Specification and Derivation of Relational Database Programs

Roberto Souto Maior de Barros

Departamento de Informática,
Universidade Federal de Pernambuco,
Caixa Postal 7851, Cidade Universitária,
CEP 50.732-970, Recife-PE, Brazil.

roberto@di.ufpe.br

Abstract

The development of database applications is usually carried out informally. The derivation of database programs directly from formal specifications is a well known and unsolved problem. Most of the previous work 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. With appropriate tool support, writing formal specifications according to the method and deriving the corresponding relational database programs can be straightforward. Moreover, it should produce code which is standardized and thus easier to understand and maintain.
1 Introduction

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

In addition, it is unlikely that a generic comprehensive solution to the problem of deriving real applications will be proposed in the near future. Hence, it was advisable to restrict the scope of the research to some well understood domain. The database area, and especially the relational database model [2, 3], seemed to be the perfect target for the utilization of formal methods in this context.

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

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

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 main objective of this paper is to present a summary of the research carried out at The University of Glasgow as part of a Ph.D. degree [4]. The investigation was restricted to the specification and reification of relational database applications. Furthermore, it also considered all relevant kinds of constraints as well as more complicated transactions.

The formal specification language used in this work is Z [5], for a number of reasons. Firstly, model-oriented specification languages seem to be more appropriate to specify database transactions, especially because of the convenient notion of state. Secondly, Z is an established language, probably the most widely used formal specification language, which has been under development for over a decade and is currently being standardized. An extensive literature is also available. Finally, Z is a very flexible language and permits the adoption of different levels of abstraction, even within the same specification document. This gives the specifier the necessary freedom to adopt the most appropriate level of abstraction for each part of the specification.

The first part of this work was the provision of a general method 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).

The second part of this work investigated the derivation of relational database programs directly from formal specifications written according to the method and presented a simple mapping. The mapping discusses the problems involved in the derivation of relational
database programs directly from formal specifications without binding the investigation to any specific database system or language. In other words, the mapping is general and should be applicable to many RDBMSs.

Finally, the third part of this work concerns the prototype tool built to support the method and implement the mapping. The prototype was built to show that the specification of relational database applications using the method and the construction of the corresponding database programs can be reasonably straightforward if appropriate tool support is provided; to provide evidence that the syntax and semantics of the method are sound and that it is possible to build a full scale syntactic editor for the method; and to demonstrate that the mapping can be adapted to specific RDBMSs, that it is possible to derive database programs automatically, at least for a large number of applications, and that building a tool to implement the mapping for a particular RDBMS is not too difficult.

The research was restricted to the relational model for a number of reasons. Firstly, the specification method developed is reasonably simple and does not enforce any constraints on either the real implementation of relations or the choice of a specific database system and language. Also, the proof of properties about such specifications, though not investigated in detail, seems to be fairly easy, involving only first order logic and set theory. Finally, the very high-level nature of its query languages means it seems likely that it is not necessary to use refinement techniques to derive application programs and, moreover, this process did not seem to be arduous.

This paper is organized as follows: Section 2 presents a concise summary of the method. Section 3 provides information about the mapping. Section 4 covers details of the functionality and implementation of the prototype. Section 5 gives suggestions for further work. Section 6 describes related work. Finally, some conclusions are presented in Section 7.

2 The Method

As already mentioned, an important first part of this research was the development of a method for the formal specification of relational database applications. The method provided a formal starting point for the investigation of all other aspects of the work. Therefore, it was vital to improve it as much as possible before proceeding to investigate the other parts because a weak method would probably make the whole work fail.

The 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.

This section presents a very brief summary of the method. The method addresses the definition of domains and relations, the specification of constraints, and querying and updating of relations, including error handling. More advanced features such as transactions, sorting of results, aggregate functions, etc. are also addressed. The complete description of the method was given in [4]. Previous versions were published elsewhere [6, 7].

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. Some examples of domain definitions are presented below.
\[\text{ENUM} \equiv \mathbb{N}\]
\[\text{AGE} \equiv 0..18\]
\[\text{DNUM} \equiv \{ n : \mathbb{N} \mid n > 100 \}\]
\[\text{SEX} \equiv \text{Male} \mid \text{Female} \mid \text{NULLSEX}\]

\[\text{NAME}\]

Relations are specified as sets of tuples, which respects the original relational model defined by Codd [3]. 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.

\[\text{EMPL} \equiv \{\text{Enum} : \text{ENUM}; \text{Sex} : \text{SEX}; \text{Age} : \text{AGE}; \text{DNum} : \text{DNUM}; \ldots\}\]

The relation extension is a variable of type \(\mathbb{P}\) 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 \text{REQUIRED} and \text{KEY_OF} respectively, as well as other static attribute constraints. For example:

\[
\begin{array}{|l|}
\hline
\text{Employee} \\
\text{empls} : \mathbb{P} \text{EMPL} \\
\text{REQUIRED} \text{ empl} \text{ ENum} \land \\
\text{REQUIRED} \text{ empl} \text{ Sex} \land \\
\ldots \\
\text{KEY_OF} \text{ empl} \text{ ENum} \land \\
\forall e : \text{empls} \mid e.\text{Age} > 25 \\
\hline
\end{array}
\]

The state schema, e.g. \(\text{DB}\), 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 \text{FOR_KEY} operator.

\[
\begin{array}{|l|}
\hline
\text{DB} \\
\text{Employee} \\
\text{Depart} \\
\ldots \\
\text{FOR_KEY} \text{ dept} \text{ ManENum} \text{ empl} \text{ ENum} \land \\
\ldots \\
\forall e : \text{empls} \mid (\exists w : \text{works} \mid w.\text{ENum} = e.\text{ENum}) \land \\
\forall d : \text{dept} \mid d.\text{NEmp} = \# \{ e : \text{empls} \mid e.\text{DNum} = d.\text{DNum} \}
\hline
\end{array}
\]
The foreign key specification above means that, for all tuples of relation \textit{depts}, attribute \textit{ManE\textsubscript{Num}} must either be null or match the primary key attribute \textit{E\textsubscript{Num}} of some tuple of relation \textit{empls}.

As in standard Z, other state schemas called \textit{\Delta DB} and \textit{\Xi DB} as well as an initialization schema called \textit{Init\_DB} are defined. The details are omitted.

Read-only operations, i.e. \textit{select}, \textit{join}, and \textit{project}, are specified by schemas such that: (1) they include the \textit{\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 \textit{P \ A}, where \textit{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.

$$\Xi DB$$

\begin{align*}
\exists pj : proj & \quad pj.P\text{Num} = p? \land \\
\text{sempl\_work}\! & = \{ e : \text{empls}; w : \text{works} \mid \\
& \quad w.P\text{Num} = p? \land e.E\text{Num} = w.E\text{Num} \\
& \quad \circ (e.E\text{Num}, e.Salary, w.Hours) \}
\end{align*}

Update operations are specified by schemas that (1) include the \textit{\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 \textit{\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 \textit{empls'} = \textit{empls} \cup \textit{sempl?'}, for inserts; \textit{empls'} = \textit{DELETE} \textit{sempl} \textit{E\textsubscript{Num} se?}, for deletes based on the primary keys; and \textit{works'} = \{ w : \text{works} \circ \text{if} w \in \textit{swork}? \text{then} \ w \ \backslash \ (P\text{Num} = p?) \text{else} \ w \}, for updates of attributes based on a select condition; where \textit{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 (\textit{Restricted}, \textit{Cascades}, and \textit{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 (\textit{\bowtie}) of basic operations written according to other rules of the method. Notice that the version of the piping operator (\textit{\bowtie}) used here is not part of standard Z. It allows for the output \textit{and} primed state variables (\textit{all results}) of the first schema to be matched against the input \textit{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 \textit{Transac\_Ok} is the correct behavior of the transaction, \textit{Oper\_1}, ..., \textit{Oper\_n} are the components of the transaction, and \textit{<condition>} is an additional predicate as mentioned above.

\[ \text{Transac\_Ok} \equiv ( \text{Oper\_1} [b1 / a1, ...] >> ... >> \text{Oper\_n} [b2 / a2, ...] | \text{<condition>} ) \]

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.

3 The Mapping

The second major problem addressed by this work was the derivation of database programs directly from formal specifications. The investigation was restricted to the relational model and considered all relevant kinds of constraints as well as more complicated transactions.

Specifically, a generic mapping aimed at generating relational database programs directly from formal specifications written according to the method was proposed. The mapping described, for a comprehensive subset of the method, what the target implementation code should look like, without binding it to any particular database system or language. However, most examples were written in DBPL [8], the RDBMS used to build the prototype.

The efficiency of the generated code, though taken into account, was not a primary concern. In fact, it was sometimes disregarded in order to make the mapping as smooth as possible. However, this does not mean the generated programs are going to be terribly slow because a number of these operations are optimized by the compiler.

Finally, although an effort was made to keep the generated programs as close to the specifications as possible so that the mapping is simple, it was 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 was also sometimes necessary to incorporate design decisions into the mapping so that the generated programs were syntactically correct.

Full details of the mapping are given in [4]. A simplified version was also published elsewhere [9].

4 The Prototype

This research also involved a substantial piece of implementation. Specifically, a prototype tool was developed. It aims to support the method and instantiate the mapping for a particular RDBMS, namely the DBPL system [8].

The prototype is composed of a syntactic editor for the method and a built-in tool which translates the specifications to database commands. Its outputs are specifications written in \textit{Z} (using the syntax provided by the \texttt{zed.sty} [10] style option for \LaTeX) and relational database applications written in DBPL, respectively.
Since the tool is only a prototype, it does not support the full method. For instance, the syntactic editor accepts a large subset of all possible specifications which are correct according to the method, even though many of the incorrect ones are not rejected.

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.

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.

The DBPL system is an academic tool developed at the University of Hamburg, Germany, which extends the programming language Modula-2 [11] with a new persistent data type called relation and high-level relational expressions based on the predicate calculus.

The main reason for adopting DBPL is the fact that the new type relation and the corresponding access expressions are well integrated with the Modula-2 language to form the database programming language DBPL. As a consequence, it avoids the impedance mismatch which is common in the case of query languages such as SQL [12] being embedded in programming languages such as C or COBOL. In addition, the DBPL system implements a bigger subset of the theoretical relational model than most systems currently available.

The prototype was developed using the Synthesizer Generator [13, 14], which is a powerful tool for implementing language-based editors. It allows for the generation of syntactic editors fairly quickly, as long as the syntax and semantics of the target language are well defined. In particular, the view facility of the Synthesizer Generator was used to automatically generate parts of relational database programs written in DBPL. The syntactic editor that supports and enforces the method is a bonus.

The effort to learn the basic features of the system was also fairly small. It took about two weeks to get the first specification running and another two weeks to experiment with most of the features of the Synthesizer Generator system.

As far as I can see, the main challenge was to come up with a good design for instantiating the general mapping to the particular RDBMS chosen (DBPL) within the time available. Writing the Synthesizer Specification Language (SSL) code for the syntactic editor and using the view facility to generate database programs per se were reasonably straightforward.

Although the prototype has not been finished, a considerable part of the aforementioned functionality has 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 been written.

5 Further work

One natural extension to the method refers to the modularization of the specifications that result from using the method. This can be achieved by using the modularization structures Document and Chapter of Zc [15], also proposed for incorporation in Z [16]. The idea is to split the specification of systems (documents) into several modules (chapters) based on the connections between objects. Specifically, the specification of complex relational databases should be split into several Chapters based on the connections between the
relations, i.e. the foreign keys. The problems that may arise from such a separation and a detailed explanation of what is needed to avoid them may be investigated in the future.

The full treatment for error handling could also be subject of future work. The main objective would be to try to identify, for each of the operations prescribed by the method, all the possible kinds of constraints that might be violated. Moreover, it might even be possible to identify specific equations in the simplified precondition of the transactions that correspond to certain operation and constraint pairs. The results of such investigation could then lead to a more straightforward way of developing the precondition of database transactions written according to the method. The automatic generation of parts of the predicate of the error schemas associated with the transactions might also be feasible.

Also, reasoning techniques could be applied to specifications written according to the method. The aim would be to try and come up with standard theorems about common properties of such specifications and prove them so that the users would be discharged from proving them again. The main benefit would be to prove that the method is sound and that transactions specified according to method do indeed maintain the consistency of the database. One possible approach to prove such theorems could be to generate, using another view in the prototype tool, a version of the specifications written in the specification language adopted by some other system supporting theorem proving, e.g. ADABTPL [17].

Another possibility is to adapt the generic mapping to generate code for another relational database system, possibly a system providing SQL as the query language.

There are a number of other directions in which this research could advance. One of them would be to work on guidelines aimed at maximizing the reuse of specifications of subtransactions. Another way to proceed would be to use a controlled experiment to compare the specification of simple relational database applications written using the method against others written without the method. To be meaningful, such an experiment would have to be carried out using several groups of people with different backgrounds. Finally, it is possible that the method can be adapted for developing object-oriented database.

6 Related work

The approaches most related to the work described here are briefly introduced below.

The work developed by the database group at the University of Hamburg [18, 19], suggests that conceptual database designs should be written using an expressive semantic data and transaction model, namely the TDL language [20]. The database structures and constraints, initially written in TDL, should then be formally transformed into equivalent abstract machines, as defined by Abrial [21]. In the following step, these abstract machines should be refined into other abstract machines that are equivalent to programs written in the strongly typed database programming language DBPL. In other words, they provide specifications that are sufficiently refined to be directly translated to DBPL. The emphasis of this approach was on the derivation of efficient DBPL programs and on proving, formally, that these programs do not violate the database constraints.

The work of Xiaolei Qian [22] discusses the use of refinement techniques (called transaction synthesis by the author) to transform declarative specifications into procedural implementations. In this work, the transaction synthesis is the process of finding a transaction that satisfies the specification. This synthesis is formalized as the process of finding constructive proofs of specification theorems and extracting appropriate transactions from these proofs. Proofs are represented as tables called deductive tableaux which consist of
three lists of formulas: an assertion list, a goal list, and a transaction entry list. The synthesis system consists of deduction rules that manipulate the tableaux preserving its validity. The proof system used to carry out the transaction synthesis is an extended version of the deductive-tableau proof system for first-order logic developed by Manna and Waldinger [23].

Sheard and Stemple [17] present a thorough and theoretically sound treatment for the verification of database transactions safety. They describe a theorem prover that can be used to prove that database transactions are safe in the sense that they do not violate the set of specified database constraints. The formal theory used by the tool is based on the Boyer and Moore [24] style but is extended with higher order functions and theorems. The specification language is called the Abstract Database Type Programming Language (ADABTPL).

A recent conference paper by Pastor and Olivé [25] proposes a method for the generation of transaction specifications concerned with updating views and guaranteeing the integrity of the database. The context of their work is deductive databases [26] and their method augments the deductive database schema with a set of transition rules and internal event rules. A transition rule is a predicate defined in terms of the current state and the integrity constraints of the database, whereas an event rule is a predicate that specifies which operations (usually insertions and deletions) can happen as a result of a database update operation. In addition, the authors describe a prototype tool, implemented in Prolog, which is capable of producing pseudo-code written in English and in Catalan, as well as Prolog implementation code written according to their method.

7 Conclusion

In summary, this paper presented an overview of the research carried out during the last four years at The University of Glasgow and is about the utilization of formal techniques for the development of relational database applications. In particular, it argued that the formal specification and derivation of relational database programs can be made reasonably simple by the provision of appropriate methods and tool support.

In the perfect world, applications ought to be formally specified and modularization techniques used, when necessary, to make the formal specifications easier to understand. In addition, reasoning and/or refinement techniques could be applied before the implementation is actually developed.

This work has addressed the problems of specifying relational database applications formally and of deriving relational database programs directly from the specifications.

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 was provided. The method is aimed at formalizing the design of real relational database transactions and, thus, it ought to 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.

One of the conclusions of this research was that the choice of $Z$ as the formal language for the specification of relational database transactions was an appropriate decision. In the main, the specification method uses only standard $Z$ [5]. Still, most aspects of the method are clear and simple and are defined using a suitable level of abstraction. Also, the extensions to $Z$ used or suggested in this work were kept to a minimum.

Nevertheless the choice of $Z$ in this work does not preclude using other model-oriented languages. This means that the method is generic and that different users may use different specification languages to specify their applications. In particular, a previous paper on this method [27] was written in Zc [15], a strongly-typed $Z$-like language, with minor modifications only.

This work has also proposed a simple translation process to map specifications that result from using the method to relational database applications. The mapping addresses the problems involved in such a process without binding the investigation to any specific database system or language. Also, it is not restricted to the correct behaviour of simple atomic operations. On the contrary, it considers all the relevant kinds of constraints as well as more complicated transactions.

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

A prototype syntactic tool which aims to support and enforce a reasonably large subset of the method was also developed.

The prototype was built using the Synthesizer Generator [13, 14], which is a powerful tool for implementing language-based editors. The implementation of the mapping was carried out using the view facility of the Synthesizer Generator. Writing the SSL code per se was fairly straightforward. The effort to learn the basic features of the system was also reasonably small.

The Synthesizer Generator helped to create appropriate support to using the method for the specification of relational database applications as well as to deriving relational database programs from the specifications.

To conclude, it is believed the mapping process mentioned 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

This paper summarizes material from my Ph.D. thesis [4]. Thus, it is a pleasure to thank again my supervisors, Ray Welland and David Harper. The author is actually supported by a grant from CNPq. The financial support for the PhD course was provided by CAPES and UFPE and are also acknowledged.
References


