrowtail ANAME \\
\mid \operatorname{dom} f = rel.Att \land domain\_preserving\_attribute\_map\ f = True \\
\bullet\ (rename\,(rel, f)).Rel = \\
\qquad\qquad \{r, rr : ROW;\ anm : ANAME \\
\qquad\qquad \mid r \in rel.Rel \land anm \in rel.Att \land \\
\qquad\qquad\qquad \operatorname{dom} rr.row = \operatorname{ran} f \land \\
\qquad\qquad\qquad r.row\ anm = rr.row\,(f\ anm) \\
\qquad\qquad \bullet\ rr\} \\
(rename\,(rel, f)).Att = \operatorname{ran} f
\end{array}
$$

Given these requirements, the union operation is the set-theoretic union of two relations:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{UNION}} \\
\hline
\Xi RELDATAMODEL \\
REL1? : RNAME \\
REL2? : RNAME \\
Att\_Map? : ANAME \rightarrowtail\!\!\!\!\!\rightarrow ANAME \\
UANames? : \mathbb{P}\, ANAME \\
URel! : RELATION \\
\hline
REL1? \in \operatorname{dom} RDM \\
REL2? \in \operatorname{dom} RDM \\
(RDM\ REL1?).Att\ \mathit{domain\_compatible}\,(RDM\ REL2?).Att \\
UANames?\ \mathit{domain\_compatible}\,(RDM\ REL1?).Att \\
\operatorname{dom} Att\_Map? = (RDM\ REL1?).Att \\
\operatorname{ran} Att\_Map? = (RDM\ REL2?).Att \\
\exists f1 : (RDM\ REL1?).Att \rightarrowtail\!\!\!\!\!\rightarrow UANames? \land \\
\exists f2 : (RDM\ REL2?).Att \rightarrowtail\!\!\!\!\!\rightarrow UANames? \\
\bullet\ URel!.Rel = (rename((RDM\ REL1?), f1)).Rel \\
\qquad\qquad\quad \cup (rename((RDM\ REL2?), f2)).Rel \\
URel!.Att = UANames? \\
\hline
\end{array}
$$

The input variables for union are the two relation names *REL1?* and *REL2?*, the injective function *Att_Map?* which establishes the one-to-one correspondance between the two attribute name sets, and the attribute name set *UANames?* which contains the attribute names for the resulting relation. The first two conditions express the fact that the input relation names name two relations in the actual system *RDM*. To specify that the two relations should be union compatible we use the *domain_compatible* function defined earlier. The input attribute name set *UANames* also should be domain compatible with the two attribute name sets. The domain of the bijective function *Att_Map?* is the attribute name set of the first relation and its range is the attribute name set of the second relation. The resultant relation *URel!* is the union of the two input relations whose attributes have been renamed according to *UANames?*

The intersection operation selects the common tuples from two relations. Like union, it requires the two relations to be union-compatible. Attributes have to be aligned in the same way as for the union operator.

INTERSECTION

$$
\begin{array}{l}
\Xi RELDATAMODEL \\
REL1? : RNAME \\
REL2? : RNAME \\
Att\_Map? : ANAME \rightarrowtail ANAME \\
IANames? : \mathbb{P}\ ANAME \\
IRel! : RELATION \\
\hline
REL1? \in \operatorname{dom} RDM \\
REL2? \in \operatorname{dom} RDM \\
(RDM\ REL1?).Att\ domain\_compatible\ (RDM\ REL2?).Att \\
IANames?\ domain\_compatible\ (RDM\ REL1?).Att \\
\operatorname{dom} Att\_Map? = (RDM\ REL1?).Att \\
\operatorname{ran} Att\_Map? = (RDM\ REL2?).Att \\
\exists f1 : (RDM\ REL1?).Att \rightarrowtail IANames?, \\
\quad f2 : (RDM\ REL2?).Att \rightarrowtail IANames? \\
\bullet\ IRel!.Rel = (rename((RDM\ REL1?), f1)).Rel \\
\qquad\qquad \cap (rename((RDM\ REL2?), f2)).Rel \\
IRel!.Att = IANames?
\end{array}
$$

The input for intersection is same as for union: two relation names *REL1?* and *REL2?*, the injective function *Att_Map?* which maps the attribute sets of the relations, and the attribute name set *IANames* for the resulting relation. The pre-conditions for intersection are the same as for union, that is the two relation names should name two existing relations which are union compatible, the domain and range of the input attribute name mapping should be the attribute sets of the two relations. The set of rows in the resulting relation is the set intersection of the two sets of rows of the input relations.

The difference operation removes common tuples from the first relation. The relations should be union-compatible and the attributes must be aligned in the same way as for the union operator.

$$
\begin{array}{|l}
\hline
\textit{DIFFERENCE} \\
\Xi \textit{RELDATAMODEL} \\
\textit{REL1?} : \textit{RNAME} \\
\textit{REL2?} : \textit{RNAME} \\
\textit{Att\_Map?} : \textit{ANAME} \mathrel{⤖} \textit{ANAME} \\
\textit{DANames?} : \mathbb{P}\ \textit{ANAME} \\
\textit{DRel!} : \textit{RELATION} \\
\hline
\textit{REL1?} \in \operatorname{dom} \textit{RDM} \\
\textit{REL2?} \in \operatorname{dom} \textit{RDM} \\
(\textit{RDM REL1?}).\textit{Att}\ \textit{domain\_compatible}\ (\textit{RDM REL2?}).\textit{Att} \\
\textit{DANames?}\ \textit{domain\_compatible}\ (\textit{RDM REL1?}).\textit{Att} \\
\operatorname{dom} \textit{Att\_Map?} = (\textit{RDM REL1?}).\textit{Att} \\
\operatorname{ran} \textit{Att\_Map?} = (\textit{RDM REL2?}).\textit{Att} \\
\exists f1 : (\textit{RDM REL1?}).\textit{Att} \mathrel{⤖} \textit{DANames?}, \\
\qquad\quad f2 : (\textit{RDM REL2?}).\textit{Att} \mathrel{⤖} \textit{DANames?} \\
\bullet\ \textit{DRel!}.\textit{Rel} = (\textit{rename}((\textit{RDM REL1?}), f1)).\textit{Rel} \\
\qquad\qquad \setminus (\textit{rename}((\textit{RDM REL2?}), f2)).\textit{Rel} \\
\textit{DRel!}.\textit{Att} = \textit{DANames?} \\
\hline
\end{array}
$$

The input variable $Att\_Map?$ which occurs in the specification of the union, intersection and difference operations shows the mapping between the two attribute sets. Each attribute in the first relation is related to one attribute from the second relation, both attributes having the same domain. We assumed the general case when not all domains of attributes of a relation are distinct, so the user has to explicitly specify the alignment of the attributes. In the case when the domains of the attributes of a relation are distinct, then there is no need for this input variable, the alignment being determined by ensuring that aligned pairs have the same domain.

The Select operation delivers as a result those rows whose values for given attribute names satisfy given predicates. Any of the following ten comparators can be used: equality, inequality, less than, less than or equal to, greater than, greater than or equal to, greatest less than, greatest less than or equal to, least greater than, least greater than or equal to.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SELECTION}} \\
\hline
\Xi \mathit{RELDATAMODEL} \\
\mathit{REL?} : \mathit{RNAME} \\
\mathit{PRED?} : \mathit{ROW} \pfun \mathit{BOOL} \\
\mathit{SRel!} : \mathit{RELATION} \\
\hline
\mathit{REL?} \in \operatorname{dom} \mathit{RDM} \\
\mathit{SRel!}.\mathit{Rel} = \{ r : \mathit{ROW} \\
\qquad\qquad\quad \mid (r \in (\mathit{RDM}\ \mathit{REL?}).\mathit{Rel}) \\
\qquad\qquad\quad \wedge\ \mathit{PRED?}\ r = \mathit{True} \\
\qquad\qquad\quad \bullet\ r \} \\
\mathit{SRel!}.\mathit{Att} = (\mathit{RDM}\ \mathit{REL?}).\mathit{Att} \\
\hline
\end{array}
$$

The first condition specifies that the input relation should name an existing relation. The input function *PRED?* specifies whether or not a row meets some criteria in order to be selected. The result is the relation of rows which satisfy the selection criteria represented by the function PRED?.

Projection operates on a single relation and delivers as a result sub-segments of rows of the named relation. Projection removes all occurrences, except one, of each sub-segment that occurs more than once.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{PROJECTION}} \\
\hline
\Xi RELDATAMODEL \\
REL? : RNAME \\
AL? : \mathrm{iseq}\, ANAME \\
PRel! : RELATION \\
\hline
REL? \in \mathrm{dom}\, RDM \\
\mathrm{ran}\, AL? \subseteq (RDM\ REL?).Att \\
PRel!.Rel = \{r, pr : ROW \mid r \in (RDM\ REL?).Rel \\
\qquad\qquad\qquad \wedge\ pr.Row = \mathrm{ran}\, AL? \lhd r.Row \\
\qquad\qquad \bullet\ pr\} \\
PRel!.Att = \mathrm{ran}\, AL? \\
\hline
\end{array}$$

The input variables are the relation name $REL?$ and the $AL?$ which is an injective sequence containing the attribute names on which the projection is performed. The first condition says that the relation named by the input relation name should exist in the system. The next condition indicates that the input attribute names should be a subset of the attribute set of the relation. The resulting relation is the set of projected rows on the given attributes in the list $AL?$. The order among row elements in the result is the order in which the attribute names were specified in the input injective list. Projection is the only relational operation for which the order among attributes in the resulting relation is important. The specification above does not capture this, as it uses the general specification of a relation for which order among attributes is of no importance.

The cartesian product of two relations delivers as a result, the relation whose rows are the concatenation of each row of the first relation with all rows of the second relation.

$$
\begin{array}{|l}
\hline
\textit{CARTPRODUCT} \\
\Xi \textit{RELDATAMODEL} \\
\textit{REL1?} : \textit{RNAME} \\
\textit{REL2?} : \textit{RNAME} \\
\textit{CPRel!} : \textit{RELATION} \\
\hline
\textit{REL1?} \in \operatorname{dom} \textit{RDM} \\
\textit{REL2?} \in \operatorname{dom} \textit{RDM} \\
\textit{CPRel!}.\textit{Rel} = \{ r1, r2, cpr : \textit{ROW} \\
\qquad\qquad \mid r1 \in (\textit{RDM}\ \textit{REL1?}).\textit{Rel} \wedge r2 \in (\textit{RDM}\ \textit{REL2?}).\textit{Rel} \\
\qquad\qquad\qquad \wedge\ cpr.\textit{Row} = r1.\textit{Row} \cup r2.\textit{Row} \\
\qquad\qquad \bullet\ cpr \} \\
\textit{CPRel!}.\textit{Att} = (\textit{RDM}\ \textit{REL1?}).\textit{Att} \cup (\textit{RDM}\ \textit{REL2?}).\textit{Att} \\
\hline
\end{array}
$$

The predicate part of this operation has three conditions. The first two are preconditions which state that the two named relations should exist in the database. The third one defines the resulting relation using the set comprehension form. The concatenation of rows is specified using the union of the mappings corresponding to the rows. This is possible because the two attribute sets are disjoint sets according to the naming technique used in RM/V2 where a combination of relation name and column name denotes precisely one column in the entire database, provided the column name is the name of a column within that relation. Whenever two relations involved in an operation have a common attribute, we can differentiate between the two attributes by using this combination. The attribute name set of the resulting relation is then the set union of the disjoint attribute name sets of the two relations.

Natural join operates on two relations. The rows of resulting relation are the composition of exactly those rows from the relations which for corresponding attributes have equal values. Only one set of the corresponding attributes is retained in the result. To make the column naming clear and avoid impairing the commutativity, the retained comparand attribute is assigned whichever of the two attribute names occurs first alphabetically. To specify this, we make use of the function *last* which, given a pair of attribute names, returns only the one which occurs last alphabetically in this pair. The important property of this function is that, given the same pair of attribute names it will always return the same attribute name, no matter the order of the attributes.

$$
\begin{array}{|l}
\mathit{last} : \mathit{ANAME} \times \mathit{ANAME} \longrightarrow \mathit{ANAME} \\
\hline
\forall a1, a2 : \mathit{ANAME} \mid a1 != a2 \\
\qquad\qquad \bullet\ \mathit{last}\,(a1, a2) = \mathit{last}\,(a2, a1) \\
\qquad\qquad\qquad \wedge\ \mathit{last}\,(a1, a2) \in \{a1, a2\}
\end{array}
$$

The function *last_set* returns the set of attributes appearing *last* in a set of attribute name pairs.

$$
\begin{array}{|l}
\mathit{last\_set} : \mathbb{P}\,\mathit{ANAME} \times \mathbb{P}\,\mathit{ANAME} \longrightarrow \mathbb{P}\,\mathit{ANAME} \\
\hline
\forall \mathit{attset} : \mathbb{P}\,\mathit{ANAME} \times \mathbb{P}\,\mathit{ANAME} \\
\qquad \bullet\ \mathit{last\_set}\ \mathit{attset} = \{a1, a2 : \mathit{ANAME} \\
\qquad\qquad\qquad \mid (a1, a2) \in \mathit{attset} \wedge a1 != a2 \\
\qquad\qquad\qquad \bullet\ \mathit{last}\,(a1, a2)\}
\end{array}
$$

The specification of the natural join of two relations is as follows:

$$
\begin{array}{|l}
\textit{NATJOIN} \\
\hline
\Xi \mathit{RELDATAMODEL} \\
\mathit{REL1?} : \mathit{RNAME} \\
\mathit{REL2?} : \mathit{RNAME} \\
\mathit{Att\_Map?} : \mathit{ANAME} \text{⇻} \mathit{ANAME} \\
\mathit{NJRel!} : \mathit{RELATION} \\
\hline
\mathit{REL1?} \in \operatorname{dom} \mathit{RDM} \\
\mathit{REL2?} \in \operatorname{dom} \mathit{RDM} \\
\operatorname{dom} \mathit{Att\_Map?} \subseteq (\mathit{RDM}\ \mathit{REL1?}).\mathit{Att} \\
\operatorname{ran} \mathit{Att\_Map?} \subseteq (\mathit{RDM}\ \mathit{REL2?}).\mathit{Att} \\
\forall\, \mathit{anm} : \mathit{ANAME} \mid \mathit{anm} \in \operatorname{dom} \mathit{Att\_Map?} \\
\qquad \bullet\ \mathit{domain\_of\_attribute}\ \mathit{anm} = \\
\qquad\qquad \mathit{domain\_of\_attribute}\ (\mathit{Att\_Map?}\ \mathit{anm}) \\
\mathit{NJRel!}.\mathit{Rel} = \{ r1, r2, \mathit{njr} : \mathit{ROW};\ \mathit{anm} : \mathit{ANAME} \\
\qquad\qquad \mid r1 \in (\mathit{RDM}\ \mathit{REL1?}).\mathit{Rel} \land r2 \in (\mathit{RDM}\ \mathit{REL2?}).\mathit{Rel} \\
\qquad\qquad \land\ (\mathit{anm} \in \operatorname{dom} \mathit{Att\_Map?} \Rightarrow \\
\qquad\qquad\qquad r1.\mathit{Row}\ \mathit{anm} = r2.\mathit{Row}\ (\mathit{Att\_Map?}\ \mathit{anm})) \land \\
\qquad\qquad \mathit{njr}.\mathit{Row} = ((\operatorname{dom} r1.\mathit{Row} \cup \operatorname{dom} r2.\mathit{Row}) \setminus \\
\qquad\qquad\qquad \mathit{last\_set}(\operatorname{dom} \mathit{Att\_Map?}, \operatorname{ran} \mathit{Att\_Map?})) \lhd (r1.\mathit{Row} \cup r2.\mathit{Row}) \\
\qquad\qquad \bullet\ \mathit{njr} \} \\
\mathit{NJRel!}.\mathit{Att} = (\mathit{RDM}\ \mathit{REL1?}).\mathit{Att} \cup (\mathit{RDM}\ \mathit{REL2?}).\mathit{Att} \setminus \\
\qquad\qquad \mathit{last\_set}(\operatorname{dom} \mathit{Att\_Map?}, \operatorname{ran} \mathit{Att\_Map?}) \\
\hline
\end{array}
$$

The input variables are the two relation names, *REL1?* and *REL2?*, and *Att_Map?*, which indicates the attribute names which participate in the natural join. The first two predicates indicate that the two input relation names should name two existing relations. The third predicate specifies that the domain of the mapping *Att_Map?* should be a subset of the attribute set of the first relation, and its range should be a subset of the attribute set of the second relation. The domain from which corresponding attributes draw their values should be the same. The resulting relation is specified using the set comprehension form: it is the set of values taken by the expression $((\mathrm{dom}\, r1.Row \cup \mathrm{dom}\, r2.Row) \setminus last\_set(\mathrm{dom}\, Att\_Map?, \mathrm{ran}\, Att\_Map?)) \lhd (r1.Row \cup r2.Row)$ when the variables $r1$, $r2$, and anm take all values which satisfy both the declaration part and the predicate part. This expression is the domain restriction of the relation $(r1.Row \cup r2.Row)$ to the set $((\mathrm{dom}\, r1.Row \cup \mathrm{dom}\, r2.Row) \setminus last\_set(\mathrm{dom}\, Att\_Map?, \mathrm{ran}\, Att\_Map?)$. This domain restriction for rows in the resulting relation indicates that only one set of participating attributes is retained.

The theta-join of two relations REL1 and REL2 has as result a relation of rows formed by the concatenation of exactly those rows for which given predicates over pairs of attributes (one from each relation) are satisfied.

$$
\begin{array}{|l}
\textit{THETAJOIN} \\
\hline
\Xi \mathit{RELDATAMODEL} \\
\mathit{REL1?} : \mathit{RNAME} \\
\mathit{REL2?} : \mathit{RNAME} \\
\mathit{AL1?} : \mathrm{seq}\, \mathit{ANAME} \\
\mathit{AL2?} : \mathrm{seq}\, \mathit{ANAME} \\
\mathit{PL?} : \mathrm{seq}(\mathit{VALUE} \leftrightarrow \mathit{VALUE}) \\
\mathit{TJRel!} : \mathit{RELATION} \\
\hline
\mathit{REL1?} \in \mathrm{dom}\, \mathit{RDM} \\
\mathit{REL2?} \in \mathrm{dom}\, \mathit{RDM} \\
\#\mathit{AL1?} = \#\mathit{AL2?} = \#\mathit{PL?} \\
\mathrm{ran}\, \mathit{AL1?} \subseteq (\mathit{RDM}\ \mathit{REL1?}).\mathit{Att} \\
\mathrm{ran}\, \mathit{AL2?} \subseteq (\mathit{RDM}\ \mathit{REL2?}).\mathit{Att} \\
\forall i : \mathbb{N} \mid i \in 1 \mathinner{..} \#\mathit{AL1?} \\
\quad \bullet\ \mathit{domain\_of\_attribute}(\mathit{AL1?}(i)) = \\
\qquad \mathit{domain\_of\_attribute}(\mathit{AL2?}(i)) \\
\mathit{TJRel!}.\mathit{Rel} = \{ r1, r2, \mathit{tjr} : \mathit{ROW} \\
\qquad \mid r1 \in (\mathit{RDM}\ \mathit{REL1?}).\mathit{Rel} \land r2 \in (\mathit{RDM}\ \mathit{REL2?}).\mathit{Rel} \\
\qquad \land (\forall i : \mathbb{N} \mid i \in 1 \mathinner{..} \#\mathit{AL1?} \\
\qquad\quad \bullet\ (r1.\mathit{Row}\,(\mathit{AL1?}(i)), r2.\mathit{Row}(\mathit{AL2?}(i))) \in \mathit{PL?}(i)) \\
\qquad \land\ \mathit{tjr}.\mathit{Row} = r1.\mathit{Row} \cup r2.\mathit{Row} \\
\qquad \bullet\ \mathit{tjr} \} \\
\mathit{TJRel!}.\mathit{Att} = (\mathit{RDM}\ \mathit{REL1?}).\mathit{Att} \cup (\mathit{RDM}\ \mathit{REL2?}).\mathit{Att} \\
\hline
\end{array}
$$

The two input lists AL1?, AL2? contain the attributes which participate in the operation. These attributes should be in the attribute set of the corresponding relation. The list PL? contains the predicates that should hold over pairs of attributes. These predicates are actually the comparison operators equality, inequality, less than, less than or equal to, greater than, greater than or equal to, greatest less than, greatest less than or equal to, least greater than and least greater than or equal to. These operators are represented as relations between two values. The three lists should have the same length and the corresponding attributes from AL1? and AL2? should be domain compatible. The rows in the resulting relation are the concatenation of rows from the two relations for which all predicates in PL? are satisfied. The attribute set of the resulting relation is the set union of the atribute sets of the two participating relations.

The equi-join operation is similar to natural join, the same equality operator is used, the only difference being that equi-join retains both sets of attributes that participate in the join. The pre-conditions are similar to the preconditions for theta-join, where the predicate list doesn't exist anymore, all predicates being replaced by the equality operator.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{EQUIJOIN}} \\
\hline
\Xi \mathit{RELDATAMODEL} \\
\mathit{REL1?} : \mathit{RNAME} \\
\mathit{REL2?} : \mathit{RNAME} \\
\mathit{AL1?} : \mathrm{seq}\, \mathit{ANAME} \\
\mathit{AL2?} : \mathrm{seq}\, \mathit{ANAME} \\
\mathit{EJRel!} : \mathit{RELATION} \\
\hline
\mathit{REL1?} \in \mathrm{dom}\, \mathit{RDM} \\
\mathit{REL2?} \in \mathrm{dom}\, \mathit{RDM} \\
\#\mathit{AL1?} = \#\mathit{AL2?} \\
\mathrm{ran}\, \mathit{AL1?} \subseteq (\mathit{RDM}\ \mathit{REL1?}).\mathit{Att} \\
\mathrm{ran}\, \mathit{AL2?} \subseteq (\mathit{RDM}\ \mathit{REL2?}).\mathit{Att} \\
\forall i : \mathbb{N} \mid i \in 1 \mathrel{..} \#\mathit{AL1?} \\
\qquad \bullet\ \mathit{domain\_of\_attribute}(\mathit{AL1?}(i)) = \\
\qquad\qquad \mathit{domain\_of\_attribute}(\mathit{AL2?}(i)) \\
\mathit{EJRel!}.\mathit{Rel} = \{ r1, r2, \mathit{ejr} : \mathit{ROW} \\
\qquad\qquad \mid r1 \in (\mathit{RDM}\ \mathit{REL1?}).\mathit{Rel} \land r2 \in (\mathit{RDM}\ \mathit{REL2?}).\mathit{Rel} \\
\qquad\qquad \land (\forall i : \mathbb{N} \mid i \in 1 \mathrel{..} \#\mathit{AL1?} \\
\qquad\qquad\qquad \bullet\ r1.\mathit{Row}(\mathit{AL1?}(i)) = r2.\mathit{Row}(\mathit{AL2?}(i))) \land \\
\qquad\qquad \mathit{ejr}.\mathit{Row} = r1.\mathit{Row} \cup r2.\mathit{Row} \\
\qquad\qquad \bullet\ \mathit{ejr} \} \\
\mathit{EJRel!}.\mathit{Att} = (\mathit{RDM}\ \mathit{REL1?}).\mathit{Att} \cup (\mathit{RDM}\ \mathit{REL2?}).\mathit{Att} \\
\hline
\end{array}
$$

Division operates on two relations, REL1 and REL2. We treat the relation named by REL2 as representing one set of properties and the relation named by REL1 as representing entities. Entities are defined on the set-difference of the two attribute sets of the relations, and the properties are defined on the set attributes of the divisor REL2. The resultant relation is the set of entities (rows) that possess all the properties specified for REL2.

$$
\begin{array}{l}
\textit{DIVISION} \\
\hline
\Xi \textit{RELDATAMODEL} \\
\textit{PROJECTION} \\
\textit{REL1?} : \textit{RNAME} \\
\textit{REL2?} : \textit{RNAME} \\
\textit{DRel!} : \textit{RELATION} \\
\hline
\textit{REL1?} \in \operatorname{dom} \textit{RDM} \\
\textit{REL2?} \in \operatorname{dom} \textit{RDM} \\
\textit{REL?} = \textit{REL1?} \\
\exists\, \textit{atts} : \mathbb{P}\, \textit{ANAME} \mid \textit{atts} \subseteq (\textit{RDM REL1?}).\textit{Att} \\
\qquad\qquad \bullet\ \textit{atts domain\_compatible}\, (\textit{RDM REL2?}).\textit{Att} \\
\qquad\qquad\qquad \wedge \operatorname{ran} \textit{AL?} = (\textit{RDM REL1?}).\textit{Att} \setminus \textit{atts} \\
\textit{DRel!}.\textit{Rel} = \{ r1, pr : \textit{ROW} \\
\qquad\qquad \mid r1 \in (\textit{RDM REL1?}).\textit{Rel} \wedge pr \in \textit{PRel!}.\textit{Rel} \\
\qquad\qquad \wedge\ (\forall\, r2 : \textit{ROW} \mid r2 \in (\textit{RDM REL2?}).\textit{Rel} \\
\qquad\qquad\qquad \bullet\ pr.\textit{Row} \cup r2.\textit{Row} \subseteq r1.\textit{Row}) \\
\qquad\qquad \bullet\ pr \} \\
\textit{DRel!}.\textit{Att} = \textit{PRel!}.\textit{Att} \\
\hline
\end{array}
$$

This specification of the division operation uses the definition of the projection operation to express the fact that the resultant relation is a subset of the set of rows of the projected relation REL1? on the attributes contained in REL1? but not in REL2?. Consequently, the *PROJECTION* schema is included in this schema definition. The two input relation names should designate two existing relations. The attribute name set of the dividend relation should contain a subset *atts* which is domain compatible to the attribute name set of the divisor relation. The dividend relation *REL*1? is the relation which is projected in the *PROJECT* schema and the attribute list on which projection is to be performed contains the attributes which are in the attribute set of the dividend relation but not in the set *atts*. The attribute name set of the resulting relation is the set difference between the attribute name set of the dividend relation *REL*1? and its subset which is domain compatible to the attribute name set of the divisor relation *REL*2?. A row is in the set of rows of the resulting relation if it is a row in the projected dividend relation *REL*1? and if its composition with all rows in *REL*2? are rows in *REL*1?.

4.2.2 The Manipulative Operators

The manipulative operators are those concerned with making changes to the content of the database. We restrict ourselves to the specification of **insert, update, primary key update with cascaded update, delete** and **delete with cascaded deletion.** The definition of each of these manipulative operators is a schema such that:

1. It includes the expression $\Delta RELDATAMODEL \setminus (primarykey, foreignkey)$ which alerts us to the fact that the schema is describing a *state change* and indicates that the *primarykey* and *foreignkey* variables are not affected by the manipulative operators;

2. It declares the input variables; normally there are no output variables.

3. It indicates the updates to be performed according to the specific operator which is applied and specifies which relations are changed by the operation, and which relations are not.

4. It specifies the conditions that should be satisfied in order to maintain database integrity.

The **insert** operator permits a collection of one or more rows to be inserted into a relation. The position where these rows are inserted is of no importance. A row is withheld from insertion to avoid duplicate rows in the result or to avoid duplicate values in the primary key of the relation. The formal specification of this operator is given by the following schema:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{INSERT}} \\
\Delta \mathit{RELDATAMODEL} \setminus (\mathit{primarykey}, \mathit{foreignkey}) \\
\mathit{REL?} : \mathit{RNAME} \\
\mathit{Rows\_to\_insert?} : \mathbb{P}\, \mathit{ROW} \\
\hline
\mathit{REL?} \in \operatorname{dom} \mathit{RDM} \\
\forall \mathit{ir} : \mathit{ROW} \mid \mathit{ir} \in \mathit{ROWs\_to\_insert?} \\
\bullet\, (\mathit{RDM}\ \mathit{REL?}).\mathit{Att} = \operatorname{dom} \mathit{ir}.\mathit{Row} \\
(\mathit{RDM'}\ \mathit{REL?}).\mathit{Rel} = (\mathit{RDM}\ \mathit{REL?}).\mathit{Rel} \cup \\
\qquad (\mathit{Rows\_to\_insert?} \setminus \\
\qquad \{\mathit{ir} : \mathit{ROW};\ \mathit{att} : \mathbb{P}\, \mathit{ANAME} \\
\qquad\qquad \mid \mathit{ir} \in \mathit{Rows\_to\_insert?} \\
\qquad\qquad \wedge\ \mathit{primarykey}\ \mathit{REL?} = \mathit{att} \\
\qquad\qquad \wedge\ (\exists\, r : \mathit{ROW} \mid r \in (\mathit{RDM}\ \mathit{REL?}).\mathit{Rel} \\
\qquad\qquad\qquad \bullet\ \mathit{att} \lhd \mathit{ir}.\mathit{Row} = \mathit{att} \lhd r.\mathit{Row}) \\
\qquad \bullet\ \mathit{ir}\} \\
(\mathit{RDM'}\ \mathit{REL?}).\mathit{Att} = (\mathit{RDM}\ \mathit{REL?}).\mathit{Att} \\
\forall \mathit{rnm} : \mathit{RNAME} \\
\mid \mathit{rnm} \in \operatorname{dom} \mathit{RDM} \wedge \mathit{rnm} \neq \mathit{REL?} \\
\bullet\ \mathit{RDM'}\ \mathit{rnm} = \mathit{RDM}\ \mathit{rnm} \\
\hline
\end{array}
$$

This schema definition includes an expression based on the $\Delta \mathit{RELDATAMODEL}$ schema and the schema hiding operator to indicate that only the *RDM* variable will change, but not the *primarykey* and *foreignkey* variables. The input consists of the relation name and the set of rows to be inserted. The first predicate in the predicate part specifies that the input relation name should name an existing relation. The next predicate indicates that the rows to be inserted should have the same attribute names as those of the input relation, so consequently they have the same domain. The change of the state space of the database is defined by specifying the resulting relation in which the given set of rows has been inserted. This resulting relation is the union of the set of rows of the initial relation with the set of rows to be inserted minus those rows which are withheld from insertion to avoid duplicate values for the primary key of the relation. All other relations in the database do not change.

The justification of the **update** operator is that sometimes it is necessary to change the values of one or more components of one or more rows that already exist within a relation. The information that must be supplied with this operator consists of the name of the relation to be updated, the specification of the rows in that relation to be updated, the attribute names that identify the row components of these rows to be updated, and the new values for these components. Referential integrity may be damaged if the column to which the update is applied happens to be the primary key of the pertinent relation or a foreign key. The next operator should be used to update a non-primary and non-foreign key attribute only. When updating a foreign key value only, the user should make sure that the new value for this foreign key exists as the value of a primary key defined on the same domain. Otherwise, the update operation will be rejected. The update operator is specified by the following schema definition:

$$
\begin{array}{l}
\textit{UPDATE} \\
\hline
\Delta RELDATAMODEL \setminus (primarykey, foreignkey) \\
REL? : RNAME \\
Key\_values? : \mathrm{iseq}(ANAME \pfun VALUE) \\
New\_values? : \mathrm{seq}(ANAME \pfun VALUE) \\
\hline
REL? \in \mathrm{dom}\, RDM \\
\forall key : ANAME \pfun VALUE \mid key \in \mathrm{ran}\, Key\_values? \\
\quad \bullet (\exists r : ROW \mid r \in (RDM\ REL?).Rel \\
\quad\quad \bullet key = (primarykey\ REL?) \lhd r.Row) \\
\forall val : ANAME \pfun VALUE \mid val \in \mathrm{ran}\, New\_values? \\
\quad \bullet \mathrm{dom}\, val \subseteq (RDM\ REL?).Att \land \\
\quad\quad \mathrm{dom}\, val \cap primarykey\ REL? = \varnothing \land \\
\quad\quad (\forall REL2 : RNAME \mid (REL?, REL2) \in \mathrm{dom}\, foreignkey \\
\quad\quad\quad \bullet \mathrm{dom}\, val \cap \mathrm{dom}(foreignkey\,(REL?, REL2)) = \varnothing) \\
\# Key\_values? = \# New\_values? \\
(RDM'\ REL?).Rel = \\
\quad \{r : ROW \mid r \in (RDM\ REL?).Rel \land \\
\quad\quad (\forall key : ANAME \pfun VALUE \mid key \in \mathrm{ran}\, Key\_values? \\
\quad\quad\quad \bullet (primarykey\ REL?) \lhd r.Row \neq key) \\
\quad \bullet r\} \\
\quad \cup \\
\quad \{r, ur : ROW;\ key : ANAME \pfun VALUE;\ i : \mathbb{N} \\
\quad\quad \mid r \in (RDM\ REL?).Rel \land \\
\quad\quad\quad (primarykey\ REL?) \lhd r.Row = Key\_values\ i \land \\
\quad\quad\quad ur.Row = r.Row \oplus (New\_values\ i) \\
\quad\quad \bullet ur\} \\
(RDM'\ REL?).Att = (RDM\ REL?).Att \\
\forall rnm : RNAME \mid rnm \in \mathrm{dom}\, RDM \land rnm \neq REL? \\
\bullet RDM'\ rnm = RDM\ rnm \\
\hline
\end{array}
$$

This schema definition includes the $\Delta RELDATAMODEL$ schema. The rows to be updated are specified by the input variable $Key\_values?$ which is an injective sequence of mappings from attribute names to values. Each element of this sequence represents a primary key value corresponding to a row to be updated. The variable *New_values?* is a sequence of mappings from attribute names to values indicating attribute names to be updated and their new values. The predicate part of the schema defines the constraints that should apply on the input variables and the state space of the database after the update was completed. The predicate

$$\begin{aligned}&\forall\, key : ANAME \rightarrowtail\!\!\!\!\!\!\!\;\;\; VALUE \mid key \in \operatorname{ran} Key\_values? \\ &\quad \bullet (\exists\, r : Row \mid r \in (RDM\ REL?).Rel \\ &\qquad \bullet key = (primarykey\ REL?) \lhd r.Row)\end{aligned}$$

specifies that each input key value should designate an existing row to be updated. The next predicate:

$$\begin{aligned}&\forall\, val : ANAME \rightarrowtail\!\!\!\!\!\!\!\;\;\; VALUE \mid val \in \operatorname{ran} New\_values? \\ &\quad \bullet \operatorname{dom} val \subseteq (RDM\ REL?).Att \land \\ &\qquad \operatorname{dom} val \cap primarykey\ REL? = \varnothing \land\end{aligned}$$

$$(\forall REL2 : RNAME \mid (REL?, REL2) \in \mathrm{dom}\, foreignkey$$

$$\bullet\ \mathrm{dom}\, val \cap \mathrm{dom}(foreignkey\,(REL?, REL2)) = \emptyset)$$

restricts the attributes to be updated to the attributes not in the primary key or the foreign keys of the input relation. The two input lists *Key_values?* and *New_values?* should have the same length so that each row to be updated has a corresponding set of new values for the attributes to be updated. The resulting relation is composed by the union of two sets: the set of rows which are not affected by this operation (those whose primary key values are not in the input *Key_values?* list), and the set of the updated rows. These sets are represented using the set comprehension notation.

The **primary key update with cascaded update of foreign keys** is an operation which should be done correctly, otherwise integrity in the database will be lost. An important check is that each allegedly new value for a primary key is not only from the domain specified for that key, but is also new with respect to that simple or composite key: that is, at this time the new value does not occur elsewhere in that primary column. This command not only updates a primary key value, but also updates in precisely the same way all of the matching foreign key values drawn from the same domain as the primary key.

$PK\_UPDATE\_CASCADED$

$$
\begin{array}{l}
\Delta RELDATAMODEL \setminus (primarykey, foreignkey) \\
REL? : RNAME \\
Key\_values? : \mathrm{iseq}(ANAME \mathrel{+\!\!\!\!\rightarrow} VALUE) \\
New\_values? : \mathrm{seq}(ANAME \mathrel{+\!\!\!\!\rightarrow} VALUE) \\
\hline
REL? \in \mathrm{dom}\, RDM \\
\forall key : ANAME \mathrel{+\!\!\!\!\rightarrow} VALUE \mid key \in \mathrm{ran}\, Key\_values? \\
\bullet\ (\exists r : ROW \mid r \in (RDM\ REL?).Rel \bullet key = (primarykey\ REL?) \lhd r.Row) \\
\forall val : ANAME \mathrel{+\!\!\!\!\rightarrow} VALUE \mid val \in \mathrm{ran}\, New\_values? \\
\quad \bullet\ \mathrm{dom}\, val \subseteq (RDM\ REL?).Att \\
\# Key\_values? = \# New\_values? \\
(RDM'\ REL?).Rel = \{r : ROW \mid r \in (RDM\ REL?).Rel \land \\
\qquad\quad (\forall key : ANAME \mathrel{+\!\!\!\!\rightarrow} VALUE \mid key \in \mathrm{ran}\, Key\_values? \\
\qquad\qquad \bullet\ (primarykey\ REL?) \lhd r.Row \neq key) \\
\qquad\quad \bullet\ r\} \cup \\
\qquad \{r, ur : ROW; key : ANAME \mathrel{+\!\!\!\!\rightarrow} VALUE; i : \mathbb{N} \\
\qquad \mid r \in (RDM\ REL?).Rel \land \\
\qquad\quad (primarykey\ REL?) \lhd r.Row = Key\_values?\ i \land \\
\qquad\quad ur.Row = r.Row \oplus (New\_values?\ i) \\
\qquad \bullet\ ur\} \\
\forall rnm : RNAME \mid rnm \in \mathrm{dom}\, RDM \land (rnm, REL?) \in \mathrm{dom}\, foreignkey \\
\bullet\ (RDM'\ rnm).Att = (RDM\ rnm).Att \land \\
\qquad (RDM'\ rnm).Rel = \{r : ROW \mid r \in (RDM\ rnm).Rel \land \\
\qquad\quad (\forall key : ANAME \mathrel{+\!\!\!\!\rightarrow} VALUE \mid key \in \mathrm{ran}\, Key\_values? \\
\qquad\quad \bullet\ (\exists anm : ANAME \mid anm \in \mathrm{dom}(foreignkey\ (rnm, REL?)) \\
\qquad\qquad\qquad \bullet\ r.Row\ anm \neq key\ ((foreignkey\ (rnm, REL?))(anm))) \\
\qquad\quad \bullet\ r\} \cup \\
\qquad \{r, ur : ROW; key : ANAME \mathrel{+\!\!\!\!\rightarrow} VALUE; i : \mathbb{N} \\
\qquad \mid r \in (RDM\ rnm).Rel \land key \in \mathrm{ran}\, Key\_values? \land \\
\qquad (\mathrm{dom}\, (foreignkey\ (rnm, REL?))) \lhd r.Row = Key\_values?\ i \land \\
\qquad ur.Row = r.Row \oplus (New\_values?\ i) \\
\qquad \bullet\ ur\} \\
(RDM'\ REL?).Att = (RDM\ REL?).Att \\
\forall rnm : RNAME \mid rnm \in \mathrm{dom}\, RDM \land rnm \neq REL? \\
\qquad\quad \land\ (rnm, REL?) \notin \mathrm{dom}\, foreignkey \\
\bullet\ RDM'\ rnm = RDM\ rnm
\end{array}
$$

The previous update operation can be extended to cascade updates not only for foreign keys, but also for all sibling primary keys, that is, primary keys defined on the same domain as the primary key of the input relation.

The **delete** operator permits to delete multiple rows from a relation. The schema corresponding to this operation includes the $\Delta RELDATAMODEL$ schema and has two input variables: the name of the relation involved and the set of primary keys of rows to be deleted.

DELETE

$$\begin{array}{l}
\Delta RELDATAMODEL \setminus (primarykey, foreignkey) \\
REL? : RNAME \\
Key\_values? : \mathbb{P}(ANAME \pfun VALUE) \\
\hline
REL? \in \mathrm{dom}\ RDM \\
(RDM'\ REL?).Rel = (RDM\ REL?).Rel \setminus \\
\qquad \{r : ROW; key : ANAME \pfun VALUE \\
\qquad\qquad \mid r \in (RDM\ REL?).Rel \\
\qquad\qquad \land\ key \in Key\_values? \\
\qquad\qquad \land\ (primarykey\ REL?) \lhd r.Row = key \\
\qquad \bullet\ r\} \\
(RDM'\ REL?).Att = (RDM\ REL?).Att \\
\forall rnm : RNAME \mid rnm \in \mathrm{dom}\ RDM \land rnm \neq REL? \\
\bullet\ RDM'\ rnm = RDM\ rnm
\end{array}$$

The first condition in the predicate part of the DELETE schema says the input relation name should name an existing relation. No condition is imposed on the key values in the input set *Key_values?* because the condition that the user has incorporated in the delete command might not be satisfied by any row. So, 'multiple' rows deletion includes the special cases of zero and one, and these cases do not receive special treatment. The result is the set difference between the initial set of rows and the set of rows that have been deleted. The last predicate indicates that all the other existing relations remain unchanged. The primary and foreign keys are also not affected. Execution of this command will often violate referential integrity if the deleted primary key values happen to be referenced by foreign keys in other tables. Since referential integrity is not fully checked until the end of a transaction, this violation may be just a transient state that is permitted to exist within the pertinent transaction only.

The delete operator with cascaded deletion is similar to the previous one, except it takes into account the fact that a value of an attribute of each of the rows being deleted happens to be the value of the primary key of another relation. Execution of this operation causes the DBMS to propagate deletions to those rows in the database that happen to contain dependent foreign-key values as components.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\,DELETE\_WITH\_CASCADED\_DELETION\,\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta RELDATAMODEL \setminus (primarykey, foreignkey) \\
REL? : RNAME \\
Key\_values? : \mathbb{P}(ANAME \mathrel{+\!\!\!\!\rightarrow} VALUE) \\
\hline
REL? \in \operatorname{dom} RDM \\
(RDM'\ REL?).Rel = (RDM\ REL?).Rel \setminus \\
\qquad\quad \{r : ROW; key : ANAME \mathrel{+\!\!\!\!\rightarrow} VALUE \\
\qquad\qquad \mid r \in (RDM\ REL?).Rel \\
\qquad\qquad \wedge\ key \in Key\_values? \\
\qquad\qquad \wedge\ key = (primarykey\ REL?) \lhd r.Row \\
\qquad\quad \bullet\ r\} \\
\forall rnm : RNAME; key : ANAME \mathrel{+\!\!\!\!\rightarrow} VALUE \\
\mid rnm \in \operatorname{dom} RDM \wedge key \in Key\_values? \wedge \\
\qquad (rnm, REL?) \in \operatorname{dom} foreignkey \\
\bullet\ (RDM'\ rnm).Att = (RDM\ rnm).Att \wedge \\
\qquad (RDM'\ rnm).Rel = (RDM\ rnm).Rel \setminus \\
\qquad \{r : ROW; key : ANAME \mathrel{+\!\!\!\!\rightarrow} VALUE \\
\qquad\quad \mid r \in (RDM\ rnm).Rel \wedge key \in Key\_values? \\
\qquad\quad \wedge\ (\forall anm : ANAME \mid anm \in \operatorname{dom}(foreignkey(rnm, REL?)) \\
\qquad\qquad \bullet\ r.Row\ anm = key\ ((foreignkey\ (rnm, REL?))(anm))) \\
\qquad \bullet\ r\} \\
(RDM'\ REL?).Att = (RDM\ REL?).Att \\
\forall rnm : RNAME; key : ANAME \mathrel{+\!\!\!\!\rightarrow} VALUE \\
\mid rnm \in \operatorname{dom} RDM \wedge key \in Key\_values? \wedge \\
\qquad (rnm, REL?) \notin \operatorname{dom} foreignkey \\
\bullet\ (RDM'\ rnm).Rel = (RDM\ rnm).Rel \\
\hline
\end{array}
$$

The schema definition includes the $\Delta RELDATAMODEL$ schema. The rows to be deleted are specified by the input variable $Key\_values?$ which contains all primary key values corresponding to the rows to be deleted. The resulting set of rows for the input relation is composed by the set difference between the initial set of rows and the set of rows indicated by the input variable $Key\_values?$. All relations having foreign keys with respect to the primary key of the input relation have their resulting set of rows specified as the set difference between their initial set of rows and the set of rows indicated by foreign key values from the variable $Key\_values?$. These sets are represented using the set comprehension notation.

Chapter 5

Proving Properties

Using the formal specification of the RM/V2, we can easily reason and prove the well-known and important relational properties. Some of these are obvious and emerge from the definition of domains, rows and relations:

- Since a row is a partial function from attribute names to values, all attribute-values are atomic. In other words, all relations are *normalized.* So the requirement of the relational algebra that all relations should be in (at least) *first normal form* is satisfied.

- Within a given relation, attributes are unordered. This follows from the fact that the collection of attributes of a relation is also defined as a set.

- Within the row-set of a given relation, no two rows are identical, and the rows are unordered. This follows from the fact that sets by definition do not contain duplicate elements and the order of elements is of no importance.

- As a consequence of the previous point, a relation will always have a primary key. At least the combination of *all* attributes of a relation has the uniqueness property, and hence at least one candidate key always exists.

We prove next some important properties of the join operator. The proofs make use of the definition for the join operator and properties of set theory and predicate logic.

P1. Join is commutative:

$$\forall R1, R2 : RELATION \bullet NATJOIN(R1, R2) = NATJOIN(R2, R1)$$

Proof:

To prove this property we have to prove that the output variables of the resulting schemas are equal, that is

$$NJRel(R1, R2)!.Rel = NJRel(R2, R1)!.Rel$$

and

$$NJRel\,(R1, R2)!.Att = NJRel\,(R2, R1)!.Att.$$

Let $T1$ be the resulting set of rows $NJRel(R1, R2)!.Rel$ of $NATJOIN(R1, R2)$. Then, according to the set comprehension definition from the $NATJOIN$ schema:

$T1 = NJRel(R1, R2)!.Rel =$

$\{r1, r2 : Row;\ anm : ANAME$

$\mid r1 \in (RDM\ R1?).Rel \wedge r2 \in (RDM\ R2?).Rel$

$\wedge\ (anm \in \mathrm{dom}\ Att\_Map1? \Rightarrow$

$r1.Row\ anm = r2.Row\,(Att\_Map1?\ anm))$

$\bullet\ ((\mathrm{dom}\ r1.Row \cup \mathrm{dom}\ r2.Row)\backslash$

$last\_set(\mathrm{dom}\ Att\_Map1?, \mathrm{ran}\ Att\_Map1?))$

$\lhd(r1.Row \cup r2.Row)\}$

where $Att\_Map1?$ is the mapping between the attributes of $R1$ and $R2$ which participate in the join. Similarly, let $T2$ be the resulting set of rows $NJRel(R2, R1)!.Rel$ of $NATJOIN(R2, R1)$:

$T2 = NJRel(R2, R1)!.Rel =$

$$
\begin{aligned}
&\{r2, r1 : Row;\ anm : ANAME \\
&\mid r2 \in (RDM\ R1?).Rel \land r1 \in (RDM\ R2?).Rel \\
&\land\ (anm \in \operatorname{dom} Att\_Map2? \Rightarrow \\
&\qquad r2.Row\ anm = r1.Row\,(Att\_Map2?\ anm)) \\
&\bullet\ ((\operatorname{dom} r2.Row \cup \operatorname{dom} r1.Row) \setminus \\
&\qquad last\_set(\operatorname{dom} Att\_Map2?, \operatorname{ran} Att\_Map2?)) \\
&\qquad \lhd (r2.Row \cup r1.Row)\}
\end{aligned}
$$

where $Att\_Map2?$ is the mapping between the attributes of $R2$ and $R1$ which participate in the join. It is obvious that $Att\_Map2?$ is the inverse of $Att\_Map1?$, so the expressions

$$anm \in \operatorname{dom} Att\_Map1? \Rightarrow r1.Row\ anm = r2.Row\,(Att\_Map1?\ anm)$$

and

$$anm \in \operatorname{dom} Att\_Map2? \Rightarrow r2.Row\ anm = r1.Row\,(Att\_Map2?\ anm)$$

are equivalent. For the same reason,

$$\operatorname{dom} Att\_Map1? = \operatorname{ran} Att\_Map2?$$

and

$$\operatorname{ran} Att\_Map1? = \operatorname{dom} Att\_Map2?$$

so:

$$last\_set(\operatorname{dom} Att\_Map1?, \operatorname{ran} Att\_Map1?) =$$
$$last\_set(\operatorname{dom} Att\_Map2?, \operatorname{ran} Att\_Map2?)$$

So, $T1$ and $T2$ represent the same resulting set of rows. Proving that the resulting sets of attribute names are equal is similar:

$$\begin{aligned}
A1 = {} & NJRel(R1, R2)!.Att = \\
& (RDM\ R1?).Att \cup (RDM\ R2?).Att \setminus \\
& \quad last\_set(\operatorname{dom} Att\_Map1?, \operatorname{ran} Att\_Map1?) \\
A2 = {} & NJRel(R2, R1)!.Att = \\
& (RDM\ R2?).Att \cup (RDM\ R1?).Att \setminus \\
& \quad last\_set(\operatorname{dom} Att\_Map2?, \operatorname{ran} Att\_Map2?)
\end{aligned}$$

But

$$last\_set(\operatorname{dom} Att\_Map1?, \operatorname{ran} Att\_Map1?) =$$

$$last\_set(\operatorname{dom} Att\_Map2?, \operatorname{ran} Att\_Map2?)$$

so $A1$ and $A2$ represent the same set of attribute names.

□

P2. Join is associative:

$\forall R1, R2, R3 : RELATION$

$\bullet\ NATJOIN(NATJOIN(R1, R2), R3) = NATJOIN(R1, NATJOIN(R2, R3))$

Proof:

$T1 = NJRel(NJRel(R1, R2), R3)!.Rel =$

$\{r1\_2, r3 : Row;\ anm : ANAME$

$\mid r1\_2 \in NJRel(R1, R2)!.Rel \land r3 \in (RDM\ R3?).Rel$

$\land\ (anm \in \operatorname{dom} Att\_Map12\_3? \Rightarrow$

$r1\_2.Row\ anm = r3.Row\,(Att\_Map12\_3?\ anm))$

$\bullet\ ((\operatorname{dom} r1\_2.Row \cup \operatorname{dom} r3.Row) \setminus last\_set(\operatorname{dom} Att\_Map12\_3?, \operatorname{ran} Att\_Map12\_3?))$

$\lhd (r1\_2.Row \cup r3.Row)\}$

$T2 = NJRel(R1, NJRel(R2, R3))!.Rel =$

$\{r1, r2\_3 : Row;\ anm : ANAME$

$\mid r1 \in (RDM\ R1?).Rel \wedge r2\_3 \in NJRel(R2, R3).Rel$

$(anm \in \mathrm{dom}\ Att\_Map1\_23? \Rightarrow$

$r1.Row\ anm = r3.Row\,(Att\_Map1\_23?\ anm))$

$\bullet\ ((\mathrm{dom}\ r1.Row \cup \mathrm{dom}\ r2\_3.Row) \setminus last\_set(\mathrm{dom}\ Att\_Map1\_23?, \mathrm{ran}\ Att\_Map1\_23?))$

$\lhd (r1.Row \cup r2\_3.Row)\}$

Same logic as above applies for $Att\_Map12\_3?$ and $Att\_Map1\_23?$, so:

$$anm \in \mathrm{dom}\ Att\_Map12\_3? \Rightarrow r1\_2.Row\ anm = r3.Row\,(Att\_Map12\_3?\ anm)$$

is equivalent to

$$anm \in \mathrm{dom}\ Att\_Map1\_23? \Rightarrow r1.Row\ anm = r3.Row\,(Att\_Map1\_23?\ anm)$$

and

$$last\_set(\mathrm{dom}\ Att\_Map12\_3?, \mathrm{ran}\ Att\_Map12\_3?) =$$
$$last\_set(\mathrm{dom}\ Att\_Map1\_23?, \mathrm{ran}\ Att\_Map1\_23?)$$

So, $T1$ and $T2$ represent the same sets of resulting rows.

$A1 = \mathit{NJRel}(\mathit{NJRel}(R1, R2), R3)!.\mathit{Att} =$

$\quad \mathit{NJRel}(R1, R2).\mathit{Att} \cup (\mathit{RDM}\ R3).\mathit{Att} \setminus$

$\quad\quad \mathit{last\_set}(\mathrm{dom}\ \mathit{Att\_Map}12\_3?, \mathrm{ran}\ \mathit{Att\_Map}12\_3?) =$

$\quad ((\mathit{RDM}\ R1).\mathit{Att} \cup (\mathit{RDM}\ R2).\mathit{Att} \setminus$

$\quad\quad \mathit{last\_set}(\mathrm{dom}\ \mathit{Att\_Map}12?, \mathrm{ran}\ \mathit{Att\_Map}12?)) \cup (\mathit{RDM}\ R3).\mathit{Att} \setminus$

$\quad\quad \mathit{last\_set}(\mathrm{dom}\ \mathit{Att\_Map}12\_3?, \mathrm{ran}\ \mathit{Att\_Map}12\_3?) =$

$\quad (\mathit{RDM}\ R1).\mathit{Att} \cup (\mathit{RDM}\ R2).\mathit{Att} \cup (\mathit{RDM}\ R3).\mathit{Att} \setminus$

$\quad\quad \mathit{last\_set}(\mathrm{dom}\ \mathit{Att\_Map}12?, \mathrm{ran}\ \mathit{Att\_Map}12?) \setminus$

$\quad\quad \mathit{last\_set}(\mathrm{dom}\ \mathit{Att\_Map}12\_3?, \mathrm{ran}\ \mathit{Att\_Map}12\_3?)$

and

$A2 = \mathit{NJRel}(R1, \mathit{NJRel}(R2, R3))!.\mathit{Att} =$

$\quad (\mathit{RDM}\ R1).\mathit{Att} \cup \mathit{NJRel}(R2, R3).\mathit{Att} \setminus$

$\quad\quad \mathit{last\_set}(\mathrm{dom}\ \mathit{Att\_Map}1\_23?, \mathrm{ran}\ \mathit{Att\_Map}1\_23?) =$

$\quad ((\mathit{RDM}\ R1).\mathit{Att} \cup ((\mathit{RDM}\ R2).\mathit{Att} \cup (\mathit{RDM}\ R3).\mathit{Att} \setminus$

$\quad\quad \mathit{last\_set}(\mathrm{dom}\ \mathit{Att\_Map}23?, \mathrm{ran}\ \mathit{Att\_Map}23?)) \setminus$

$\quad\quad \mathit{last\_set}(\mathrm{dom}\ \mathit{Att\_Map}1\_23?, \mathrm{ran}\ \mathit{Att\_Map}1\_23?) =$

$(RDM\ R1).Att \cup (RDM\ R2).Att \cup (RDM\ R3).Att \setminus$

$\quad last\_set(\mathrm{dom}\ Att\_Map23?, \mathrm{ran}\ Att\_Map23?) \setminus$

$\quad last\_set(\mathrm{dom}\ Att\_Map1\_23?, \mathrm{ran}\ Att\_Map1\_23?)$

where $Att\_Map12?$ is the input mapping of $NATJOIN(R1, R2)$, $Att\_Map12\_3?$ is the mapping of $NATJOIN(NATJOIN(R1, R2), R3)$, $Att\_Map23?$ represents the input mapping of $NATJOIN(R2, R3)$, and $Att\_Map1\_23?$ represents the input mapping of $NATJOIN(NATJOIN(R1, R2), R3)$. But:

$last\_set(\mathrm{dom}\ Att\_Map12?, \mathrm{ran}\ Att\_Map12?) \cup$

$\quad last\_set(\mathrm{dom}\ Att\_Map12\_3?, \mathrm{ran}\ Att\_Map12\_3?) =$

$last\_set(\mathrm{dom}\ Att\_Map23?, \mathrm{ran}\ Att\_Map23?) \cup$

$\quad last\_set(\mathrm{dom}\ Att\_Map1\_23?, \mathrm{ran}\ Att\_Map1\_23?)$

So, $A1$ and $A2$ represent the same set of attribute names.

□

P3. Join is idempotent:

$\forall R : RELATION \bullet NATJOIN(R, R) = R$

Proof:

$NJRel(R, R)!.Rel =$

$\{r1, r2 : Row;\ anm : ANAME$

$\mid r1 \in NJRel(RDM, R?)!.Rel \land r2 \in (RDM\ R?).Rel$

$\land (anm \in \operatorname{dom} Att\_Map? \Rightarrow$

$r1.Row\ anm = r2.Row\,(Att\_Map?\ anm))$

$\bullet\ ((\operatorname{dom} r1.Row \cup \operatorname{dom} r2.Row)$

$\setminus last\_set(\operatorname{dom} Att\_Map?, \operatorname{ran} Att\_Map?))$

$\lhd (r1.Row \cup r2.Row)\}$

In this case $\operatorname{dom} Att\_Map? = \operatorname{ran} Att\_Map?$, $r1$ and $r2$ represent the same row, and $last\_set(\operatorname{dom} Att\_Map?, \operatorname{ran} Att\_Map?) = \varnothing$, so:

$((\operatorname{dom} r1.Row \cup \operatorname{dom} r2.Row) \setminus last\_set(\operatorname{dom} Att\_Map?, \operatorname{ran} Att\_Map?) \lhd (r1.Row \cup r2.Row) = (\operatorname{dom} r1 \setminus \varnothing) \lhd r1.Row = r1.Row$

So,

$NJRel(R, R)!.Rel = R.Rel$

$NJRel(R, R)!.Att =$

$R.Att \cup R.Att \setminus last\_set(\mathrm{dom}\ Att\_Map?, \mathrm{ran}\ Att\_Map?) =$

$R.Att \setminus \varnothing =$

$R.Att$

$\square$

P4. Joining a relation to an empty relation of its own type yields an empty relation of the same type:

$\forall R : RELATION$

$\bullet\ NATJOIN(R, empty\_rel\ R) = empty\_rel\ R$

Proof:

$NATJOIN(R, empty\_rel\ R)!.Rel =$

$\{r1, r2 : Row;\ anm : ANAME$

$\mid r1 \in R.Rel \wedge r2 \in (empty\_rel\ R).Rel$

$\wedge\ (anm \in \mathrm{dom}\ Att\_Map? \Rightarrow$

$r1.Row\ anm = r2.Row\,(Att\_Map?\ anm))$

$\bullet\ ((\mathrm{dom}\ r1.Row \cup \mathrm{dom}\ r2.Row) \setminus$

$last\_set(\mathrm{dom}\ Att\_Map?, \mathrm{ran}\ Att\_Map?))$

$\lhd(r1.Row \cup r2.Row)\}$

The only row of an empty relation is the empty mapping, so the constraint $r1.Row\ anm = r2.Row\,(Att\_Map?\ anm)$ is not satisfied by any row in R; hence the resulting set of this join will be the empty set. Relation R and relation $empty\_rel\ R$ have the same attribute name set, so the mapping $Att\_Map? = R.Att \longrightarrow R.Att$. The resulting attribute set of the join is:

$NATJOIN(R, empty\_rel\ R)!.Att =$

$\quad R.Att \cup (empty\_rel\ R).Att =$

$\quad R.Att \cup R.Att =$

$R.Att$

□

P5. The trivial relation is an identity for the join operator:

$\forall R : RELATION$

- $NATJOIN(R, trivial\_rel\ R) = R = NATJOIN(trivial\_rel\ R, R)$

Proof:

$NATJOIN(R, trivial\_rel\ R)!.Rel =$

$$\{r1, r2 : Row;\ anm : ANAME$$

$$\mid r1 \in R.Rel \wedge r2 \in (trivial\_rel\ R).Rel$$

$$\wedge\ (anm \in \mathrm{dom}\ Att\_Map? \Rightarrow$$

$$r1.Row\ anm = r2.Row\ (Att\_Map?\ anm))$$

$$\bullet\ ((\mathrm{dom}\ r1.Row \cup \mathrm{dom}\ r2.Row) \setminus$$

$$last\_set(\mathrm{dom}\ Att\_Map?, \mathrm{ran}\ Att\_Map?))$$

$$\lhd (r1.Row \cup r2.Row)\}$$

The trivial relation has no attribute, so the mapping $Att\_Map?$ is the empty mapping. Then the expression:

$$anm \in \mathrm{dom}\ Att\_Map? \Rightarrow r1.Row\ anm = r2.Row\,(Att\_Map?\ anm)$$

is always true. The only row of the trivial relation is the empty mapping, so:

$(\mathrm{dom}\ r1.Row \cup \mathrm{dom}\ r2.Row) \setminus last\_set(\mathrm{dom}\ Att\_Map?, \mathrm{ran}\ Att\_Map?)) \lhd (r1.Row \cup r2.Row) =$

$(\mathrm{dom}\, r1.Row \cup \emptyset) \setminus \emptyset) \lhd r1.Row = \mathrm{dom}\, r1.Row \lhd r1.Row = r1.Row$

So, the the resulting set of rows of the operation $NATJOIN(R, trivial\_rel\, R)$ is the set of rows of the relation R. Using the same logic, we can prove that the output set of rows of the operation $NATJOIN(trivial\_rel\, R, R)$ is the set of rows of R.

The set of attibute names of the trivial relation is the empty set, so:

$NATJOIN(R, trivial\_rel\, R)!.Att =$

$\quad R.Att \cup (trivial\_rel\, R).Att =$

$\quad R.Att \cup \emptyset =$

$R.Att$

Same logic applies for $NATJOIN(trivial\_rel\, R, R)!.Att = R.Att$. So, the trivial relation is an identity for the join operator.

□

P6. Natural join is equivalent to cartesian product followed by selection and projection

Proof:

To prove this property, we have to prove that the resulting set of rows of the natural join and the resulting set of rows of the projection which follows the cartesian product and the selection are the same, and that their resulting attribute name sets are the same. The first property is equivalent to the following two properties:

(1) $\forall r : ROW \mid r \in NJRel!.Rel \bullet r \in CPRel!.Rel \land$

(2) $\forall r : ROW \mid r \in CPRel!.Rel \bullet r \in NJRel!.Rel$

Proof of (1):

$\forall r : ROW \mid r \in NJRel!.Rel \bullet$

$\quad (\exists r1, r2 : ROW \mid r1 \in (RDM\ R1).Rel \land r2 \in (RDM\ R2).Rel$

$\quad\quad \bullet (\forall anm : ANAME \mid anm \in \operatorname{dom} Att\_Map?$

$\quad\quad\quad \bullet r1.Row\ anm = r2.Row\,(Att\_Map?\,anm)))$

So, the row $r1\_2$ represented by the concatenation of the two rows, $r1.Row \cup r2.Row$, is in the set of rows of the cartesian product of relations $R1$ and $R2$. This set of rows constitutes the input for the selection operator. The other input for selection, the function *PRED*?, has the following definition in this case:

$$
\begin{array}{|l}
PRED? : ROW \rightarrow\!\!\!\!\!\rightarrow BOOL \\
\hline
\forall r : ROW \mid r \in CPRel!.Rel \\
\quad \bullet\ PRED?\ r = True \Leftrightarrow \\
\quad\quad (\forall anm : ANAME \mid anm \in \mathrm{dom}\, Att\_Map? \\
\quad\quad\quad \bullet\ r.Row\ anm = r.Row\,(Att\_Map?\ anm))
\end{array}
$$

The row represented by $r1.Row \cup r2.Row$ satisfies this property, so it will be in the resulting set of rows of the selection operation, which is the input for the projection operator:

$$r1\_2 \in SRel!.Rel$$

The row r from the resulting set of rows of the natural join is of the form:

$$r.Row = ((\mathrm{dom}\, r1.Row \cup \mathrm{dom}\, r2.Row) \setminus last\_set(\mathrm{dom}\, Att\_Map?, \mathrm{ran}\, Att\_Map?)) \lhd (r1.Row \cup r2.Row)$$

The input sequence *AL*? of the projection operation is in this case:

$\operatorname{ran} AL? = (R1.Att \cup R2.Att) \setminus last\_set(\operatorname{dom} Att\_Map?, \operatorname{ran} Att\_Map?))$

$= (\operatorname{dom} r1.Row \cup \operatorname{dom} r2.Row) \setminus last\_set(\operatorname{dom} Att\_Map?, \operatorname{ran} Att\_Map?))$

which is exactly the domain of $r.Row$. So, projecting the row represented by $r1.Row \cup r2.Row$ on the attribute names indicated by the $AL?$ list will result in the original row r of the natural join of relations $R1$ and $R2$. The proof of property (2) is similar.

The resulting set of attribute names of the natural join and the cartesian product are equal:

$NJRel!.Att = (RDM\ R1?).Att \cup (RDM\ R2?).Att \setminus$

$last\_set(\operatorname{dom} Att\_Map?, \operatorname{ran} Att\_Map?) =$

$\operatorname{ran} AL? =$

$CPRel!.Att$

□

P7. Updating a relation is equivalent to deleting the affected rows followed by inserting new rows containing the new values

Proof:

Proving this property is equivalent to proving that every row in the resulting set of rows of the update operation is in the resulting set of rows of the insert operation which follows the deletion, and every row from this resulting set of rows is in the result of the update operation.

The input for the update operation is represented by the relation $R?$, the sequence *Key_values?* representing the primary key values of the rows to be updated, and the sequence *New_values?* representing the attributes to be updated and their new values. The same relation name $R?$ and sequence of primary key values *Key_values?* are the input for the delete operator, in this case *Key_values?* representing the rows to be deleted. The input for the insert operator in this case is the relation $R?$ and the set of rows *Rows_to_insert?* representing the rows to be inserted. The *Rows_to_insert?* variable has the following definition:

$$
\begin{array}{|l}
\forall r : ROW \mid r \in Rows\_to\_insert? \\
\bullet\ (\exists i : \mathbb{N} \mid i \in 1 \,..\, \# Key\_values? \\
\quad \bullet\ (primarykeyR?) \lhd r.Row = Key\_values?\ i) \land \\
\quad (\mathbf{let}\ f : ANAME \mathrel{+\!\!\!\rightarrow} VALUE \mid f = \mathrm{ran}\ New\_values? \\
\quad \bullet\ (\forall\, anm : ANAME \bullet \\
\qquad (anm \in \mathrm{dom}\ f \Rightarrow r.Row\ anm = f\ anm) \land \\
\qquad (anm \notin \mathrm{dom}\ f \Rightarrow r.Row\ anm = old\_value)))
\end{array}
$$

where *old_value* represents the original value corresponding to attibutes of row r which are not updated. So, any row whose primary key value is not in the *Key_values?* list is not updated, so it will remain the same in the resulting set of rows of the update operation. The same row is not deleted by the delete operation, and it is not changed by the insert operation. The rows which are updated by the update operation are the rows in the set:

$$
\begin{array}{|l}
\{ r, ur : ROW;\ key : ANAME \mathrel{+\!\!\!\rightarrow} VALUE;\ i : \mathbb{N} \\
r \in (RDM\ REL?).Rel \land \\
\quad (primarykey\ REL?) \lhd r.Row = Key\_values\ i \land \\
\quad ur.Row = r.Row \oplus (New\_values\ i) \\
\bullet\ ur \}
\end{array}
$$

This definition corresponds to the definition for *Rows_to_insert?*, so the set of updated rows will be the same as the set of inserted rows. □

Chapter 6

Conclusions and Future Work

This thesis presents a formal specification in Z of the relational data model, version 2, of E.F.Codd. The contributions of this work include:

- the specification fully formalizes the basic concepts of an industrially significant data model, the relational data model, including the retrieval and manipulative operators;
- new features introduced by RM/V2 are incorporated into the relational model;
- the formal specification provides a convenient source of reference and understanding on the relational data model.

The specification uses Z as the formal specification language because of its simplicity, modularity capabilities and because of the link that exists between Z and the relational algebra. The specification was type-checked by the FUZZ type checker.

The specification starts with the definition of domains, rows and relations. The state space of the system is defined by the *RELDATAMODEL* schema which makes use of other schemas, global variables and axiomatic definitions previously defined. Formal definitions are given for the empty, trivial and universal relations, which are used later on when proving important relational properties. The definition of keys includes both the uniqueness and minimality properties. The specification of missing values defines the case of two possible null values, *missing-but-applicable* and *missing-and-inapplicable*. These are considered when defining the two integrity rules, entity integrity and referential integrity. The semantic features of the relational data model are fully supported by this specification. These are:

- domains, primary keys, and foreign keys;

- duplicate values are permitted within columns of a relation, but duplicate rows are prohibited;
- systematic handling of missing information independent of the type of datum that is missing.

Throughout the specification we consider the case of composite keys and make use of the naming technique as it is defined in RM/V2. The retrieval and manipulative operations completes the definition of the relational data model. Using this formal specification we make proofs for some well-known relational properties.

The use of formal reasoning to prove properties of the system under consideration is a very important feature of formal specifications. Using formal reasoning, we can not only demonstrate at an early stage that the system enjoys certain properties, but we can also check the parts of the specification involved in proofs. During formal reasoning about the relational data model, we were able to detect inconsistent definitions in the specification, and to go back and complete this with new definitions needed by proofs.

The use of *Z* as the formal specification language allows us to define the state space of the model using separate definitions for different concepts involved, then to combine them in the *RELDATAMODEL* schema which represents the global state of the system. The constraints represented by the two data integrity rules are also defined separatelly, using different schemas. The schema calculus allows the specification to be partitioned and presented in smaller pieces. This modularity capability of *Z* is a major advantage over *VDM*, which has no such feature. The formal setting of *Z* also allows questions on properties of the model to be answered confidently using formal reasoning. The set theory forms an adequate basis for building complex data structures which are needed in specifications. The *FUZZ* package allowed us to check the *Z* specification with the *Z* scope and type rules.

The *Z* specification of the RM/V2 is not complete in the sense that not all the features introduced by the RM/V2 are included. We do not provide a formal definition of the usual logical connectives from the set theory with respect to the four-valued logic of RM/V2. The specification does not include the definition of views and view updatability. This work can also be extented to include concepts like user-defined integrity constraints and advanced operators like outer-join or recursive join. Having the specification of the relational data model as the starting point, we can go further and specify deductive databases. All these additional features are part of future work.

Bibliography

[Bar91] Barros R. and Harper D.J., "Formal Development of Relational Database Applications". In: Harper D.J. and Norrie M.C.: *Specifications of Database Systems, Workshops in Computing Series*, pp. 21–43, Springer-Verlag, 1991.

[Bar93] Barros R. "Formal Specification of Relational Database Applications: A Method and an Example". Research Report. University of Glasgow, Department of Computing Science. 1993.

[Bjö82] Björner D. and Lövengreen H.H. "Formalization of Data Models". In: Björner D., Jones C. et al.: *Formal Specification and Software Development*, pp.379–442. Prentice Hall International. 1982.

[Bjö90] Björner D., Hoare C.A.R. and Langmaak H. (eds). *VDM '90: VDM and Z - Formal Methods in Software Development*, vol. 428 of *Lecture Notes in*

Computer Science, Springer-Verlag, 1990.

[But94] Butler G. "Technical Trends in Industrial Software Engineering: Quality, Reuse, Modeling". Submitted Paper. Concordia University, Department of Computer Science. 1994

[Cha92] Chan D., Harper D. and Trinder P. "A Reference Object Oriented Data Model Specification". Technical Report. University of Glasgow, Department of Computing Science. 1992.

[Cod70] Codd E.F. "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM*, 13(6):377–387, June 1970.

[Cod79] Codd E.F. "Extending the Database Relational Model to Capture More Meaning". *ACM Transactions on Database Systems* 4:4, 1979.

[Cod90] Codd E.F. *The Relational Model for Database Management: Version 2.* Addison-Wesley. 1990.

[Dav91] Davis C. K. "A Denotational Approach to Object-Oriented Query Language Definition". In: Harper D.J. and Norrie M.C.: *Specifications of Database*

Systems, Workshops in Computing Series, pp. 88–104, Springer-Verlag, 1991.

[Dat83] Date C.J., *An Introduction to Database Systems: Volume 2.* Addison-Wesley. 1983.

[Dat86] Date C.J., *Relational Databases: Selected Writings.* Addison-Wesley. 1986.

[Die90] Diepen M.J. and Hee K.M. "A formal semantics for Z and the link between Z and relational algebra". In: Björner D., Hoare C.A.R and Langmaak H. (eds), *VDM '90: VDM and Z - Formal Methods in Software Development*, vol. 428, pp. 526–552 of *Lecture Notes in Computer Science*, Springer-Verlag, 1990.

[Fit90] Fitzgerald J.S. and Jones C.B. "Modularizing the Formal Description of a Database System". In: Björner D., Hoare C.A.R. and Langmaak H. (eds), *VDM '90: VDM and Z - Formal Methods in Software Development*, vol. 428, pp.189-210 of *Lecture Notes in Computer Science*, Springer-Verlag, 1990.

[Gog75] Goguen J.A. et al. "Abstract Data Types as Initial Algebras and Correctness of Data Representations". *Proceedings of Conference on Computer Graphics, Pattern Recognition, and Data Structures*, ACM, May 1975, pp. 89–93.

[Gu77] Guttag J. "Abstract Data Types and the Development of Data Structures". *Communications ACM*, 20 (6), pp.396–405.

[Hal90] Hall A. "Seven Myths of Formal Methods". *IEEE Software* 7, 5 , September 1990, pp.11–19.

[Har92] Harper D.J. and Norrie M.C. (eds). *Specifications of Database Systems.* Springer-Verlag. 1992.

[Hay92] Hayes I. "VDM and Z: A Comparative Case Study". *Formal Aspects of Computing.* (1992) 4: 76–99.

[Haye92] Hayes I. *Specification Case Studies.* Prentice Hall International. 1992.

[Hoa85] Hoare C.A.R. *Communicating Sequential Processes*, Prentice Hall International, 1985.

[Jon86] Jones C.B. *Systematic Software Development Using VDM.* Prentice Hall International, 1986.

[Jon90] Jones C.B. and Shaw R.C. (eds). *Case Studies in Systematic Software Development.* Prentice Hall International. 1990.

[Jos91] Joseph M.B. and Redmond-Pyle D. "Entity-Relationship Models Expressed in Z: A Synthesis of Structured and Formal Methods". Technical Report. Oxford University Computing Laboratory. 1991.

[Jose91] Joseph M.B. and Redmond-Pyle D. "A Library of Z Schemas for use in Entity-Relationship Modelling". Technical Report. Oxford University Computing Laboratory. 1991.

[Kan90] Kanellakis P.C. "Elements of Relational Database Theory". *Handbook of Theoretical Computer Science*, vol. B, pp. 1175–1156, Elsevier Science Publishers, 1990.

[McGo94] David McGoveran. "The Relational Model Turns 25... and We're Still Trying to Get it Right". *DBMS*, pp. 46-60, October 1994.

[Mišo92] Mišić V., Velašević D. and Lazarević B. "Formal Specification of a Data Dictionary for an Extended ER Data Model". *The Computer Journal*, vol. 35, no. 6, pp. 611–622, 1992.

[Nor92] Moira C. Norrie. "A Specification of an Object-Oriented Data Model with Relations". *Specifications of Database Systems*, pp. 213–227, Springer-Verlag, 1992.

[Pot91] Potter B., Sinclair J. and Till D. *An Introduction to Formal Specification and Z.* Prentice Hall International. 1991.

[Saa91] Saake G. and Jungclaus R. "Specification of Database Applications in the TROLL Language". In: Harper D.J. and Norrie M.C.: *Specifications of Database Systems*, *Workshops in Computing Series*, pp. 228–254, Springer-Verlag, 1991.

[Sha90] Roger C. Shaw. "The ISTAR Database". In: Jones C.B. and Shaw R.C. (eds). *Case Studies in Systematic Software Development*, pp. 47–90. Prentice Hall International. 1990.

[Som89] Sommerville I. *Software Engineering.* Addison-Wesley. 1989.

[Sp88] Spivey J.M. "An Introduction to Z and Formal Specifications". Technical Report. Oxford University Computing Laboratory. 1988.

[Spi88] Spivey J.M. *The fuzz manual.* Computing Science Consultancy, 2 Willow Close, Garsington, Oxford. 1988.

[Sp92] Spivey J.M. *The Z Notation: A Reference Manual*, second edition. Prentice Hall International. 1992.

[Su85] Sufrin B. and Hughes J. "A Tutorial Introduction to Relational Algebra". *Programming Research Group, Oxford, Draft for comment*

[Tha90] Thayer R. and Dorfman M. *System and Software Requirements Engineering. IEEE Computer Society Press Tutorial*, 1990.

[Tsa95] Tsai J.P., Yang S. J. and Chang Y.H. "Timing Constraint Petri Nets and their Application to Schedulability Analysis of Real-Time System Specifications". *IEEE Transactions on Software Engineering*, vol. 21, no. 1, January 1995, pp. 32–49.

[Tse91] Tse T.H. and Pong L. "An Examination of Requirements Specification Languages". *The Computer Journal*, vol. 34, no.2, pp. 143–152, 1991.

[Tur85] Turner R. and Lowden B.G.T. "An Introduction to the Formal Specification of Relational Query Languages". *The Computer Journal*, vol. 28, no. 2, pp. 162–169, 1985.

[Wal90] Walshe A. "NDB: The Formal Specification and Rigorous Design of a Single User Database System". In: Jones C.B. and Shaw R.C. (eds). *Case Studies in Systematic Software Development*, pp. 11–45. Prentice Hall International. 1990.

[Wi87] Wing J.M. "Writing Larch Interface Language Specifications". *ACM Transactions Programming Languages and Systems*, Jan. 1987, pp. 27–44.

[Wi90] Wing J.M. "A Specifier's Introduction to Formal Methods". *IEEE Transactions on Software Engineering*, Sept. 1990, pp. 8–22.

[Wo92] Wordsworth J.B. *Software Development with Z*. Addison-Wesley, 1992.

[Wor91] Worboys M.F. "Database Specification using Transaction Sets". In: Harper D.J. and Norrie M.C.: *Specifications of Database Systems*, *Workshops in Computing Series*, pp. 300-319, Springer-Verlag, 1991.