Towards The Automatic Validation of Conceptual Specifications For Database Systems - A Formal Approach

Alfio Ricardo Martini Jose M. V. de Castilho, Phd

Instituto de Informatica e PGCC da UFRGS - RS - Brazil
Fax : (051)3365576
E-Mail : MARTINI@INF.UFRGS.BR

Abstract

This paper describes a software tool for the rapid prototyping of database systems, based on their formal conceptual specifications in logic. The prototypes are generated as PROLOG programs. This approach enables one to get user's hands on experience with the software system before its implementation. Throughout this paper, we discuss basic reasons for such formal approach, the structure of the specification language, as well as an overview of the software tool architecture. At the end, we present some conclusions about the contribution of this work to enhance end-users satisfaction with the final product of the data base design procedure.

Key words: first order logic, database systems, conceptual design, formal specifications.

1 Introduction

The conceptual design for database systems, sometimes called data modeling, involves formulating the data objects of the universe of discourse in terms of some data modeling formalism to produce a specification of what it is that the required database is to represent [13].

Informal specifications alone are certainly not appropriate, because they are incomplete, sometimes inconsistent, inaccurate and ambiguous, and they rapidly become bulky, unstructured and therefore non reliable.

Nowadays, in the software engineering community, there is a general agreement that in large or complex software developments (e.g., database systems), formal specifications are a must in order to obtain high quality software [7].

As a consequence, among many others, obvious advantages arise when a formal approach for developing complex software objects is taken [8]:

- They play a role of contract between users and implementor;
- They are a starting point for automatic verification and validation of software systems;
- They can be directly (or, at least, transformed to a suitable form before being) executed;

Rapid (computer-aided) prototyping provides a means for building the scaled-down version of a system more quickly than using other conventional approaches. The final product of the prototyping activity is a working model that can be used for many purposes, such as requirements validation, feasibility study for the full system, and functional specification of a system design.

This work proposes a strategy for systematic construction of system prototypes, from database conceptual models which are formal (rather than informal, such as, for instance, the E-R Model [2]).

In the following sections we discuss the structure of the formal language, an overview of a software tool for the rapid prototyping of database systems and its basic functionality. At the end we present some conclusions about the relevance of this work.

2 The Specification Language

Logic [4] is the most powerful (and general) formal description language known, although not quite user-friendly. As a result, it is reasonable to suppose that it could be the basis of a good communication language between a user and an automatic programming system [10]. Also, besides having a solid theoretical basis upon which one can pursue database theory in general (as discussed in [6]), logic has a *constructive* characteristic, i.e., it enables rapid prototyping by compilation to a executable lower level language.

The conceptual models considered in this work are presented as formal specifications expressed in a mathematical logic language which enables one to describe database systems in a very high level of abstraction [1].

Like any other one-sorted first-order language, the present specification language alphabet consists of primitive symbols such as (1) parenthesis, (2) variables, (3) the

usual connectives, ~ (not), ^ (and), V (or), => (implication), <=> (equivalence), and (4) ForAll and Exists, for the universal and existential quantifiers, respectively.

Furthermore, the language has precisely four 2-place predicate symbols used in infix notation; *_compose_*, *_spec_*, *_type_*, and *_elem_*.

These predicate symbols are based in the "entity type constructor" concept, proposed in the E Model ([11], [12]), a semantic data model for database modeling, which is basically a data type extension to the Entity-Relationship Model [2]. Intuitively, *_compose_* relates entity instances to (each one) of their component entity instances: *_spec_* relates entity instances to their specialized entity instances; *_elem_* relates entity instances of set type to their element instances; and *_type_* relates entity instances to their type denotations.

There are also two 1-place function symbols; *rep(_)* and *card(_)*. The first delivers an integer (or a string) representation of the value of its entity instance argument, and the second, the cardinality of an entity instances set. Constants, denoted by character strings between quotes or by integer numbers, are also part of the language alphabet.

Terms, or arguments, for the predicate and function symbols are entity variables, except for "type", where the second argument is an entity type denotation

A formal specification, in this approach, consists of three parts:

- The declaration of constructed or primitive Entity Types (the "structural" part of the specification) of Integrity Constraints
- The declaration of Update Operations (specified through their pre and post conditions expressed in the logic language),and
- The declaration of Query Operations (also expressed in logic);

As an example, parts of a specification are shown below:

```
SPECIFICATION Employment_Agency

   STRUCTURE:
      person := STRING
      company := STRING
      project := STRING
      contract = person X company
      task = person X project
      .
      .
      .
   CONSTRAINTS:
```

```
    FORALL P, F
       ((P type person ^ F type task
         ^ P compose F) => EXISTS C
        (C type contract ^ P compose C))
    .
    .
    .
OPERATIONS:

    apply(p:string):
    (PRE: not EXISTS P
            (P type person ^ rep(P) = p)
     POST: EXISTS P
             (P type person ^ rep(P) = p))
    .
    .
    .
END_SPEC
```

3 Prototyping System's Overview

The system is basically a translator from the formal specification to PROLOG [3] Horn clauses. Its architecture can be easily understood by examining Fig 1.

The syntactic/semantic analyzer was written in C, and is responsible for checking the well-formedness of the specification as well as its desirable semantic properties (correctness of variable and type declarations, symbol table manipulation, etc...). As an output from this module, one gets a file containing all the integrity constraints and update/query operations expressed in a suitable intermediate language, which can be input to the translator itself.

The translator module is written in C-PROLOG, and consists of four other modules. The *data structure translator* module builds from the data structure definition part of the system's specification, a set of logic formulas that explain in full the meaning of the shorthand notation text defining the database structure.

The *integrity constraints translator* module constructs from logic formulas expressing integrity conditions imposed on the data of the system's database, PROLOG commands (rules) that could perform the corresponding integrity checking. This module is also used to translate, into PROLOG commands, the *structural* formulas produced by the *data structure module.* Both sets of PROLOG commands are used later, during the system's prototyping phase, to test the quality of the system's database contents.

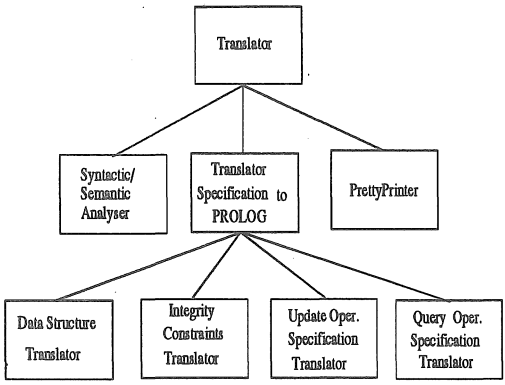

Figure 1: Program's main Architecture

The next module, the *update operations specification translator*, detailed in Fig. 2, executes the main part of the translation process. It synthesizes, from the update operations specification part of the full systems's specification, a set of PROLOG predicates that should behave, when called, like the specified operations of the database system. They consider the existence of a base of "facts" which plays the role of the system's database. As it comprehends the most characteristic and creative part of the translator, this module's behavior will be explained through a small example in the next section.

The last module performs the translation of the query operations specifications to PROLOG queries. It is practically identical to the integrity constraints translator.

4 Update Operations Synthesys

The update operations synthesis profits from the fact that both the specification language and PROLOG are offsprings from first order logic languages. So, some translation steps seem to be rather trivial. However, PROLOG has some procedural characteristics that had to be taken care of in many other steps, so as to accomplish the desired translation.

The module is composed by three submodules : *normal form transformation, atomic formulas translator, and PROLOG operation composition.* The first one implements a series of simbolic transformations that produces from the original pre and post- conditions update operations specification, a formula in prenex normal form, prefixed with only existential quantifiers, negated or not, and whose body is a disjunction of conjuctions of atomic formulas, negated or not. The presentation of the synthesis example starts at this point. Let the normalized formula, representing an

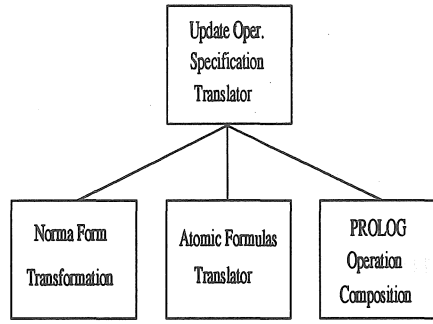

Figure 2: Update Operations Translator Architecture

aplly for a *job* operation (the database system considered here is more or less the same presented in [5]), where a person not already present in the database apllies as candidate for a job (see the apply specification in the example presented in section 2).

```
i)      ~There_Exists Pa There_Exists Pb
            (Pa type person ^ rep(Pa) = p
             ^ Pb m-type person ^ m-rep(Pb) = p)
```

Pa e Pb are variables that may assume entity instances as values, p is (the name of) the person that is applying as candidate for a job; Pa *type* person is an atomic formula that is true if the entity instance denoted by Pa is of type "person". The function symbol *rep*(Pa) yields (the name of) the person denoted by p. The prefix *m-* marks the corresponding atomic formulas as post condition atomic formulas.

In the atomic formulas translation process, each atomic formula is translated to some PROLOG construct, according to the kind of quantification to which it is submitted, and to its condition as negated or not. A small knowledge base of pairs $< atomic\_formula\_pattern, PROLOG\_construct\_pattern >$ is used to perform each atomic formula translation. In the example, the following sequence of constructs is obtained:

```
ii)         not(type(Pa,person), r(Pa,p)),
            gs(Pb), assert(type(Pb,person)),
            assert(r(Pb,p))
```

After this translation step, the following task is to build, as a PROLOG predicate, the whole operation.

```
iii)        operation(apply,p) :-
                    not(type(Pa,person),  r(Pa,p)),
                    gs(Pb), assert(type(Pb,person)),
                    assert(r(Pb,p)), !.

            operation(apply,p) :- put("pre-condition failure"), true.
```

5 Conclusions

In this paper a software tool was described for the automatic generation of databases systems prototypes from their formal conceptual models. A first order language was addopted because of its power, general expressiveness, and its "constructive approach".

Our approach of generating database systems prototypes from their formal conceptual models has interesting advantages:

- The prototypes can be generated very quickly, enabling designers and users to test a series of different system's functionalities.
- Users can make validating experiments with the prototypes and thus, specified requirements can be easily and efficiently revised and improved.
- The formal nature of the specification provides a basis for the use of mathematical reasoning, for the automatic verification of correctness of databases specifications.

References

[1] CASTILHO, J. M. V. de *Especificacoes Formais em Logica para o Projeto de Bancos de Dados*, Tese de Doutorado, DI-PUC/RJ, 1982.

[2] CHEN, P. P. The Entity-Relationship Model-Toward a Unified View of Data, *ACM Transactions on Database Systems*, 1(1), pp.9-36 (March 1976).

[3] CLOCKSIN, W. F. and MELLISH, C. S. *Programming in Prolog*, 2. ed., Berlin, Springer-Verlag, 1984.

[4] ENDERTON, H. B. *A Mathematical Introduction to Logic*, Academic Press, New York, 1972.

[5] VELOSO, P. A. S, CASTILHO, J. M. V. de, FURTADO, A. L. Systematic Derivation of Complementary Specifications, In:*Proceedings 7th VLDB Conference*, Cannes, pp. 409-421 (1981).

[6] GALLARE, H. and MINKER, J. Logic and Databases: A Deductive Approach, *ACM Computing Surveys*, Vol. 16, No. 2, June 1984, pp. 153-185.

[7] HOREBECK, I.V. and LEWI, J. *Algebraic Specifications in Software Engineering - An Introduction*, Springer-Verlag (1989).

[8] JONES, C. B. *Software Development - A Rigorous Approach*, Prentice-Hall International (1980).

[9] LUQI, Software Evolution Trough Rapid Prototyping, *Computer*, Vol. 22, No. 5, May-1989, pp. 15-25.

[10] RICH, C. and WATERS, C. R. Automatic Programming: Myths and Prospects, *Computer*, Vol. 21, No. 8, August 1988, pp. 40-51.

[11] SANTOS, C.S. dos, NEWHOLD, E.J., FURTADO, A.L. A Data-Type Approach to the Entity-Relationship Model, In : P. P. Chen (ed), *Entity-Relationship approach to systems analysis and design*, North-Holland, 1980.

[12] SANTOS, C. S. dos, OLIVEIRA, J. P. W. de, CASTILHO, J. M. V. de O Modelo E, In : *Anais Semish 13*, Olinda, jul 1986, pp. 342-350.

[13] WHITTINGTON, R.P. *Database Systems Engineering*, Clarendo Press, Oxford (1988).