A Generic Process to Refine a B Specification into a Relational Database Implementation

Régine Laleau and Amel Mammar

CEDRIC-IIE(CNAM) 18 allée Jean Rostand, 91025 Evry, France
{laleau, mammar}@iie.cnam.fr

Abstract. In this paper, an approach for refining B abstract specifications describing data-intensive applications into relational database implementations is presented. Using the refinement process of the B method, a set of generic refinement rules are described that take into account both data and operations. The last step consists of mapping the final refined component into a relational database implementation. The different rules have been checked with the AtelierB prover. The aim of the work is to automate the refinement steps. This is possible thanks to the genericity feature of the rules. The approach is illustrated through a running example.

Keywords

B method, Refinement Process, Relational Database Implementation.

1 Introduction

The derivation of database programs from formal specifications is a well known but incompletely solved problem. Most of the previous works on the area was restricted to the derivation of the database structure, mainly because among the different specification languages used in classical systems analysis and design methods (such as UML or OMT) only those describing the system data can be considered as formal. Another frequent mistake has been to consider that once the derivation of the database structure is achieved, the production of transactions is straightforward. However it is not obvious since a major feature of database applications is the great number of integrity constraints that must be verified at all times. Thus programs must be designed such that the constraints are satisfiable in order to guarantee the consistency of the database. This can result in specifying elaborate transactions.

Our solution consists of using the B formal method to design database applications. The system is first specified at an abstract level. We have proposed an approach to combine semi-formal methods (UML and OMT) with the B specification language [1] to facilitate this task. The result is a complete description of the system (both data and operations) that can be proved to be consistent. The following phase is to use the B refinement process in order to derive

database programs. Up to now, we have only considered the development of relational database [7] implementations. In this paper we describe the process that produces the core of a relational implementation: database structure with its main constraints and update SQL statements generated with respect to the constraints. More elaborate transactions generation is an ongoing work.

Up to now, proof activity in a formal development is manual and costly. In order to extend the use of formal methods outside the domain of critical systems, automatic provers need to be developed, as it is suggested in [4]. It is only possible if provers are specialized for particular domains where programs share common characteristics. An important feature of our work is the provision of a generic process refinement available for any relational database specification. Generic rules are defined for each step of the refinement and proved. This should cut down on the number of proofs required and logically enable the development of an automatic refiner.

The remainder of this paper is organized as follows: in Section 2, a brief overview of the B refinement process and the description of relevant characteristics of SQL relational language is given. In Section 3 a concise summary of the global method is presented. Section 4 gives a brief report on different related works. Section 5 presents the main features of the refinement process and describes the most interesting rules that allow an abstract B model to be refined into a concrete B model. More details can be found in [14]. The last step, presented in Section 6, consists of the mapping of the concrete model to a relational database implementation. Section 7 briefly describes through an example how the soundness of the refinement rules can be checked. Lastly, conclusions are presented in Section 8.

2 Overview of the B Refinement Process and the Relational Model

2.1 Overview of the B Refinement Process

Refinement is the process of transforming an abstract specification into a less abstract one. These transformations operate first on data by adding new variables or replacing the existing ones by others which are supposed to be more concrete (closer to a target implementation). Of course, any transformation on variables automatically involves transformations on operations in order to obtain a consistent specification. A refinement can operate on an abstract machine or another refinement component. The last step of the refinement process produces an implementation component which is to be used as the basis for translation to executable code. Both specification and refinement give rise to proof obligations. Specification proofs ensure that operations preserve the invariant, whereas refinement proofs ensure the correctness of a refined component with respect to its initial component.

In order to prove the correctness of refinements, we used the relational semantics of substitutions based on the definition of the two predicates Trm(S)

and Prd(S) which are defined in the B-Book:

$$\mathrm{Trm}(S) \mathrel{\underline{\underline{\Delta}}} [S](x = x) \ . \tag{1}$$

$$\mathrm{Prd}(S) \mathrel{\underline{\underline{\Delta}}} \neg [S](x' \neq x) \ . \tag{2}$$

Intuitively:
Trm(S): gives the necessary and sufficient condition for the good execution of S,
Prd(S): gives the link between the values of the variable x before(denoted x) and after(denoted x') the execution of S.
With this definition, the correctness of refinement can be expressed as follows:
Let S and T be two substitutions that act respectively on the set of variables P and Q. T refines S according to the gluing invariant J(P,Q), denoted S $\sqsubseteq_{J(P,Q)}$ T, iff:

(i) $\exists u, v.\ J(u, v)$.
(ii) $\forall X, P, Q.\ (Trm(S) \wedge J(P, Q)) \rightarrow Trm(T)$.
(iii) $\forall X, P, Q, X', Q'.\ (Trm(S) \wedge J(P, Q) \wedge Prd(T)) \rightarrow \exists P'.(Prd(S) \wedge J(P', Q'))$.

where:
- X is the set of variables that occur in S and not in P or Q,
- X', P', Q' are respectively the set of values of variables X, P and Q after the execution of S and T.
(i) means that the gluing invariant must be satisfiable(it isn't a contradiction) ,
(ii) means that: for each possible interpretation of X and P that ensures the good execution of S, the corresponding set Q (that satisfies the gluing invariant) ensures the execution of T,
(iii) means that: for each possible result Q' of T, there must exist a corresponding result P' of S such that P' and Q' verify the gluing invariant J.

2.2 Overview of the Relational Model

The relational model was defined by Codd [6]. Relations, that we call tables according to the SQL world to avoid confusion with the B concept, are specified as sets of tuples. A tuple is an element of a cartesian product. The formal definition of a table contains two parts : the table intention and the table extension. The intention is a tuple type which defines the table attributes, each of which must be of a valid type. The extension is the set of instances of type tuple which exists at a given moment.

We use SQL-92 syntax [16]. Integrity constraints can be defined on tables:
– NOT NULL constraint : defined on an attribute (or a group of attributes) specifies that the attribute must be valued.
– UNIQUE constraint : defined on an attribute (or a group of attributes) specifies that each attribute value is unique in the table extension.
– Key constraints : each table has at least one key which is an attribute (or a group of attributes). One of them is defined as the PRIMARY KEY of the table.

The others are defined by specifying NOT NULL and UNIQUE constraints.
– Referential constraint : defined on an attribute (or a group of attributes), denoted R1, of a Table T1 towards a key of a Table T2, denoted K2. It specifies that the set of R1 attributes values is included in the set of K2 attributes values.

Other constraints, introduced by the keyword CHECK, can be defined. They represent predicates that must be verified by one or a group of tables.

3 Description of our Approach

The aim of our project is to automatically derive relational database implementations from abstract specifications described with UML notations [18] (object diagram, state diagrams and collaboration diagrams). Moreover each implementation can be proved to be consistent with the corresponding abstract specification. The functional architecture is represented by the following figure:

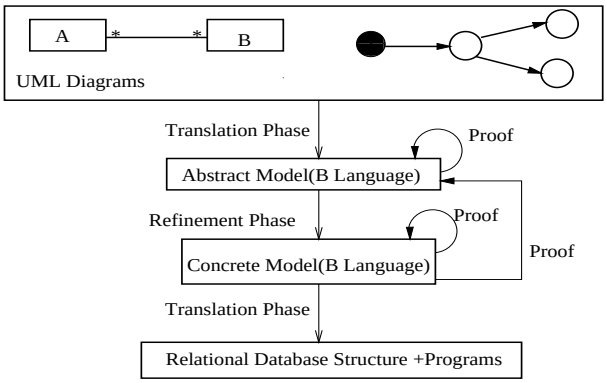

Fig. 1. Formal mapping of a UML conceptual model into a relational implementation using the B method

In order to map a UML conceptual model into a B abstract model, precise rules on the use of UML concepts have been defined [12]. In this paper, we are interested in generating the core of a relational implementation: relational database structure and basic update statements. They are derived only from class diagram that are detailed in the next section.

3.1 From UML Class Diagrams to B Specifications

In this section, we outline the mapping of UML class diagrams into B specifications. Only the main concepts (class, attribute and association) are presented. A complete description can be found in [11], [17].
The simple diagram of Figure 2 is used to illustrate the mapping:

The class *Order* is described by two attributes *numor* and *date*. The label {K} specifies that *numor* is a key for this class (note that a key can be composed of a set of attributes). The class *Product* is described by a key attribute *numpr*

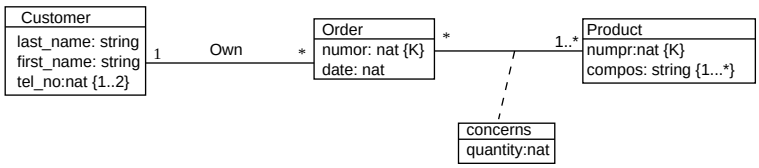

Fig. 2. An UML class diagram

and a multivalued attribute *compos*, specified by the label {1..*}. A *Customer* is specified by his/her last and first names and an attribute *tel_no* (telephone number) which is mandatory and multivalued (specified by the label {1..2} that means that *tel_no* can have 2 values at most). Each order concerns one or more products. Each product may appear in zero or more orders. For each product related to an order, an ordered quantity is specified. Each order is placed by exactly one customer who may place zero or more orders.

• **Classes.** Each class A is formalized by an abstract given set of all possible instance of A and a variable v_A representing the set of existing instances of A. Each attribute Att of a class A is modeled by a variable v_{Att} which type is a binary relation between v_A and the attribute type T_{Att}. Depending on the cardinality of the attribute, the relation may become a total or partial function. T_{Att} is a basic type (including enumerated sets) or built on a basic type using PartOf or Sequence constructor. We impose that UML basic types correspond to B basic types. Up to now, we have not yet considered how to specify specific or aggregate types (such as address type composed of a street name and a town). A key is translated by a total injective function because its value is unique and mandatory in every object.
• **Associations.** We consider only binary associations. This limit is not very important since n-ary associations can always be transformed into binary associations with additional constraints. An association Ass between two classes X and Y is modeled by a variable v_{Ass} defined as a relation between the existing instances of the two classes v_X and v_Y. As for attributes, the type of the relation depends on the cardinality of each role (a role is an association end). An attribute of an association is formalized as an attribute of a class.
In database applications, basic update operations, which are application independent, can be automatically generated from class diagrams. Cardinality constraints are taken into account. In the example, only some of these operations are presented.

Example: The example is translated into the B specification as follows:
MACHINE MODEL
SETS Order, Customer, Product
VARIABLES customer, last_name, first_name, tel_no, order, numor, date, own, product, numpr, compos, concerns, quantity
INVARIANT customer $\subseteq$ Customer $\wedge$ last_name $\in$ customer $\rightarrow$ STRING

$\wedge$ first_name $\in$ customer $\rightarrow$ STRING $\wedge$ tel_no $\in$ customer $\leftrightarrow$ NAT $\wedge$
$\forall x.(x \in \text{customer} \Rightarrow 1 \leq \text{card}(\text{tel_no}[\{x\}]) \leq 2) \wedge$
order $\subseteq$ Order $\wedge$ numor $\in$ order $\rightarrowtail$ NAT $\wedge$ date $\in$ order $\rightarrow$ NAT $\wedge$
own $\in$ order $\rightarrow$ customer $\wedge$ product $\subseteq$ Product $\wedge$
numpr $\in$ product $\rightarrowtail$ NAT $\wedge$ compos $\in$ product $\rightarrow$ STRING $\wedge$
concerns $\in$ order $\leftrightarrow$ product $\wedge$ dom(concerns)=order $\wedge$
quantity $\in$ concerns $\rightarrow$ NAT

INITIALISATION customer, last_name, first_name, tel_no, order, numor, date, own, product, numpr, compos, concerns, quantity:=$\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset$

OPERATIONS /**only some operations are given**/

Add_Customer (cus_1, ln_1, fn_1, t_1)

PRE $cus_1 \in$ Customer-customer $\wedge$ $ln_1 \in$ STRING $\wedge$ $fn_1 \in$ STRING $\wedge$
$t_1 \subseteq$ NAT $\wedge$ card(t_1) $\leq 2 \wedge$ card(t1) ≥ 1

THEN customer:=customer $\cup \{cus_1\}$ || last_name := last_name $\cup \{cus_1 \mapsto ln_1\}$
||first_name:= first_name $\cup \{cus_1 \mapsto fn_1\}$||tel_no := tel_no $\cup \{cus_1\}$ * t1

END;

/***Parameters for adding an order necessarily contain a customer, a product and a quantity in order to guarantee consistent cardinalities**/

Add_Order(ord_1, $numor_1$, $date_1$, cus_1, pd_1, $quantity_1$)

PRE $ord_1 \in$ Order-order $\wedge$ $numor_1 \in$ NAT-ran(numor) $\wedge$ $date_1 \in$ NAT $\wedge$
$cus_1 \in$ customer $\wedge$ $pd_1 \in$ product $\wedge$ $quantity_1 \in$ NAT

THEN order:=order $\cup \{ord_1\}$ || numor:=numor $\cup \{ord_1 \mapsto numor_1\}$ ||
date:=date $\cup \{ord_1 \mapsto date_1\}$ || own:=own $\cup \{ord_1 \mapsto cus_1\}$ ||
concerns:=concerns $\cup \{ord_1 \mapsto pd_1\}$ ||
quantity:=quantity $\cup \{(ord_1 \mapsto pd_1) \mapsto quantity_1\}$

END;

/***Removing an order requires removal of its related concerns links.**/

Rem_Order(ord_1)

PRE $ord_1 \in$ order

THEN order:=order-{ord1} || numor:=$\{ord_1\} \lhd$ numor || date:=$\{ord_1\} \lhd$ date
|| own:=$\{ord_1\} \lhd$ own || concerns:=$\{ord_1\} \lhd\!\!\lhd$concerns ||
quantity:=($\{ord_1\}$*product) $\lhd$ quantity

END;

Add_Product(pd_1, n_1, $comp_1$)

PRE $pd_1 \in$ Product-product $\wedge$ $n_1 \in$ NAT-ran(numpr) $\wedge$ $comp_1 \subseteq$ STRING

THEN product:=product $\cup \{pd_1\}$ || numpr:=numpr $\cup \{pd_1 \mapsto n_1\}$ ||
compos:=compos $\cup \{pd_1\}$*$comp_1$

END;

Rem_Product(pd_1)

PRE $pd_1 \in$ product - ran(concerns)

THEN product:=product-$\{pd_1\}$ || numpr:=$\{pd_1\} \lhd$ numpr ||
compos:=$\{pd_1\} \lhd$ compos

END

END

Let us remark that usually the specification we obtain with our rules is modularized (for more details see [11], [17]). In this paper, the specification is presented as a single module in order to simplify the description of the refinement method.

3.2 Refinement of B Specifications into Relational Database Implementations

All classical systems analysis and design methods propose an algorithm to translate class diagrams into relational schemas [3]. The steps are:
(i) transformation of inheritance links. Three solutions are possible: either the superclass is kept and its subclass characteristics are shifted into it, or the opposite solution, or both super and subclasses are kept and inclusion constraints added.
(ii) transition to the first normal form : in the relational model, all the attribute domains are atomic. Thus a multivalued attribute of a class diagram is replaced either by a new class or by new atomic attributes according to its cardinality.
(iii) transition from an object-based model to a value-based model. In the relational model, each table must have a key. It is a value-based model, that means that each tuple is identified by a key which is a set of attributes. On the contrary, the UML model is an object-based model : each object is identified by an object identifier, independent of its value.
(iv) transformation of association links : an association between classes A and B with at least one monovalued role (attached for example to class A) is replaced by a new attribute in A, which is linked to the key of B by a referential constraint. Other associations are replaced by a new class with two attributes which are linked to the keys of A and B by referential constraints.
(v) finally, each class is mapped into a table.
Using these rules, the example gives the following SQL schemas:

CREATE TABLE Order(
numor **INT PRIMARY KEY,**
date **INT NOT NULL,**
own_1 **INT NOT NULL REFERENCES** Customer(numcus))

CREATE TABLE Product(
numpr **INT PRIMARY KEY**)

CREATE TABLE Customer (
numcus **INT PRIMARY KEY**,
last_name **CHAR(30) NOT NULL**,
first_name **CHAR(30) NOT NULL,**
tel_no_1 **NAT NOT NULL,**
tel_no_2 **NAT**)

CREATE TABLE Compos(

label **CHAR(30) NOT NULL**,
$numpr_2$ **INT NOT NULL REFERENCES** Product(numpr),
PRIMARY KEY (label , $numpr_2$))

CREATE TABLE Concerns(
$numor_1$ **INT NOT NULL REFERENCES** Order(numor),
$numpr_1$ **INT NOT NULL REFERENCES** Product(numpr),
quantity **INT NOT NULL,**
PRIMARY KEY ($numor_1$, $numpr_1$))

Our method produces the same schema using the B refinement process. Basic SQL statements corresponding to the basic operations defined at the conceptual level are also generated (cf Section 6).

An important objective of our work is to automate this refinement process. If this automation seems an arduous task in general, we think it should become possible if we consider applications of a particular domain, which are implemented using the same refinement process. In this framework the automation consists of :

- defining generic refinement rules
- proving the correctness of the rules

These two points are detailed in Sections 5 and 6.

4 Related Work

Direct related works are those of Barros [2] and Günther&al. [13]. Both generate from formal specifications implementations in DBPL which is a more powerful language that SQL. In the second work, they use the B refinement process. However they present their approach only through an example and no formal rules are specified. Moreover no precision is given about a tool.

Barros' method consists of first specifying the most important relational concepts in Z and then defining rules to translate such a Z specification into a DBPL implementation. This last part corresponds to the last step of our method that consists of mapping a concrete B model into a SQL implementation.

The works of Evans [10] and Castelli&al. [5] are more general than our work. They deal with the definitions of formal rules to transform diagrams. Refinement is considered as a specific case of diagrams transformation. They are not interested in automating the refinement process that produces a relational implementation.

The work of Locuratolo and her collaborators [15] deals with refinement for database design but considers object oriented databases.

5 Refinement Rules

Now, a description of the specific B refinement process we have defined is presented. The abstract model is obtained using the method described in Section 3.1 The "concrete model" is to be defined such that the mapping to a relational

implementation is automatic. Thus it is not described in B0 since SQL is more abstract than B0. The different refinements correspond to the algorithm presented in Section 3.2 For each of them, two kinds of rules are specified : first a data refinement rule then refinement rules for the substitutions that occur in the basic operations.

RULE Ri_D IS
ABSTRACT_VARIABLES *a_abstract*
CONSTRAINT C
CONCRETE_VARIABLES *a_concrete*
INVARIANT $I \wedge J$
END

RULES Ri_S IS
REFINE $P \mid Abstract_S$
REFINEMENT $Q \mid Concrete_S$
END

• **Data Refinement Rule (Ri_D)**: this rule specifies for each abs-tract variable *a_abstract* that satisfies the constraint C, the concrete variable *a_concrete*, its type invariant I and the gluing invariant J. The gluing invariant J gives the existing relation between the variables *a_abstract* and *a_concrete*.
• **Substitution Refinement Rules (Ri_S)**: this set of rules refines substitutions that act on variables refined by the corresponding Ri_D. In this paper, we consider two kinds of basic operation: add and remove. Each rule depends on the kind of the basic operation. This gives two refinement rules: *Ri_A* for Add and *Ri_R* for Remove. More precisely, each possible substitution on the abstract variable refined by *Ri_D*, generates one rule *Ri_A* or *Ri_R*. It means that: when the data refinement rule Ri_D is performed then each abstract substitution $P|Abstract_S$ is refined by the concrete one $Q|Concrete_S$. If P does not depend on *a_abstract* then we have decided not to refine P (thus "P=Q"). In this case, preconditions are omitted. For lack of space, only the rules representative of the transformation are presented. For more details, see [14].

5.1 Transition to the First Normal Form

Multivalued attributes and attributes with a complex type (ie. types built on the PartOf or Sequence constructors) must be transformed. Rule 1 achieves this.

Rule 1. Multivalued Attributes. A multivalued attribute *Att* is represented by a relation between the class variable and its type. Its refinement depends on the cardinality of the attribute: if it is bounded by an integer k (k is determined by the designer.), *Att* is split into k new monovalued attributes corresponding to the concrete variables $f_1, \ldots, f_k$. If the cardinality is unbounded or if it is greater than k, the data refinement rule can be considered as a rewriting rule that makes the relation f more suitable for further refinement. A special class is introduced (variable C) with no given set and f is refined by f_1 which becomes a standard relation between two classes.

Data Refinement Rule (Bounded Cardinality)
RULE R11_D IS ABSTRACT_VARIABLES f CONSTRAINT $f \in A \leftrightarrow T \wedge$ $\forall x.(x \in A \Rightarrow j \leq card(f[\{x\}]) \leq k)$ CONCRETE_VARIABLES $f_1, \ldots f_j, \ldots, f_k$ INVARIANT $f_1 \in A \rightarrow T \wedge \ldots \wedge f_j \in A \rightarrow T \wedge$ $f_{j+1} \in A \nrightarrow T \wedge \ldots \wedge f_k \in A \nrightarrow T \wedge$ $f_1 \cup \ldots \cup f_k = f$ END

Data Refinement Rule (Unbounded Cardinality)
RULE R12_D IS ABSTRACT_VARIABLES f CONSTRAINT $f \in A \leftrightarrow T$ CONCRETE_VARIABLES f_1, C, v_{key}^C INVARIANT $C = T \wedge f_1 \in A \leftrightarrow C \wedge$ $f_1 = f \wedge v_{key}^C = id(C)$ END

Note : v_{key}^C denotes a function which represents a key attribute of C.
Applying R11_D to *tel_no* results in the definition of two new attributes specified as follows: $tel_no_1 \in customer \rightarrow NAT \wedge tel_no_2 \in customer \nrightarrow NAT$
The gluing invariant is: $tel_no_1 \cup tel_no_2 = tel_no$
Applying R12_D to *compos* gives:

$Type_comp = STRING \wedge$
$compos_1 \in product \leftrightarrow Type_comp \wedge$
$compos_1 = compos \wedge$
$v_{key}^{Type_comp} = id(STRING)$

The substitution refinement rules corresponding to R11_D are:

Add Substitution
RULE R11_A IS REFINE $f := f \cup \{a\} * b$ REFINEMENT $ANY g_1, \ldots, g_k \; WHERE$ $\wedge_{i=1..k} \; g_i \subseteq T \wedge_{i=1..j} \; card(g_i) = 1 \wedge_{i=j+1..k} \; card(g_i) \leq 1$ $\cup_{i=1..k} g_i = b \wedge \forall i, j \; (g_i \cap g_j = \emptyset)$ $THEN$ $\|_{i=1..k} \; IF \; g_i \neq \emptyset \; THEN \; f_i = f_i \cup \{a \mapsto choice(g_i)\}$ END END END

Remove Substitution
RULE R11_R IS REFINE $f := \{a\} \lhd f$ REFINEMENT $\|_{i=1..k} \; f_i := \{a\} \lhd f_i$ END

Note : $\wedge_{i=1..k} p_i$ is an abbreviation for $p_1 \wedge \ldots \wedge p_k$ and $\|_{i=1..k} \; S_i$ for $S_1 \| \ldots \| S_k$.
In R11_A, a denotes an object of class A, b is a set of values included in T that becomes the new value for the object a of the abstract multivalued attribute represented by f. Intuitively, the refinement rule consists of splitting set b into k atomic values linked to the concrete variables f_i .
In *Add_Customer*, the substitution $tel_no := tel_no \cup \{cus_1\} * t_1$ is refined by:
$ANY \; g_1, \; g_2 \; WHERE$
$g_1 \subseteq NAT \wedge g_2 \subseteq NAT \wedge g_1 \cup g_2 = t_1 \wedge card(g_1) = 1 \wedge card(g_2) \leq 1 \wedge$
$g_1 \cap g_2 = \emptyset \; THEN$

$IF\ g_1 \neq \emptyset\ THEN\ tel_no_1 = tel_no_1 \cup \{cus_1 \mapsto choice(g_1)\}\ END \parallel$
$IF\ g_2 \neq \emptyset\ THEN\ tel_no_2 = tel_no_2 \cup \{cus_1 \mapsto choice(g_2)\}\ END$
END
The rules corresponding to R12_D are obvious (just replace f by f_1).

5.2 From an Object-Based Model to a Value-based Model

Rule 2. Introduction of Keys. For each class, a key is to be exhibited. In B, a UML class is represented by a variable v whose type is a given set. A key corresponds to a total injective function defined on v. If such a function doesn't exist in the initial specification, a new variable v_{key}, that we call primary key, is added as follows:

RULE R2_D IS
ABSTRACT_VARIABLES v
CONSTRAINT $SET(V) \wedge v \subseteq v' \wedge v' \subseteq V$
CONCRETE_VARIABLES v_{key}
INVARIANT $v_{key} \in v \rightarrowtail T^v_{key}$
END

SET(V) means that the set V is a given set. T^v_{key} is a basic B type, such as NAT, STRING, ... chosen by the designer.
The substitution refinement rule consists of adding a new substitution to the existing ones, in order to preserve the invariant.

Add Substitution
RULE R2_A IS
REFINE $v := v \cup \{v_1\}$
REFINEMENT $v := v \cup \{v_1\} \parallel$
$ANY\ x \in T^v_{key} - ran(v_{key})\ THEN$
$v_{key} := v_{key} \cup \{v_1 \mapsto x\}$
END
END

Remove Substitution
RULE R2_R IS
REFINE $v := v - \{v_1\}$
REFINEMENT
$v := v - \{v_1\} \parallel$
$v_{key} := \{v_1\} \lhd v_{key}$
END

In the example, there is no injective function defined on *customer*, so the variable *numcus* is added: $numcus \in customer \rightarrowtail NAT$.
The substitution :$ANY\ numcus_1 \in NAT - ran(numcus)\ THEN$
$numcus := numcus \cup \{cus_1 \mapsto numcus_1\}$ is added to the Add_Customer operation.
In the case where the designer has specified several keys for a class, he/she has to choose one to become the primary key. In order to simplify presentation of the following rules, we consider that a primary key is composed by only one attribute.

5.3 Transformation of Associations

Rule 3. Monovalued Associations. This kind of associations corresponds to associations with at least one monovalued role. In B, such an association is

represented by a function f defined between two class variables C and D. It is refined into a function between C and the primary key of D.

RULE R3_D IS
ABSTRACT_VARIABLES f
CONSTRAINT $f \in C \nrightarrow D \wedge v^D_{key} \in D \rightarrowtail T^D_{key}$
CONCRETE_VARIABLES f_1
INVARIANT $f_1 \in C \nrightarrow T^D_{key} \wedge f_1 = f; v^D_{key}$
END

The substitution refinement replaces each D object by its key value:

Add Substitution
RULE R3_A IS
REFINE $f := f \cup \{c \mapsto d\}$
REFINEMENT
$f_1 := f_1 \cup \{c \mapsto v^D_{key}(d)\}$
END

Remove links from C
RULE R3_R1 IS
REFINE $f := \{c\} \lhd f$
REFINEMENT
$f_1 := \{c\} \lhd f_1$
END

Remove links from D
RULE R3_R2 IS
REFINE $f := f \rhd \{d\}$
REFINEMENT
$f_1 := f_1 \rhd \{v^D_{key}(d)\}$
END

The *own* function of our example is refined by the own_1 function defined as follows:
$own_1 \in order \rightarrow NAT \wedge own_1 = own; numcus$
The different substitutions that act on *own* are refined:
In Add_Order : $own := own \cup \{ord_1 \mapsto cus_1\} \sqsubseteq$
$own_1 := own_1 \cup \{ord_1 \mapsto numcus(cus_1)\}$
In Rem_Order : $own = \{ord_1\} \lhd own \sqsubseteq own_1 := \{ord_1\} \lhd own_1$

Rule 4. Other Associations. This rule refines all the associations that can't be refined by the previous rule. The idea is to "replace" the variable corresponding to the association by two new variables (v_1 *and* v_2) which correspond to the key attribute of each class linked by the association and such that (v_1, v_2) is a key of this association.

RULE R4_D IS
ABSTRACT_VARIABLES f
CONSTRAINT $f \in C \leftrightarrow D \wedge v^C_{key} \in C \rightarrowtail T^C_{key} \wedge v^D_{key} \in D \rightarrowtail T^D_{key}$
CONCRETE_VARIABLES $v_1,\ v_2$
INVARIANT
$(v_1 \parallel v_2) \in f \rightarrowtail T^C_{key} \times T^D_{key} \wedge \quad v_1 = (dom(f)) \lhd v^C_{key} \wedge v_2 = (ran(f)) \lhd v^D_{key}$
END

The substitution refinement rules for a standard association are :

Add Substitution
RULE R4_A IS
REFINE $f := f \cup \{c \mapsto d\}$
REFINEMENT
$f := f \cup \{c \mapsto d\} \parallel$
$v_1 := v_1 \cup \{c \mapsto v^C_{key}(c)\} \parallel$
$v_2 := v_2 \cup \{d \mapsto v^D_{key}(d)\}$
END

Remove links from C
RULE R4_R1 IS
REFINE $f := \{c\} \lhd\!\!\!- f$
REFINEMENT
$f := \{c\} \lhd\!\!\!- f \parallel$
$v_1 := \{c\} \lhd\!\!\!- v_1 \parallel$
$v_2 := f[\{c\}] \lhd\!\!\!- v_2$
END

Remove links from D
RULE R4_R2 IS
REFINE $f := f \rhd\!\!\!- \{d\}$
REFINEMENT
$f := f \rhd\!\!\!- \{d\} \parallel$
$v_1 := f^{-1}(d) \lhd\!\!\!- v_1 \parallel$
$v_2 := \{d\} \lhd\!\!\!- v_2$
END

The *concerns* relation of our example is refined as follows:
$(numor_1 \parallel numpr_1) \in concerns \rightarrowtail NAT \times NAT \wedge$
$numor_1 = (dom(concerns)) \lhd numor \wedge numpr_1 = (ran(concerns)) \lhd numpr$
In *Add_Order*: $concerns := concerns \cup \{ord_1 \mapsto pd_1\} \sqsubseteq$
$concerns := concerns \cup \{ord_1 \mapsto pd_1\} \parallel numpr_1 := numpr_1 \cup \{pd_1 \mapsto numpr(pd_1)\} \parallel$
$numor_1 := numor_1 \cup \{ord_1 \mapsto numor(ord_1)\}$
Let us remark that rule R4_D is also applicable to relations derived by Rule R12_D, corresponding to multivalued attributes. However the Add substitution refinement rule is different :

Add Substitution
RULE R4_A_Multi IS
REFINE $f := f \cup \{c\} * d$
REFINEMENT
$f := f \cup \{c\} * d \parallel v_1 := v_1 \cup \{c \mapsto v^C_{key}(c)\} \parallel v_2 := v_2 \cup id(d)$
END

Applying Rules R4_D and R4_A_Multi to $compos_1$ gives:
$(numpr_2 \parallel label) \in compos_1 \rightarrowtail NAT \times STRING \wedge$
$numpr_2 = (dom(compos_1)) \lhd numpr \wedge label = (ran(compos_1)) \lhd v^{Type_comp}_{key}$
In *Add_Product*, the refined substitutions are:
$compos_1 := compos_1 \cup \{pd_1\} * comp_1 \parallel numpr_2 := numpr_2 \cup \{pd_1 \mapsto numpr(pd_1)\} \parallel$
$label = label \cup id(comp_1)$

5.4 Extension of Partial Functions

Rule 5. This rule consists in extending each partial function, defined from A to T , into a total one. It is necessary because, in the relational model, each attribute has a value, possibly the Null value. In B, extending a basic type to this Null value is not possible, thus we propose to consider for each type a special value which represents this Null value. For example Null value for a string is the empty string, for an integer Maxint. The Boolean type should be simulated by using natural numbers {0,1}. Intuitively, the extension of a partial function f consists of assigning the Null value for each element that does not belong to the domain. Formally, this refinement is specified as follows. Only the Add substitution refinement rule corresponding to the case of optional monovalued attribute is given. Each time a new object is added, a substitution that assigns a Null value to a partial function is added.

Data Refinement Rule
RULE R5_D IS ABSTRACT_VARIABLES f CONSTRAINT $f \in A \nrightarrow T$ CONCRETE_VARIABLES f_1 INVARIANT $f_1 \in A \rightarrow T \cup \{Null\} \wedge$ $f = f_1 \rhd \{Null\}$ END

Add Substitution
RULE R5_A IS REFINE $A := A \cup \{a\}$ REFINEMENT $A := A \cup \{a\}$ || $f_1 := f_1 \cup \{a \mapsto Null\}$ END

Remove Substitution
RULE R5_R IS REFINE $f := \{a\} \lhd f$ REFINEMENT $f_1 := \{a\} \lhd f_1$ END

5.5 From a Functional Representation to a Set Representation

Rule 6. Definition of the Table Structures

This rule consists of mapping a functional model to a set model or more precisely to a model based on the cartesian product which is the formal definition of a table. Rules are defined for building the table corresponding to a class (R6Clas_D) and the table corresponding to an association refined by R4 (R6Ass_D). All the attributes defined by a function from the same source set are gathered in a cartesian product that will give the table structure.

Class Table Structure
RULE R6Clas_D IS ABSTRACT_VARIABLES $f_1, f_2, \ldots, f_n$ CONSTRAINT $f_1 \in A \rightarrowtail T_1 \wedge$ $f_2 \in A \rightarrow T_2 \wedge$ $\ldots$ $f_n \in A \rightarrow T_n$ CONCRETE_VARIABLES TA INVARIANT $TA \in A \rightarrow T_1 \times T_2 \times \ldots \times T_n \wedge$ $TA = f_1 \otimes f_2 \otimes \ldots \otimes f_n$ END

Association Table Structure
RULE R6Ass_D IS ABSTRACT_VARIABLES $f_1, f_2, \ldots, f_n, f$ CONSTRAINT $f \in C \rightarrow D \wedge$ $(f_1 \parallel f_2) \in f \rightarrowtail T_1 \times T_2 \wedge$ $f_1 = (dom(f)) \lhd v^C_{key} \wedge$ $f_2 = (ran(f)) \lhd v^D_{key} \wedge$ $\ldots$ $f_n \in f \rightarrow T_n$ CONCRETE_VARIABLES Tf INVARIANT $Tf \in f \rightarrow T_1 \times T_2 \times \ldots \times T_n \wedge$ $Tf = (f_1 \parallel f_2) \otimes f_3 \otimes \ldots \otimes f_n$ END

Hereafter are the corresponding substitution rules:

Add Substitution
RULE R6Clas_A IS
REFINE
$b_1 \in T_1 - ran(f_1) \mid$
$A := A \cup \{a\} \parallel (\parallel_{i=1..n} \ f_i := f_i \cup \{a \mapsto b_i\})$
REFINEMENT
$b_1 \in T_1 - dom^{(n-1)}(ran(TA)) \mid A := A \cup \{a\} \parallel$
$TA := TA \cup \{a \mapsto (b_1, b_2, \ldots, b_n)\}$
END

Remove Substitution
RULE R6Clas_R IS
REFINE
$A := A - \{a\} \parallel$
$\parallel_{i=1..n} \ f_i := \{a\} \lhd f_i$
REFINEMENT
$A := A - \{a\} \parallel$
$TA := \{a\} \lhd TA$
END

where $dom^n(f)$ is an abbreviation of $\overbrace{dom((\ldots(dom(f))))}^{n \ times}$.
In the Add rules, the precondition has to be refined because it depends on f_1 which is removed in the refined component.
Applying these rules to the functions *numor*, *date* and own_1 gives:
$Torder \in order \rightarrow NAT \times NAT \times NAT \wedge Torder = numor \otimes date \otimes own_1$
Add_Order is refined by the substitution:
$ord_1 \in NAT - dom(dom(ran(Torder))) \mid order := order \cup \{ord_1\} \parallel$
$Torder := Torder \cup \{ord_1 \mapsto (numor_1, date_1, numcus(cus_1))\}$
and Rem_Order is refined by :
$order := order - \{ord_1\} \parallel \ Torder := \{ord_1\} \lhd Torder$

Similar substitution rules exist for associations.

6 Mapping into SQL Tables

By the previous refinement rules we obtained a concrete model which is a formal description of the relational model and such that all its components can be automatically translated into SQL. This ensures the consistency of the relational implementation with the concrete model and thus the abstract model. The automatic mapping is performed as follows.

6.1 Description of the Mapping

T1. Definition of Tables. Each variable TA or Tf which is the source set of a direct product P is mapped into a SQL table (with the same name).

T2. Definition of Table Attributes. For a table TA or Tf, each attribute corresponds to a function f_i that occurs in the definition of P. Its domain is defined by the target set of f_i. If f_i corresponds to a total function of the initial specification then a NOT NULL constraint is defined on the corresponding attribute. The type of the SQL attribute depends on the type of the attribute in the abstract model. In general, for one type of B correspond several SQL types. Thus the designer has to choose the most appropriate type (especially for the B String type which must be bounded in SQL).

These two steps allow the structure of *Order*, *Customer*, *Product* and *Concerns* tables to be derived.

T3. Definition of Keys. The primary key of a table TA is the mapping from the primary key defined in Rule R2_D.

For a table Tf derived from the rule R6Ass_D, the primary key is composed of the attributes corresponding to the functions of the parallel product. For example, $(numor_1, numpr_1)$ is the primary key of *Concerns* table.

Other injective functions defined on TA are translated by a UNIQUE constraint on the corresponding attributes.

T4. Definition of Referential Constraints. Referential constraints "replace" the associations of the UML class diagram. Let us suppose that f is an association from C to D, TC and TD are the corresponding tables. If f is a monovalued association, R3_D has refined it by the concrete variable $(f_1 \in v_C \nrightarrow T^D_{key})$. For the attribute of TC corresponding to f_1 a referential constraint towards the primary key of TD is added. For example, a referential constraint from own_1 to the key *numcus* of *Customer* is defined.

In other cases, f is refined by R4_D. For the attributes of Tf corresponding to the variables occurring in the parallel product, two referential constraints towards the primary keys of TC and TD are added. For example, two referential constraints are added from $numor_1$ and $numpr_1$ respectively to the key *numor* of *Order* and the key *numpr* of *Product*.

T5. Mapping of Given Sets. The mapping of a given set depends on its type:
– an enumerate set is used as a target set in the typing of a function f_i. Thus, it is translated by a SQL domain definition which becomes the type of the corresponding attribute.
– an abstract given set is used in the typing of a class variable. If no property is specified on the given set, it is just ignored in the mapping phase. Otherwise the property is mapped by a constraint on the table corresponding to the class variable.

T6. Mapping of a Total Relation. A relation f defined from A to B, may be mandatory on A $(dom(f) = A)$ or B $(ran(f) = B)$. With Rule 4_D, such a relation is refined by: $v_1 = (dom(f)) \lhd v^A_{key} \wedge v_2 = (ran(f)) \lhd v^B_{key}$.
- $dom(f) = A$ means that $v_1 = v^A_{key}$ which is translated by the SQL constraints:
In Tf : Check(v_1 IN (SELECT v^A_{key} FROM TA))
In TA : Check(v^A_{key} IN (SELECT v_1 FROM Tf))
- $ran(f) = B$ means that $v_2 = v^B_{key}$ which is translated by the SQL constraints:
In Tf : Check(v_2 IN (SELECT v^B_{key} FROM TB))
In TB : Check(v^B_{key} IN (SELECT v_2 FROM Tf))

Applying this rule to concerns generates the constraint:
In Concerns : Check($numor_1$ IN (SELECT *numor* FROM *Order*))
In Order : Check(*numor* IN (SELECT $numor_1$ FROM *Concerns*))

Remark that other properties can be defined in a UML class diagram. They are mapped into additional conjuncts in the Invariant clause of the corresponding

B abstract model. Their translation into SQL integrity constraints is an ongoing work.

T7. Generation of SQL Statements:

Each basic operation is mapped into SQL basic operations which are composed of SQL statements. An Add operation is mapped into one or more insert statements, whereas a Remove operation is mapped into one or more delete statements. Parameters of the operation whose type is a reference to an object must be replaced by parameters whose type is a reference to a key. For example **Add_Order(ord$_1$, numor$_1$, date$_1$, cus$_1$, pd$_1$, quantity$_1$)** defined in Section 3.1 is translated by :
Add_Order(numor$_1$:Int,date$_1$:Int,numcus$_1$:Int,numpr$_1$: Int,quantity$_1$:Int).
Its body becomes:
INSERT INTO Order **VALUES** (numor$_1$, date$_1$, numcus$_1$)
INSERT INTO Concerns **VALUES** (numor$_1$,numpr$_1$,date$_1$, numcus$_1$)
And **Rem_Order(ord$_1$)** is translated by :
Rem_Order(numor$_1$:Int)
DELETE FROM Order **WHERE** numor=numor$_1$
DELETE FROM Concerns **WHERE** numor=numor$_1$
Let us remark that in the last example, CASCADE DELETE could be used.

6.2 Discussion

Preconditions on basic operations used to type the input parameters are directly translated into SQL basic operation signatures. Other preconditions are ignored in the mapping. In fact they indicate which tests must be performed in the calling operations to ensure a correct execution of the SQL basic operations.

In a relational implementation, tables must be at least in third normal form to guarantee a consistent behavior of the database. In a classical algorithm as described in Section 3.2, the normalization process is performed at the last step. It can result in the modification of tables and the definition of new ones which in turn requires modification of operations. In our approach, both the database structure and the operations are derived from the conceptual model, thus cannot be modified during the last step of the process. The solution to guarantee the third normal form property is to build a class diagram correct with respect to functional dependencies. Definition of the second and third normal forms needs to be revisited with an object-oriented point of view.

7 Proof of the Refinement Rules

When a refinement process is used in the development of an application, it is necessary to prove each step of this refinement. This ensures the correctness of the final result. In our case, to establish the correctness of the relational implementation, we have to prove each refinement rule presented above.

It is important to remark that the proof of each rule depends only on the rule itself. In other words, the proof of a given rule is completely independent of the application we refine. This independence enables not only the automation of the proof process, but also the reusability of each basic proof rule. Indeed, whatever the application refined, any refinement rule always gives rise to the same proof obligation. Thus each proof is achieved in a generic way and will be reusable(instantiated) in any specific refinement. All the proofs of the refinement rules have been achieved using AtelierB [8] and can be found in [15]. For lack of space, we give just the proof of Rule R4_A:

Using Definition 2 (section 2.1), the refinement is correct iff these three conditions are satisfiable:
P1. $\exists u,\ v.\ v = u; v^D_{key}$
P2. $\forall C, f, f_1.\ ((\mathrm{Trm}(c \notin C | f := f \cup \{c \mapsto d\}) \wedge f_1 = f; v^B_{key}) \rightarrow$
$\mathrm{Trm}(c \notin C | f_1 := f_1 \cup \{c \mapsto v^D_{key}(d)\}))$
P3.$\forall\ C, C', f, f_1, f'_1.\ ((\mathrm{Trm}(c \notin C | f := f \cup \{c \mapsto d\}) \wedge f_1 = f; v^D_{key} \wedge$
$\mathrm{Pred}(c \notin C | f_1 := f_1 \cup \{c \mapsto v^D_{key}(d)\})) \rightarrow$
$\exists f'.\ (\mathrm{Pred}(c \notin C | f := f \cup \{c \mapsto d\}) \wedge f'_1 = f'; v^D_{key}))$
let us prove each goal:
Proof of P1: $\exists u, v.\ v = u; v^B_{key}$ is satisfiable by taking u=$\emptyset$ and v=$\emptyset$.
Proof of P2:
(i) Trm(c$\notin$C | f:=f$\cup$ {c$\mapsto$d})=(c$\notin$C)$\wedge$ Trm(f:=f$\cup${c$\mapsto$d})=(c$\notin$C)
(ii) Trm(c$\notin$C|f$_1$:=f$_1\cup${c$\mapsto$ v $^D_{key}$(d)})= (c$\notin$C)| Trm(f$_1$:=f$_1\cup${a$\mapsto$v$^D_{key}$(d)})= (c$\notin$C)
So, P2 is equivalent to: $\forall$ C, f, f$_1$. (c$\notin$C $\wedge$ f$_1$=f; v$^D_{key}$)$\rightarrow$ c$\notin$C
which is a tautology.
Proof of P3:
Prd(c$\notin$ C| f$_1$:=f$_1\cup${c$\mapsto$ v$^D_{key}$(d)})= (c$\notin$ C) $\rightarrow$ Prd(f$_1$:=f$_1\cup${c$\mapsto$ v$^D_{key}$(d)})
=(c$\notin$C) $\rightarrow$ (f'$_1$=f$_1\cup$ {c$\mapsto$ v$^D_{key}$(d)})
Prd(c$\notin$C|f:=f $\cup${c$\mapsto$d})=(c$\notin$ C)$\rightarrow$ Prd(f:=f $\cup${c$\mapsto$d})
=(c$\notin$C) $\rightarrow$(f'=f $\cup${c$\mapsto$d})
By using (i), P3 is rewritten as:
$\forall$ C, C', f, f$_1$, f'$_1$. (c$\notin$C | f$_1$=f; v$^D_{key}$|((c$\notin$ C)$\rightarrow$ (f'$_1$=f$_1\cup${c$\mapsto$v$^D_{key}$(d)})) $\rightarrow$
$\exists$ f. (((c$\notin$ C) $\rightarrow$ (f'=f $\cup${c$\mapsto$}))|f'$_1$=f';v$^D_{key}$))
Applying the deduction theorem gives the following hypotheses:
(H1) c$\notin$C, (H2) f$_1$=f; v$^D_{key}$, (H3)((c$\notin$ C) $\rightarrow$(f$_1$=f$_1\cup${c$\mapsto$v$^D_{key}$(d)})
The current goal is: (G$_1$) f'.(((c$\notin$ C) $\rightarrow$(f'=f $\cup${c$\mapsto$d}))$\wedge$f'$_1$=f';v$^D_{key}$)
(H1)+(H3)+(Modus Ponens) gives: (H4) (f'$_1$=f$_1\cup${c$\mapsto$ v$^D_{key}$(d)})
let us assume:(H$_5$) (f'=f $\cup${c$\mapsto$})) in (G$_1$):
(G$_2$) (((c$\notin$C)$\rightarrow$(f $\cup$ {c$\mapsto$d}=f $\cup${c$\mapsto$d}))$\wedge$f'$_1$=f';v$^D_{key}$)
(H1)+(G2)+(Modus Ponens) gives: (G$_3$) (f'$_1$=f';v$^D_{key}$)

This last goal is checked as follows:
f'; v$^D_{key}$=(f $\cup${c$\mapsto$d});v$^D_{key}$ (rewriting of f ' using H5)
=(f;v$^D_{key}$) $\cup$({c$\mapsto$d};v$^D_{key}$) (distributivity of composition over union)
=f$_1\cup${c$\mapsto$v$^D_{key}$(d)} (H$_2$ +v$^D_{key}$ is an injection)
=f'$_1$ (rewriting using H$_4$) . QED

All the other proofs are similar. They are rather simple and then could be easily automated. Table 1 gives a synthesis of the different refinement proofs automatically generated by the AtelierB prover. Most of them were automatically proven. The exceptions of the automatic prover are not due to the specific application domain we consider but essentially to the lack of rules concerning properties of some relational operators and also the lack of proof tactics. For example in Rule 6, the prover fails to establish that a direct product of partial functions is also a partial function. Thus, in order to automate the refinement process, the rule base of the prover needs to be enriched.

8 Conclusions and Future Works

In this paper, we have described the B refinement process for transforming an abstract specification (obtained from an UML diagram) into a concrete implementation

Table 1. Proof result

		(Add Operation, Remove Operation)			
RULES		Obv	nPO	nUn	Pr
Rule1	Bounded Case	(7, 2)	(0,3)	(3,1)	75%
	Unbounded case	(6, 7)	(4, 4)	(1, 0)	95%
Rule 2		(6, 2)	(3, 5)	(0, 0)	100%
Rule 3		(7, 6)	(2, 10)	(0, 0)	100%
Rule 4		(11, 8)	(2, 10)	(5, 8)	70%
Rule 5		(6,2)	(3,3)	(2,1)	82%
Rule 6		(9, 2)	(2, 1)	(1, 2)	82%

which can be directly mapped into a relational database implementation. The process is specified by refinement rules that take into account both data and operations. Its correctness is achieved by the elaboration of the proofs related to each rule. The generic characteristic of the proofs enables the automation of both the refinement process and the proofs themselves. Thus, for a specific application, the designer can use a base composed of these generic proofs that have just to be instantiated. This genericity is limited to the domain of relational database applications.
The most important benefits of our approach are :
– reducing the development cost of the B refinement process;
– allowing the designer to concentrate on the specification phase, thanks to the automation of the refinement process;
– extending the application domain of the B method, up to now only used for critical systems design. This requires development of tools specific to a domain (such as relational databases) that guide designers in the use of the B method
– providing database designers with a method which ensures the complete consistency of applications (both data and programs).
– producing code which is standardized and thus easier to understand and maintain.

We are currently working on the development of a tool in order to automate the complete refinement process. This will operate in two ways: automatic and interactive. Designers might intervene to express choices concerning, for example, the introduction of a new key. The tool will extend an existing tool that translates UML diagrams into a B abstract specification. We are also working on the reusability of each generic proof rule and the complete automation of the proof phase.

Another work consists of refining more elaborate transactions. These are specified using UML state and collaboration diagrams which are then translated in B.

References

[1] Abrial, J.R.: The B-Book, Cambridge University Press, 1996.

[2] Barros, R.: Deriving relational database programs from formal specifications, 2nd Int. Conf. FME'94, Springer-Verlag, LNCS 873, Barcelona, Spain, Oct. 94.

[3] Batini, Ceri, Navathe: Conceptual Database Design: an EntityRelationship Approach, The Benjamin/Cummings Publishing Company, 1992.

[4] Burdy, L., Meynadier, J-M.: Automatic Refinement, BUG Meeting, FM'99, Toulouse, France, September 1999.

[5] Castelli,D., Pisani,S.: A Transformational Approach to Correct Schema Refinement, Conceptual Modeling-ER'98. 17th International Conference on Conceptual Modeling. Singapor, Novembre 1998.
[6] Codd, E.F.: A Relational Model of Data for Large Shared Data Banks. Communications of the ACM, V13, N°6, June 1970, pp.377-387.
[7] Date, C.J.: An Introduction to Database Systems, Addison-Wesley, 6th edition, 1996.
[8] Digilog groupe STERIA: Atelier B–Manuel de référence, 1996, DIGILOG, BP 16000, 13791 Aix-en-Provence Cedex 3 France.
[9] Donzeau-gouge, V., Simonot, M.: Conception Rigoureuse de Programmes, coursebook of the DESS "Développement de Logiciels Sûrs", CNAM, Paris, 1999.
[10] Evans, A.S.: Reasoning with UML Class Diagrams, Workshop on Industrial Strength Formal Methods, WIFT'98, Florida, IEEE Press, 1998.
[11] Facon, P., Laleau R., Nguyen, H. P.: Mapping Object Diagrams into B Specifications, Methods Integration Workshop, Leeds, UK, March 1996.
[12] Facon, P., Laleau, R., Mammar, A.: Combining UML with the B Formal Method for the Specification of Database Applications, Research report, CEDRIC laboratory, CNAM, Paris, September 1999.
[13] Günther, T., Schewe, K.D., Wetzel, I.: On the Derivation of Executable Database Programs from Formal Specifications. Int. Symp. FME'93, Odense, Denmark, April 1993.
[14] Mammar, A., Laleau, R.: Using a Formal Refinement to Derive Relational Database Implementations from B Specifications, Research report, CEDRIC laboratory, CNAM, Paris, January 2000.
[15] Matthews, B., Locuratolo, E.: Formal Development of Databases in ASSO and B. FME'99 Word Congress on Formal Methods, Springer-Verlag, LNCS 1709, Toulouse, France, Sept.99.
[16] Melton,J., Simon, A.: Understanding the new SQL: A Complete Guide. Morgan Kaufmann Publishers, 1993.
[17] Nguyen, H.P.: Dérivation de spécifications formelles B à partir de spécifications semi-formelles, PHD thesis, CEDRIC laboratory, Paris, France, December 98.
[18] OMG: The UML Group: Unified Modeling Language, version 1.1, Rational Software Corporation, www.rational.com/uml, Santa Clara, USA, July 1997.