# Deriving Relational Database Programs from Formal Specifications

Roberto S. M. de Barros*

Department of Computing Science
University of Glasgow
Glasgow G12 8QQ, Scotland, UK.

**Abstract.** The derivation of database programs directly from formal specifications is a well known and unsolved problem. Most of the previous work on the area either tried to solve the problem too generally or was restricted to some trivial aspects, for example deriving the database structure and/or simple operations. However difficult in general, deriving relational database applications directly from Z specifications satisfying a certain set of rules (the method) is not arduous. This paper describes a set of rules on how to map such specifications to database programs. If appropriate tool support is provided, writing formal specifications according to the method and deriving the corresponding relational database programs should be straightforward. Moreover it should produce code which is standardized and thus easier to understand and maintain.

## 1 Introduction

Although some work has already been published, the utilization of formal or semi-formal techniques in the development of real life database [1] applications has not been seriously attempted yet.

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

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

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

*The author is a Brazilian Civil Servant, sponsored by CAPES, on leave from Federal University of Pernambuco (UFPE). E-mail: roberto@dcs.gla.ac.uk; roberto@di.ufpe.br from December/94.

The approach presented here is restricted to the specification and reification of relational [1, 2] database applications. Furthermore, it also considers all relevant kinds of constraints as well as more complicated transactions.

Most of the published work on the area use algebraic (property-oriented) specification languages. Even so, it seems that model-oriented specification languages are more appropriate to specify database transactions, especially because of the convenient notion of *state*. The specification language used in this work is Z [3, 4].

An important part of this work is the provision of a general method [5, 6, 7] for the specification of relational database applications using Z. The method allows for an abstract formal specification of the applications, focusing on the important aspects of the relational model and applications, without regard to the fact that some features may not be supported by specific Relational Database Management Systems (RDBMS) and query languages. It provides the formal basis in terms of which applications can be specified, verified (using formal reasoning) and implemented (reified).

In addition, the method allows for the standardization of the specifications. Hence, it is possible to design a tool to enforce the method and thus help the users writing correct specifications. Moreover, application programs may be (semi-)automatically generated from these specifications, at least for a large number of applications.

The main objectives of this paper are (1) to present a partial description of the mapping process [7, 8] for deriving relational database programs from formal specifications written in Z according to the method and (2) to describe the prototype, which is being built, that supports the method and partially implements the aforementioned mapping. It generates relational database applications to be run in DBPL [9, 10], a RDBMS built in the University of Hamburg.

Although no binding of the mapping process to specific RDBMSs and query/host languages (or 4GLs) is intended, the DBPL language is extensively used together with parts of the company database example [11] to illustrate the process. Whenever appropriate, SQL [12] and DBASE-IV [13] are also mentioned in the discussion.

The organization of this paper is as follows: In Section 2 a concise summary of the method is presented. Section 3 describes the mapping in detail, though not completely. Specific details on the functionality and implementation of the prototype are briefly discussed in Section 4. Finally, some conclusions are presented in Section 5.

# 2 The Method

This section presents a very brief summary of the method, which prescribes how to specify all the important aspects of relational database applications. This includes the definition of domains and relations, the specification of constraints, and querying and updating of relations, including error handling. The method also covers more advanced features such as transactions, sorting of results, aggregate functions, composite attributes, and views. Some features of the relational model itself are specified as pre-defined operators (generic definitions) which simplify the specifications.

Domains are sets of values from which one or more attributes draw their values. The aim is to prevent comparisons of attributes which are not based on the same domain by strongly type-checking domains based on their names. They are specified by abbreviation definitions based on other domains, possibly as a range or adding extra constraints; by free type definitions (enumerations); or as given sets. For example:

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

$$AGE == 0..18$$

$$DNUM == \{ n : \mathbb{N} \mid n > 100 \}$$

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

$$[NAME]$$

Relations are specified as sets of tuples, which respects the original relational model defined by Codd [2]. In addition, the method per se does not enforce any constraints on the way relations and operations may be implemented.

The formal definition of a relation is split into two parts: the relation intention and the relation extension. The intention is a Z tuple type (record) which defines its attributes ("variables" of the tuple type), each of which must be of a valid domain.

Basically, a tuple type is a schema without its predicate part. According to this extension [14], "only types can be used to define the domain of a variable in a schema definition", schemas cannot be used for this purpose.

$$EMPL \mathrel{\widehat{=}} [ENum : ENUM;\ Sex : SEX;\ Age : AGE;\ DNum : DNUM;\ \ldots]$$

The relation extension is a variable of type SET ($\mathbb{P}$) of the type defined earlier, which is declared in a schema. The predicate of such a schema specifies all static constraints that only depend on the relation being defined. This includes the required (not null) and candidate key constraints, specified using the operators *REQUIRED* and *KEY_OF* respectively, as well as other static attribute constraints. For example:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Employee}} \\ \hline
empls : \mathbb{P}\, EMPL \\ \hline
REQUIRED\ empls\ ENum\ \wedge \\
REQUIRED\ empls\ Sex\ \wedge \\
\ldots \\
KEY_OF\ empls\ ENum\ \wedge \\
\forall e : empls \bullet e.Age > 25 \\ \hline
\end{array}$$

The state schema, e.g. *DB*, which will represent the Database as a whole, groups all database definitions by including the relation extension schemas (example below). Its predicate specifies all static constraints depending on more than one relation, which include the foreign key constraints specified using the *FOR_KEY* operator.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{DB}} \\ \hline
Employee \\
Depart \\
\ldots \\ \hline
FOR_KEY\ depts\ ManENum\ \ empls\ ENum\ \wedge \\
\ldots \\
\forall e : empls \bullet (\exists w : works \bullet w.ENum = e.ENum)\ \wedge \\
\forall d : depts \bullet d.NEmp = \#\{ e : empls \mid e.DNum = d.DNum \} \\ \hline
\end{array}$$

The foreign key specification above means that, for all tuples of relation *depts*, attribute *ManENum* must either be null or match the primary key attribute *ENum* of some tuple of relation *empls*.

As in standard Z, other state schemas called $\Delta DB$ and $\Xi DB$ as well as an initialization schema called Init_DB are defined. The details are omitted.

Read-only operations, i.e. *select*, *join*, and *project*, are specified by schemas such that: (1) they include the $\Xi DB$ schema; (2) they declare the input (if any) and output variables of the operations; (3) their output variables are usually relations, i.e., their types are $\mathbb{P}\, A$, where $A$ is the intention (type of the tuples) of some relation; and (4) their predicates describe the result of the operations using a set comprehension. Specific constraints involving the input variable(s) of the schema may also be specified. An example combining select, project, and join operations is given below.

*Empls_salary_hours*

$$
\begin{array}{l}
\Xi DB \\[4pt]
p? : PNUM; \\
sempl\_work! : \mathbb{P}\, EMPL\_WORK \\
\hline
\exists\, pj : projs \;\bullet\; pj.PNum = p? \;\wedge \\[4pt]
sempl\_work! \;=\; \{\; e : empls;\; w : works \;\mid \\
\qquad\qquad\qquad\qquad w.PNum = p? \;\wedge\; e.ENum = w.ENum \\
\qquad\qquad\qquad\qquad\qquad \bullet\; (e.ENum, e.Salary, w.Hours)\; \}
\end{array}
$$

Update operations are specified by schemas that (1) include the $\Delta DB$ schema; (2) declare the input (if any) variables of the operations – normally there are no output variables; (3) specify what relations are changed by the operations, using a schema expression based on the $\Xi DB$ schema; and (4) describe, in their predicates, the updates in one or more relations of the database.

Examples of predicates of update operations include $empls' = empls \cup sempl?$, for inserts; $empls' = DELETE\ sempl\ ENum\ se?$, for deletes based on the primary keys; and $works' = \{w : works \bullet \textbf{if}\ w \in swork?\ \textbf{then}\ w \setminus (PNum = p?)\ \textbf{else}\ w\}$, for updates of attributes based on a select condition; where *DELETE* is another operator.

In the specific cases of deletes based on the primary key and updates of the primary key attributes the method also covers the specification of the foreign key compensating actions, (*Restricted*, *Cascades*, and *Nullifies*), which are used to prevent violations of the foreign key constraints. Again, the details are omitted.

More complicated transactions are specified using the schema piping (>>) of basic operations written according to other rules of the method. Notice that the version of the piping operator (>>) used here is not part of standard Z. It allows for the output *and* primed state variables (*all results*) of the first schema to be matched against the input *and* unprimed state variables of the second schema, respectively.

In addition, renaming variables of the component schemas is usually necessary to make variables of different operations be the same variable, avoid name clashes, and/or keep the ? and ! naming conventions for input and output variables valid in the transaction. Extra parentheses are sometimes needed to enforce an order in the association of the schemas. Occasionally, additional predicates are needed to specify constraints depending on the inputs of more than one subtransaction and/or to make the value of a variable refer to the value of an attribute of a tuple variable.

A generic transaction definition is presented below, where $Transac1\_Ok$ is the correct behaviour of the transaction, $Oper\_1, ..., Oper\_n$ are the components of the transaction, and <condition> is an additional predicate as mentioned above.

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

The method also covers other advanced features such as sorting of results, aggregate functions, composite attributes, and views; the specification and simplification of the precondition of the transactions; the specification of one or more error schemas, which associate specific error messages to each possible violation of the precondition; and the specification of the total transactions; but, once again, the details are omitted. Full details of the method are given in [6].

# 3 The Mapping

Now, a partial description of the mapping process for the derivation of relational database programs directly from specifications written according to the method is presented. It is split into four subsections: the first describes the mapping of the database structures; the second describes the mapping of basic operations; the third covers more advanced features, which includes the derivation of transactions; and the fourth deals with the extension of the applications to capture error handling.

Although the description given here is incomplete, its topics are named using labels of the type $Xn$, where $X$ can be D, standing for database rules, B for basic operation rules, A for advanced feature rules, or E for extended application rules to capture error handling; and $n$ will be a sequential number within each kind of rule, with subitems when necessary, similarly to the rules in the complete description of the method.

Even though the efficiency of the generated code is also taken into account, it is not a primary concern. On the contrary, the efficiency of the generated code is sometimes disregarded in order to make the mapping from the specification as smooth as possible. However, this does not mean the generated programs are terribly slow because a number of these operations are optimized by the compiler.

Finally, although an effort is made to keep the generated programs as close to the specifications as possible so that the mapping is simple, it is not always possible to achieve this simplicity. In some cases, in addition to the relevant data from the corresponding section of the specification, the implementation includes data from other parts of the specification method. It is also sometimes necessary to incorporate design decisions into the mapping so that the generated programs are syntactically correct.

## 3.1 Mapping the Database Structures and Constraints

In this subsection, a general discussion on the mapping of the structure part of database specifications written according to the method is presented. This includes the specification of domains, relations and their attributes and the constraints to be guaranteed.

In the specific case of RDBMSs which do not support constraints directly, the necessary constraints will have to be enforced explicitly in the implementation of the transactions that can possibly violate the integrity of the database (it is assumed the database is in a valid state before the operations).

The most direct way of generating the necessary conditional expressions in the application programs is from the preconditions of the transactions, more specifically, from the error schema(s) associated with the transaction. In fact, even when the constraints are supported, it is usually necessary to test the DBMS return codes so that specific error messages can be reported to the user. Therefore, part of the discussion about mapping the constraints is postponed to Subsection 3.4.

### D1 - Domains

If the DBMS/query language does not support domains (or user type definitions), then all domains should be enforced by means of explicit constraints. Note that, in this case, avoiding operations between attributes and/or variables of different domains which are implemented by the same basic data type is not going to be possible.

All domains in the specification (attributes and variables as well) will, ultimately, be implemented by one of the basic data types offered by the DBMS and/or query language. Sometimes, these data types include a parameter giving the size they will occupy and, so, the mapping will have to incorporate some specific value as default.

If necessary, explicit constraints are to be enforced on the values that attributes and variables drawn from the domains can take. This depends on which kind of domain definition is used in the specification, which is briefly discussed below.

In theory, there should be no domain defined as a given set in the final version of the specification. If the correct implementation for a specific domain is not known at this stage, the basic type STRING should be used.

### D1.1 - Domains Defined as Basic Types

These refer to the simplest domain definitions possible, e.g. $ENUM == \mathbb{N}$. If domains or type definitions are supported, the translation of syntax is trivial. For example, in DBPL it would be implemented by the type definition `ENUM = CARDINAL;`.

If these are not supported, then all attributes and variables drawn from domain $ENUM$ are to be mapped as if they were of type $\mathbb{N}$ in the specification, and no other constraint is needed. In this case all attributes and variables specified, for example $Att : ENUM$, would have the type natural numbers (`CARDINAL`) in the implementation.

Some DBMSs and/or query languages ask for the size occupied by the attribute and, thus, a default value is needed. For example, in DBASE IV such attributes must be written as `Att NUMERIC n,` where `n` is the number of bytes used to represent the attribute.

### D1.2 - Domains Defined as a Subset of a Basic Type

An example of this kind of domain is given by the range definition $AGE == 0..18$. If this kind of syntax is supported, a straightforward translation is done. For example, it would be implemented as the type `AGE = 0..18;` in DBPL.

If this kind of definition is not supported then the domain should be implemented as if the specification were $AGE == \{n : \mathbb{N} \mid n \leq 18\}$, which is discussed below.

This more general form of subsets, e.g. $DNUM == \{n : \mathbb{N} \mid n > 100\}$, is unlikely to be supported by any current DBMS and, hence, it should be implemented as if the specification were $DNUM == \mathbb{N}$ (case D1.1) and a specific constraint (i.e. $var > 100$) had been written for all variables $var$ drawn from domain $DNUM$.

### D1.3 - Domains Defined as an Enumeration of Values

An example of this kind of domain is given by $SEX ::= Male \mid Female \mid NULLSEX$. If enumeration types are supported, the translation is again trivial. For example, the corresponding DBPL implementation is the type `SEX = (Male, Female, NULLSEX);`.

If this specific kind of domains/types are not supported, it should be simulated by a convention using the natural numbers 0..N-1, where N is the number of different values of the domain representing its values, i.e., it is reduced to case D1.2.

### D2 - Relations

It is assumed that all RDBMSs provide a way of introducing relations. In some systems, the intention of relations (D2) and their corresponding extensions (D3) are created by separate definitions, just like in the specification method. Other systems provide a single definition. In both cases mapping the relations should involve a simple translation, and the types of the attributes would be either the domains implemented (if supported) or the basic types chosen to implement them, as described in rule D1. An example of relation specification equations is presented below:

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

$$projs : \mathbb{P}\, PROJ$$

In DBPL, the equations above can be defined in a single step or in two separate steps. The one using two separate equations is preferred because (1) it is closer to the specifications and (2) it provides a name for the record type (the tuple) which will make it easier to map other parts of the specification.

The intention of the relation above would then be written as:

```
TYPE  Proj  =  RECORD  PNum: PNUM;  DNum: DNUM;  END;
```

In DBASE IV, the relation intention and extension are defined in a single step. In this case, the above specifications would be implemented as a file called `Proj.dbf`, which would contain the information about the structure of the relation (attribute definitions) and would also store the actual data.

### D3 - Relations (Extension)

In the case of DBMSs which support the definition of the relations in two steps, mapping the specification of the extension of relations (e.g. $projs : \mathbb{P}\, PROJ$) should also be simple. In DBPL the relation extension would be implemented as follows:

```
TYPE  REL_PROJ  =  RELATION  OF  PROJ;
```

### D3.1 - Required Attributes

If the DBMS supports nulls, this should again involve a simple translation to the appropriate syntax. In SQL systems, for example, it is usually done by writing the keywords `NOT NULL` after each attribute definition.

If nulls are not supported, they should be simulated in a style similar to the *default values* approach [15] by enforcing an extra constraint on the attributes that cannot be null saying that each of them cannot assume the chosen null value for its domain.

Neither DBPL nor DBASE IV support null constraints and so they have to be simulated. Basically, for each basic type provided by the DBMS, a null constant must be chosen, though appropriate default values for the usual basic types are already provided by the method. The conditional expressions needed to enforce the constraint would then be generated from the error schema associated with the transactions and are discussed later (Subsection 3.4).

### D3.2 - Candidate Keys

For the sake of the implementation, it is assumed that the first use of the *KEY_OF* operator in each relation schema refers to the primary key whereas the others (if any) are secondary keys.

In theory, all DBMSs should provide a way of saying which attribute(s) form the primary keys and so a simple translation should be enough. Some systems do not enforce the primary key constraints though.

In DBPL, the full definition of the relation extensions include the primary key attributes. So, the full definition of Proj is given by:

```
TYPE  REL_PROJ  =  RELATION  PNum  OF  PROJ;
```

However, in order to be able to inform the user of any violations of the primary key constraints detected by the DBMS, it is necessary to test the relevant DBMS return code (`RESTRICTED()`, in DBPL) after *all* insert operations.[1]

So, as far as real database applications are concerned, the fact that primary key constraints (and more generally any kind of constraints) are supported does not necessarily mean the mapping to an implementation is going to be simpler.

Secondary keys should be defined similarly when supported.

If any of these is not supported, the uniqueness constraint should be enforced explicitly. In general, unique indexes are supported and should be used in these cases. In some DBMSs, e.g. DB2 [16], the use of indexes is compulsory. Otherwise, key constraints should be enforced from the error schemas associated with the transactions, similarly to the null constraints.

### D3.3 - Static Attribute Constraints

If the DBMS supports the specification of such constraints, the appropriate syntax should be used. Otherwise, they should be enforced from the error schemas associated with the transactions, as any other constraints.

Even though constraints of the form $\forall\, t : rel \bullet$ <condition> can be expressed in DBPL as `ALL t IN rel ( <condition> )`, where `<condition>` must be of type `BOOLEAN` and might use `OR`, `AND`, `NOT`, `ALL` ($\forall$), and `SOME` ($\exists$), the DBPL implementation does not provide a way of enforcing them. Thus, the appropriate tests that guarantee the consistency of the database should still be generated from the error schemas.

### D4 - The Database Schema

Usually, there is nothing extra to be done in this case. In DBPL the relation extensions must be declared inside a database module. The details are omitted.

[1] Updates of the primary key attribute(s), which could also lead to the violation of primary key constraints, are not allowed in DBPL.

**D4.1 - Foreign Key Constraints**

Strictly speaking, these are a special case of static attribute constraints. Thus, if not supported they should be treated as any other constraints otherwise the translation should be simple. Most DBMSs currently available, including DBPL and DBASE IV, do not support foreign key constraints.

**D4.2 - Other Static Constraints**

Strictly speaking, these are a special case of static attribute constraints (D3.3) where more than one relation is involved. The same comments thus apply.

In the special case of *derived attributes*, the DBMS might support a way of specifying them and, in this case, a simple translation would be used. If it does not, there are basically two possibilities to implement derived attributes: as queries that are called to calculate their values every time they are needed, and as explicit attributes that cannot be updated by the users. Again, the details are omitted.

**D5 - The $\Delta DB$ Schema**

Usually, there is nothing extra to be done in this case.

**D5.1 - Dynamic Constraints**

Dynamic constraints also depend on the previous values of the updated attributes, but are essentially similar to other constraints. Thus, in the unlikely case they are supported, the appropriate syntax should be used, otherwise the appropriate tests to enforce them are generated from the error schemas associated with the transactions.

**D6 - The $\Xi DB$ Schema**

Again, there is nothing extra to be done in this case.

**D7 - Initialization**

Most DBMSs create empty relations. Otherwise, the user must ensure that all relations are created empty. In DBPL the user must provide an initialization transaction for each database module. As in other parts of the translation, the need to explicitly write the types of the tuples of each relation is the only difficulty.

```
TRANSACTION Init_DB;
BEGIN
    empls  :=  REL_EMPL { };
    ... other relations initialization ...
END Init_DB;
```

## 3.2 Mapping the Database Operations

Now, the discussion moves on to the translation of the operations specified according to the method. For organizational purposes, the operations are divided into two groups: read-only operations, which do not modify the database, and update operations, which modify the database by inserting, updating or deleting tuples of relations.

The input and output variables declared in the specifications are to be translated to variable declarations and input and output commands of the implementation language respectively. Notice that only simple input/output commands are to be generated. In languages which support transactions as special procedures (e.g. DBPL), these variables may alternatively be passed as value and variable parameters respectively. Again, these should only involve straightforward translation (details are omitted).

### B1 - Read-only Operations

If the DBMS/query language supports set at a time operations, the translation should be reasonably simple. Mapping set at a time to record at a time operations should not be too difficult but it is not going to be investigated at this point, even though it is needed for the case of query languages embedded in imperative programming languages. From now on, it is assumed that set at a time operations are supported.

### B2 - Select

Mapping the set comprehension used in the specification of selects to the supported implementation syntax should be reasonably straightforward.

In DBPL, the selection $res! \ = \ \{ \ t : rel \ \mid <\text{condition}> \}$ should be translated to

```
res  :=  REL_RES  {  EACH  t  IN  rel :  <condition>  };
```

where `REL_RES` is the type of the tuple of relation variable `res` and `<condition>` is a boolean expression involving at least one of the attributes of `t`, i.e., relation `rel`.

### B3 - Theta-join

Mapping the set comprehension used in the specification of theta-joins should also be simple. In DBPL, $res! \ = \ \{ \ t1 : rel1; \ t2 : rel2 \ \ldots \ \mid \ t1.Att1 \ <\text{cop}> \ t2.Att2 \ \ldots \}$ should be translated to:

```
res  :=  REL_RES  {  {a, b, ...}  OF
                EACH  t1  IN  rel1,  EACH  t2  IN  rel2,  ...  :
                                   (t1.Att1 <cop> t2.Att2 ...)  };
```

where `REL_RES` is the type of the tuple of relation extension variable `res`, `{a, b, ...}` is the list of all attributes of all relations joined, `<cop>` is a comparison operator, and `Att1` and `Att2` are attributes of `rel1` and `rel2` respectively.

Obviously, this need to list all attributes `{a, b, ...}` of all relations joined is a drawback of DBPL, but this should *not* add any difficulty to the mapping. In addition, in practice most joins are used together with projects and so this drawback should not be particularly relevant.

### B4 - Project

The mapping of the project operations should also be straightforward. In DBPL, the equation $res! \ = \ \{ \ t : rel \ \bullet \ <\text{result}> \}$ should be translated to the following:

```
res  :=  REL_RES  {  { <result> }  OF  EACH  t  IN  rel :  TRUE  };
```

where `REL_RES` is the type of the tuple of relation extension variable `res`, and `<result>` is the list of attributes of `rel` on which the project operation is based.

It is important to point out that this mapping does not cover all possible cases because the specification method also allows for the more general (extended) project operation which permits the use of any computations based on attributes of the relation in the projection list.[2]

Notice however that it is still possible to write such projections in DBPL but the mapping would be a bit more complicated. Furthermore, these operations would not merge with selects and specially theta-joins as easily as before. The code to implement them would involve an explicit `FOR EACH` loop over the relation, and the insertion of a tuple at a time through an auxiliary tuple variable. Again, the details are omitted.

### B5 - Update Operations

In general, the $\Delta DB$ and $\Xi DB$ expressions are not going to be used for anything because, in the implementation, (1) the relations before and after the operations are usually denoted by the same name and (2) relations not used remain unchanged.

The details of the mapping of specific operations are described in the five following rules (B6 to B10) and in the following subsection.

### B6 - Insert

The translation should also be straightforward for most DBMSs, even if set at a time insertion is not supported. In DBPL, the equation $empls' = empls \cup sempl?$ is to be translated to the one below, where `sempl` is the set of new employees to be inserted.

```
empls  :+  sempl;
```

### B7 - Delete by Primary Key

In general, current DBMSs and query languages do not support the definition of higher order functions with generic types. If these were supported, the approach would be to write a generic higher order function to implement the operator *DELETE* used in the specifications, and this would make the mapping very simple.

Assuming these are *not* supported, the solution is then to translate all occurrences of *DELETE* to the appropriate delete command of the DBMS/query language chosen. Some problems might arise if, for instance, set at a time deletions are not supported or if the delete command takes complete tuples to delete instead of their primary keys only (like in the specifications), but the translation should still be reasonably simple.

In DBPL, the deletion $empls' = DELETE\ empls\ ENum\ sekey?$ would ideally be translated to the delete command below, where `REL_EMPL` is the type of the tuple of relation `empls`, and `sekey` is the set of primary keys of employees to be deleted.

```
empls  :-  REL_EMPL  { EACH  t  IN  empls :  t.ENum  IN  sekey };
```

However, this latter use of `IN` (set inclusion) is not implemented and, so, the above specification should be translated to:

```
empls  :-  REL_EMPL  {  EACH  t  IN  empls  :
                                  SOME  k  IN  sekey  (t.ENum = k)  };
```

[2] Actually, according to the definition of the language these computations should be valid in DBPL but they have not been implemented.

When the primary key of the relation is the target of one or more foreign keys, either in other relations or in the same relation, the foreign key compensating actions should also be written as part of the delete by primary key operations, similarly to the way it is done in the specification level. Although the mappings of these compensating actions were not fully investigated, a partial discussion on these mappings is presented in rules B7.1, B7.2 and B7.3.

### B7.1 - Deletes Restricted

The specification of *Restricted* is normally done by default and so no implementation is needed. Explicit redundant specification of *Restricted* should thus be ignored.

### B7.2 - Deletes Cascade

In the case of *Cascades*, the specification depends on whether the foreign key is part of a cycle of foreign keys that cascade for deletes or not, and so does the translation.

**(A)** If the foreign key $Fk1$ in relation $rel2$ is *not* part of such a cycle, and usually this is the case, the schema that specifies deletes in relation $rel1$ includes the expression

$$\textbf{let} \;\; sdr2 \;==\; \{\; t2 : rel2 \;\mid\; t2.Fk1 \in sk1? \;\bullet\; t2.Pk2 \;\} \;\bullet \\ Delete\_Rel2\_Pk \;[sdr2 \;/\; sk2?]$$

which says that the primary keys of tuples of relation $rel2$ referring to deleted tuples of relation $rel1$ by the foreign key $Fk1$ are collected and passed to the schema that specifies deletes by the primary key in relation $rel2$, i.e., $Delete\_Rel2\_Pk$.

If relation $rel2$ has a composed attribute primary key, the specification still follows the rule. For example, suppose the foreign key attribute *ENum* in relation *works* of the company database example had *Cascades* chosen for deletions on relation *empls*. Then, schema *Delete_empls_Pk* which deletes employees by the primary key would include the following equation:

$$\textbf{let} \;\; swkey \;==\; \{\; w : works \;\mid\; w.ENum \in sekey? \;\bullet\; (w.ENum, w.PNum) \;\} \;\bullet \\ Delete\_works\_Pk \;[swkey \;/\; sw?]$$

The general idea is to implement these specifications as combined select and project operations which return a set of primary keys of the relevant relations according to rules B2 and B4 already presented. These keys should then be passed to the operations that implement deletes based on the primary keys of the corresponding relations.

In DBPL, the translation is carried out quite naturally and uses an auxiliary variable (mapped from the **let** expression) and a parameterized procedure call to the relevant delete procedure (`Delete_works`). The implementation code derived from the above specification is given below:

```
swkey  :=  REL_WORK_K  { {w.ENum, w.PNum}  OF
                                 EACH  w   IN  works  :
                                    SOME  k  IN  sekey  (w.ENum = k) };

Delete_works  ( swkey );
```

where `swkey` is an auxiliary variable of type `REL_WORK_K`, which is the relation type of the *set of primary keys* of relation `works`, and `sekey` is the set of primary keys of employees to be deleted.

**(B)** When the foreign key *is* part of a cycle of foreign keys that cascade for deletes, the effect of *Cascades* is not specified in the same way, because there can be no cycles in the use of schemas as predicates.

In the general case, the specifications merely state that, after the deletion of the set of keys selected, all relations after the operation are the maximal subsets of the original relations to satisfy the database constraints. The mapping of this case in general should be very difficult and is not going to be investigated.

The problem is that the information on the relevant foreign keys is not available in this part of the specification. It is probably still possible to work out what are the foreign keys which need to be considered, but it would involve checking the foreign key constraints and maybe even the specification of the other delete schemas.

However, if the DBMS and/or query/host language supports recursion, the only difficulty should be identifying these relevant foreign keys, since the target implementation should be basically the same as presented in case (A), except that an explicit stop condition for the recursion must be provided within the procedure that implements deletes by the primary keys in *one* of the relations involved. Even so, it is important to point out that the existence of such a cycle of *Cascades* is not common and should be avoided whenever possible, because it can potentially destroy the database.

**(C)** In the particular case of delete *Cascades* where the foreign key is targeted at the same relation (loop), the operator *CASC_DELETE* is used in the specification and the relevant foreign key is explicitly mentioned. Hence, if recursion is supported, the implementation code is similar to that presented in case (A), the difference being the delete procedure call, which is recursive and is only activated if there are foreign key references to deleted tuples. So, the mapping is again relatively simple.

Assuming a foreign key specified as *FOR_KEY depts SupENum empls ENum* (supervisor of employees) were to cascade for deletes, the predicate of the corresponding delete schema would be: $empls' = CASC\_DELETE\ SupENum\ empls\ ENum\ sekey?$.

In DBPL, the body of the corresponding implementation procedure `Delete_empls`, presented below, should include the standard delete command, as presented in the general case (rule B7), followed by the code which implements the cascade deletes option using recursion. Notice that the name of the auxiliary variable, `sekey_aux`, as well as its type, `REL_EMPL_K`, are mapped from variable `sekey`, the set of primary keys of employees to be deleted, which is a parameter of *CASC_DELETE*.

```
empls      :-  REL_EMPL    {  ...  sekey  ...  };
sekey_aux  :=  REL_EMPL_K  {  e.ENum  OF
                                 EACH  e  IN  empls  :
                                       SOME  k  IN  sekey
                                             (e.SupENum = k)  };
IF  sekey_aux  <>  REL_EMPL_K { }  THEN
    Delete_empls  ( sekey_aux );
END;
```

In addition, it is worth mentioning that, although the target code is basically the same as that of case (A), the mapping is completely different and a lot simpler than that of case (A) because most of the generated code is embedded in the mapping and pasted into the implementation whenever the *CASC_DELETE* operator is used.

However, when recursion is not supported, it should not be difficult to simulate it with an explicit iteration of deletions and, thus, the mapping should still be simple.

Basically, the implementation code would use an intermediate variable, initialized with the set of keys to be deleted, and a `WHILE` loop containing the appropriate delete command which would iterate while there are foreign key references to deleted tuples.

In DBPL, the implementation code without using recursion is presented below, where the name and type of the auxiliary variable `sekey_aux` are mapped as before, and "..." stands for omitted details which are unchanged from the previous case.

```
sekey_aux  :=  sekey;
WHILE  sekey_aux <> REL_EMPL_K { }  DO
       empls  :-  REL_EMPL  {  ...  sekey_aux  ...  };
       sekey_aux  :=  REL_EMPL_K  {  ...  sekey_aux  ...  };
END;
```

### B7.3 - Deletes Nullify

In the case of *Nullifies*, as in the general delete (rule B7), the ideal implementation would be to write a generic higher order function to implement the operator *UPDATE* used in the specifications, and this would make the mapping very simple.

Assuming these are *not* supported, the solution is then to embed the knowledge of *UPDATE* in the mapping and use the appropriate command or sequence of commands of the chosen DBMS/query language to implement it[3]. Therefore, the aim here is to generate the code that changes to *null* all foreign key references to deleted tuples.

As far as the implementation is concerned, there is in general no need to explicitly say that some tuples are not going to be changed. Basically, the tuples that will be changed are to be selected, and the change to be made and then effected, i.e. written to the database. Changing to *null* the value of the relevant attribute in the selected tuples should map easily to an assignment in the implementation language.

For example, if the foreign key attribute *ManENum* (manager of departments) were to be nullified whenever managers were deleted, the predicate of the delete schema would include the equation: $depts' = UPDATE\ depts\ ManENum\ sekey?\ NULLNAT$. The corresponding DBPL implementation code is given below, where `NULLNAT` is the appropriate null value.

```
FOR  EACH  t  IN  depts :  SOME  k  IN  sekey  (t.ManENum = k)  DO
     t.ManENum  :=  NULLNAT;
END;
```

In the specifications, when *Nullifies* is chosen and the foreign key refers to the same relation, a **let** expression and a local variable are used to join the equation above with the one that specifies the deletions (B7) because the result of one operation must be the input to the other. Still, this distinction is not needed in the implementation level.

For instance, if the *Nullifies* option were chosen for the supervisor of employees foreign key attribute *SupENum*, this effect would be specified using a **let** expression. Even so, the corresponding implementation in DBPL should be:

```
FOR  EACH  t  IN  empls :  SOME  k  IN  sekey  (t.SupENum = k)  DO
     t.SupENum  :=  NULLNAT;
END;
```

[3]In fact, this strategy is to be used whenever an operator is used in the specification, unless explicitly stated otherwise. From now on this is not going to be mentioned anymore.

Observe that this mapping strategy should also be the base for the translation of the operator *UPDATE* used in other contexts, with the possible exception of the updates of primary keys, discussed in rule B10. Moreover, the implementation code embedded in the mapping of *UPDATE* should be very similar to the code to be generated for the updates of attributes (rule B9).

### B8 - Other Deletes

In the specification level, any other deletes are written in terms of combined select and project operations, which return the set of primary keys of the relevant relations, and the schemas that specify deletes based on the primary keys of the relations.

This is exactly how the general delete *Cascades* option is specified – rule B7.2 (A) of the method. Therefore, the corresponding implementation code, omitted here, should be mapped similarly.

It is worth noticing that, essentially, this kind of deletes adds nothing to the expressivity and complexity of the specification method since the same results can be achieved by writing an extra subtransaction that selects the primary keys of the tuples to be deleted and passes the result to the appropriate delete subtransaction.

Additionally, the generated code is in both cases inefficient because the relation will in general be read twice.

### B9 - Updates

In theory, the implementation code for the updates of attributes should be similar to the code generated in the mapping of the operator *UPDATE*, described in rule B7.3. The mapping however is rather different because the specifications of updates are written in terms of a select condition and an update rule, instead of using *UPDATE*.

The general rule for the specification of updates is given below, where <condition> is a boolean expression based on attributes of relation *rel* and <result> is an expression that, applied to any tuple $t$, returns the corresponding updated tuple.

$$rel' \;=\; \{\; t : rel \;\bullet\; \mathbf{if}\;\; \langle\text{condition}\rangle \;\; \mathbf{then}\;\; \langle\text{result}\rangle \;\; \mathbf{else}\;\; t \;\}$$

Even though <result> may be any expression of type *REL*, it usually is an expression which uses the *But* (\) operator to define the value of the the modified attributes, and this is the only case that is going to be investigated in detail.

As mentioned in rule B7.3, there is in general no need to explicitly say that some tuples are not going to be changed. Moreover, changing the value of attributes should map easily to assignments in the implementation language and, indeed, this is the case of *But* (\) expressions of the form $t \setminus (Att1 = v1, Att2 = v2, \; ...)$ .

Therefore, mapping the above specification to the appropriate implementation code should be straightforward. In DBPL, updates using *But* (\) expressions should be implemented by the following code:

```
FOR  EACH  t  IN  rel :  <condition>  DO
     t.Att1 := v1;  t.Att2 := v2;  ...
END;
```

Another possibility is to specify <result> as a tuple of type *REL* such that, for each of its attributes, there is a corresponding expression which delivers the new value of the attribute in the relation tuples, usually in terms of its previous values. In other words, the attributes of <result> are expressions which usually involve computations using the previous values of the attributes in the relation tuples ($t.Att1, t.Att2$, etc.), even though constants can also be used and some attributes might be unchanged. Although mapping this case to the appropriate implementation should still be simple, it is not discussed further.

**B10 - Update of Keys**

In the relational model, the update of the primary keys of one or more tuples of a relation is usually identical to the update of any other attribute of the relation.

Therefore, the corresponding implementation should be strictly similar to the one described for the case of general updates (rule B9) and also in the *Nullifies* option of deletes based on the primary keys (rule B7.3).

Depending on how these updates are specified, i.e. whether the *UPDATE* operator is used or not, the mapping should be strictly the same as that of one of the two aforementioned rules and is not going to be repeated.

When the primary key of the relation is the target of one or more foreign keys, either in other relations or in the same relation, the foreign key compensating actions should also be written as part of the updates of the primary key operations. Since these are very similar to the case of deletes based on the primary keys (B7), the details are omitted once again.

## 3.3 Mapping the Advanced Features

This subsection refers to the mapping of the more advanced features, which include transactions, sorting of results, composite attributes, etc. However, only the mapping of the transactions (correct behaviour) is presented here, due to limitations of space. The extended transactions including error handling are presented in the next section.

**A1 - Transactions**

Transactions are more complicated operations, potentially involving several simpler operations, which must execute as a whole or fail completely. Most RDBMSs support the definition of transactions and the appropriate syntax should be the target code.

The most common way of supporting transactions is by delimiting their scope with two special commands provided to the user: one to start a transaction and the other to end it successfully. Some DBMSs implicitly insert these delimiters in the beginning and at the end of application programs so that, by default, each program is a transaction. In others, these delimiters are written as a special kind of procedure.

A third command usually allows the user to abort the transaction at any time and will normally undo all the database updates already done. The component operations are simply written within the transaction scope using the normal syntax.

Regardless of these implementation details, the mapping should be simple for most DBMSs. If procedures are supported by the query/host language, they should be used to separate the correct behaviour of the transactions from the error handling code.

In DBPL, a transaction is just a special kind of procedure, the difference being it starts with the keyword `TRANSACTION`. Also, there is no automatic undo facility for unsuccessful transactions. The approach to the mapping of the correct behaviour of the transactions to DBPL is to write them and their subtransactions as procedures, named after the corresponding specifications. Input and output variables are to be passed as value and variable parameters, respectively. Parameters of the subtransactions which are not parameters of the transaction should be mapped to local variables. The target DBPL code of a simple transaction, excluding error handling, is presented below.

```
TRANSACTION  Salary_dept  ( d: DNUM;  VAR tot_sal: SALARY;
                                      VAR result:  STRING );
    PROCEDURE  Salary_dept_Ok;       (**  Correct Behaviour  **)
    VAR  sempl:  REL_EMPL;
    BEGIN
        Empls_of_dept     ( d,  sempl );
        Sum_salary_empls  ( sempl,  tot_sal );
    END Salary_dept_Ok;
BEGIN
    ... error handling code ...
    Salary_dept_Ok;
    result  :=  "Success";
END Salary_dept;
```

## 3.4 The Extended Operations for Error Handling

According to the specification method, the predicates of the error schemas associated with the transactions are written as sequences of expressions of the form presented below, connected to each other by logical disjunctions ($\vee$).

$$( \text{<condition>} \quad \wedge \quad \mathit{result!} \; = \; \text{“}\mathit{error\ message}\text{”} \; )$$

In the expression above, <condition> stands for generic predicates representing the logical conditions which violate the precondition of the transaction. In general, each of them involves a combination of predicates connected by logical conjunctions ($\wedge$), disjunctions ($\vee$), and/or negations ($\neg$), as well as existential quantifiers ($\exists$).[4]

The approach here is to map each of these generic predicates to the appropriate piece of implementation code that evaluates it and verifies, using a conditional statement, whether the precondition is violated. Whenever one of these predicates is *true*, the transaction must fail. This means that all changes which might have already been made must be undone, so that the consistency of the database is guaranteed. If an undo facility is supported it should be used whenever appropriate. Otherwise, no change should be made to the database before all such predicates are checked.

For most DBMSs, it should be easy to map the aforementioned predicates to conditional statements of the implementation language, except for the existential quantifiers. When the DBMS supports existential quantifiers, the appropriate syntax should be used and the translation ought to be simple. Otherwise, the result of the evaluation of the existential quantifiers should be assigned to auxiliary boolean variables. These variables should then be used in the conditional statement.

These existential quantifiers are always based on relations and attributes, since they are derived from the precondition of the transactions. Therefore, the evaluation

[4] Expressions involving the universal quantifier ($\forall$) can be rewritten using the existential quantifier, because $\forall x : T \bullet p$ is equivalent to $\neg (\exists x : T \bullet \neg p)$ for any predicate $p$.

of these expressions can be implemented by checking whether the relevant read-only operations (i.e. select, join, and/or project) implicitly specified by their predicates actually return any data. If they do, the result is *true*, else the result is *false*.

For example, suppose *Salary_dept* is a transaction that returns the total salary of employees working for a given department $d?$. The predicate of the corresponding error schema *Salary_dept_Error* is given by:

$$( \neg (\exists\, dp : depts \bullet dp.DNum = d?) \wedge \\ result! = \text{``}Invalid\ department\ number\text{''} )$$

In DBPL, the universal and existential quantifiers can be mapped directly to the `FORALL` and `SOME` commands, which simplifies the problem. Thus, the corresponding DBPL error handling implementation code should be:

```
IF  NOT  (  SOME  dp  IN  depts  ( dp.DNum = d )  )  THEN
    result  :=  "Invalid Department Number";
    RETURN;
END;
```

If DBPL did not support the existential quantifier, it would be simulated by testing whether the select operation $\{dp : depts \mid dp.DNum = d?\}$ returns any tuples, and the result would be assigned to an auxiliary boolean variable as follows:

```
IF  REL_DEPT  {  EACH  dp  IN  depts :  dp.DNum = d  }
                                <>  REL_DEPT  { }  THEN
    exist_aux  :=  TRUE;
ELSE
    exist_aux  :=  FALSE;
END;
```

The error handling implementation code presented before would then follow, but the auxiliary variable `exist_aux` would substitute the existential quantifier (`SOME ...`) in the conditional statement. The resulting code is presented below.

```
IF  NOT  exist_aux  THEN.
    result  :=  "Invalid Department Number";
    RETURN;
END;
```

# 4 The Prototype

This section summarizes the functionality and implementation details of the prototype that is being built to support the method. It provides a syntactic editor for the method and partially implements the mapping described in Section 3. Its outputs are specifications written in Z (using the syntax provided by the `zed.sty` [17] style option for LaTeX) and relational database applications written in DBPL, respectively.

Since the tool being developed is only a prototype, it was necessary to decide what should be included in its functionality. Firstly, it was decided the syntactic editor should accept a large subset of all possible specifications which are correct according to the method, even though checking all the syntax details to enforce the method and reject ill-formed specifications would have to be given a low priority.

Another design decision was to embed part of the semantics of the method in the editor to generate the specifications automatically as much as possible, so that the actual typing done by the user would be restricted to a minimum.

Regarding the implementation of the mapping, the policy is to produce DBPL code from at least a subset of the operations, advanced features, and error handling schemas described in the method, in addition to the structure of DBPL databases. Even so, such a subset should be large enough to permit the automatic generation of syntactically correct programs, at least for some transactions.

One of the design decisions which proved to be very useful was to write the target DBPL programs beforehand, because it provided a concrete target for the mapping process. It also helped to find errors and omissions on the description of the mapping. Sometimes, the ideal implementation code proved to be too difficult to map and, so, these programs were changed so that the mapping could be as smooth as possible.

Although the prototype is not finished yet, a considerable part of the aforementioned functionality has already been implemented. Except for the specification of more complicated transactions, the code which generates the syntactic editor for the method and the corresponding DBPL implementation code has already been written.

The tool used to build the prototype is the Synthesizer Generator [18, 19], which is a system for automating the implementation of language-based syntactic editors. Basically, the user provides a specification written in the Synthesizer Specification Language (SSL) and the system creates a syntactic editor for the language.

Specifically, the synthesizer generator supports the definition of views for the display of different information, possibly using different levels of abstraction, which are automatically updated by the system. This feature makes it very easy to generate code written in a different language as long as the mapping is well defined. In particular, this facility is used to automatically generate DBPL programs as a result of using the prototype syntactic editor that supports the method.

Finally, it is worth emphasizing the description presented above is not a mere list of features of an imaginary tool, but the actual features of the prototype being built. Furthermore, notice that the implementation work is well advanced and a limited version of the tool is already running.

# 5 Conclusion

In summary, this paper partially described a mapping process for deriving relational database programs from formal specifications written in Z according to a well defined set of rules called the method. It also briefly described a prototype which is being built using the Synthesizer Generator, supports the method, and partially implements the mapping. The prototype generates relational database applications to be run in the DBPL system, a RDBMS built in the University of Hamburg.

It was claimed most previous approaches to the derivation of databases programs had not properly addressed the problem, because the problem was either kept too general, without being restricted to any particular database model, or greatly simplified, by not addressing the specification of the database constraints and/or more complicated transactions. The work described here is restricted to the relational database model and addresses all possible constraints as well as generic transactions.

Specifically, a method for the formal specification of relational database applications is provided. The method is aimed at formalizing the design of real relational database transactions and, so, it should help practitioners in the development of real world applications. In addition, the method is generic and may be the first step in the direction of the formal development of database applications and of specification standardization in this context. Moreover, it should improve the system documentation and the quality of the application programs which should contain fewer errors.

Most of the paper was devoted to the presentation of the mapping. It discussed the problems involved in the derivation of relational database programs directly from specifications written according to the method and described the target implementation code for a subset of the method. Even though most examples were written in DBPL, the mapping is general and should be equally applicable to many RDBMSs.

A prototype syntactic tool which aims to support and enforce a reasonably large subset of the method and is currently being developed was also briefly described. The tool used is the Synthesizer Generator, which is a powerful system that allows for the generation of syntactic editors fairly quickly, as long as the syntax and semantics of the target language are well defined. Writing the SSL code per se is fairly straightforward. The effort to learn the basic features of the system was also reasonably small.

In particular, the view facility of the Synthesizer Generator is used to automatically generate parts of relational database programs written in DBPL. The syntactic editor which supports and enforces the method is a bonus. Eventually, the mapping process may be instantiated for a RDBMS offering SQL as its data-sublanguage.

To conclude, it is believed the mapping process described here as well as its actual prototype implementation for DBPL (and indeed for most RDBMSs) are neither too easy nor too difficult. Moreover, it is claimed this work provides evidence that the application of formal techniques in the development of real life software is feasible. Even though there is no formal proof that the mapping retains all the properties of the method, the well-defined semantics of the relational model together with extensive testing of the prototype suggests this is indeed the case.

# Acknowledgements

Thanks are due to Ray Welland, my supervisor, for comments and suggestions on earlier drafts of this document. The financial support from CAPES (Brazilian Federal Agency for Postgraduate Education) and UFPE (Federal University of Pernambuco) are also acknowledged.

# References

[1] Date C. J. *An Introduction to Database Systems*, volume 1. Addison-Wesley, Reading, Massachusetts, USA, fifth edition, 1990.

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

[3] Spivey J. M. *The Z Notation: A Reference Manual.* Prentice Hall International (UK) Ltd., Hemel Hempstead, UK, second edition, 1992.

[4] Potter B. F., Sinclair J. E., and Till D. *An Introduction to Formal Specification and Z.* Prentice Hall International (UK) Ltd., Hemel Hempstead, UK, 1991.

[5] Barros R. S. M. and Harper D. J. "A Method for the Specification of Relational Database Applications". In Nicholls J. E. (ed.), *Z User Workshop, York 1991*, Workshops in Computing series, pages 261–286. Springer-Verlag, 1992.

[6] Barros R. S. M. "Formal Specification of Relational Database Applications: A Method and an Example". Research report GE-93-02, Department of Computing Science, The University of Glasgow, September 1993.

[7] Barros R. S. M. *On the Formal Specification and Derivation of Relational Database Applications*. PhD thesis, The University of Glasgow, Department of Computing Science, 1994. In preparation.

[8] Barros R. S. M. "On the Derivation of Relational Database Programs from Formal Specifications". Research report GE-94-01, Department of Computing Science, The University of Glasgow, July 1994.

[9] Schmidt J. W. and Matthes F. "The Database Programming Language DBPL: Rationale and Report". FIDE technical report FIDE/92/46, Universität Hamburg, Germany, 1992.

[10] Matthes F., Rudloff A., Schmidt J. W., and Subieta K. "The Database Programming Language DBPL: User and System Manual". FIDE technical report FIDE/92/47, Universität Hamburg, Germany, 1992.

[11] Barros R. S. M. "Company Database Example". Department of Computing Science, The University of Glasgow, 1993. Third step specification.

[12] American National Standards Institute. "The Database Language SQL". Document ANSI X3.135, 1986.

[13] Senn J. A. *The student version of dBASE IV, version 1.1 User's Manual*. Addison-Wesley, Reading, Massachusetts, USA, 1991.

[14] van Diepen M. J. and van Hee K. M. "A Formal Semantics for Z and the link between Z and the Relational Algebra". In Bjørner D., Hoare C. A. R., and Langmaack H. (eds.), *VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computing Science*, pages 526–551. Springer-Verlag, 1990.

[15] Date C. J. *Relational Database: Selected Writings*. Addison-Wesley, Reading, Massachusetts, USA, 1986.

[16] Date C. J. and White C. J. *A Guide to DB2*. Addison-Wesley, Reading, Massachusetts, USA, third edition, 1989.

[17] Spivey J. M. "A Guide to the `zed` Style Option". Oxford University Computing Laboratory, Oxford, UK, December 1990. Unpublished.

[18] Reps T. W. and Teitelbaum T. *The Synthesizer Generator: a system for constructing language-based editors*. Texts and Monographs in Computer Science. Springer-Verlag, New York, USA, 1989.

[19] Grammatech, Inc., Ithaca, NY, USA. *The Synthesizer Generator Reference Manual*, fourth edition, 1992.