Autom Software Eng (2006) 13:497–528
DOI 10.1007/s10515-006-0273-5

A formal approach based on UML and B for the specification and development of database applications

Amel Mammar · Régine Laleau

Abstract This article describes a formal approach to specify and develop database applications. This approach consists of two complementary phases. In the first phase, B specifications are automatically generated from UML class, state and collaboration diagrams describing the data and the transactions of the system we are developing. In the second phase, these specifications are successively refined until they become close enough to a relational implementation. The tool supporting this approach is implemented as an extension of the Rational Rose tool to develop and visualize graphical (UML) and formal (B) notations in a single environment.

Keywords Formal methods · Integration · UML · B · Database applications · Relational implementation

1 Introduction

Database systems are the core component of various applications: e-business, financial systems, smart cards, etc. Database applications are characterized by great volumes of structured data that are subject to numerous rules of consistency called *integrity constraints*. A trustworthy database application must enforce the integrity constraints specified on its data. Generally a transaction has a simple algorithmic structure and is made up of a set of basic operations which are generic and shared by all database applications. The main difficulty when specifying a transaction is to

A. Mammar (✉)
University of Luxembourg, SE2C, 6 rue Richard Courdenhove-Kalergi, L-1359
Luxembourg-Kirchberg, Luxembourg
e-mail: amel.mammar@clearsy.com

R. Laleau
University of Paris 12, LACL, IUT Fontainebleau, Route Forestière Hurtault, 77300 Fontainebleau
e-mail: laleau@univ-paris12.fr

guarantee that it preserves the global integrity constraints of the system. To address this issue, we suggest employing formal methods.

The derivation of database applications from formal specifications is a well known but incompletely solved problem. Most of the already developed work and case tools were restricted to the derivation of the database schema by considering that the production of transactions can be straightforwardly achieved (Barros, 1994a,b, 1998; Edmond, 1995; Gunther et al., 1993). However, the specification of transactions that take integrity constraints into account raises non trivial difficulties and thus requires the use of formal methods to gain confidence about the correctness of the transactions.

For a few years, our team has been engaged in a programme of research addressing the development of trustworthy database applications using the UML modeling language and the B formal method. UML is the standard object-oriented modeling language accepted by the OMG (Object Management Group) (OMG, 2005). It is supported by several commercial tools like ROSE (Rational, 2003), XDE (Rational, 2005), MagicDraw NoMagic (2005), etc. Unfortunately, none of these tools provides possibilities for formally generating database transactions. This is mainly due to the lack of a precise semantics for the UML dynamic diagrams. Contrary to UML, the B method (Abrial, 1996) is a safe technique widely used for the development of critical systems. The B method deals with the central aspects of the software life cycle, namely: abstract specification, design by successive refinement steps, and executable code generation. Using this method, software engineers are able to check that the system fulfills the expected properties, and/or find a large range of errors at different development phases.

The approach that we have developed consists of two complementary phases: 1. generating a B specification from a UML description of an application, and 2. operating successive transformations on the B specification in order to obtain a concrete specification that can be straightforwardly translated into an application in JAVA/SQL, which is the implementation language we have chosen for describing database transactions. Among the diagrams provided by UML, we use class diagrams for describing the static aspect of a system, and state/collaboration diagrams for the dynamic aspects. This subset of UML diagrams has been endowed with a formal semantics and is called *IS-UML* in the remainder of the paper. To make our approach more accessible for software engineers, we have built a tool, called *UB2SQL* (Mammar and Laleau, 2005), that automates both the generation of B specifications from *IS-UML* and their refinement until a relational implementation is obtained. *UB2SQL* enables users to develop trustworthy database applications using two complementary notations: UML graphical notations and B formal notations. In this way, software engineers take advantage not only of the visual and intuitive aspects of UML but also of the rigorous reasoning aspects of B. In this paper, we give a general overview of the approach, through a single example, by illustrating all the consecutive steps from its *IS-UML* specification down to the JAVA/SQL implementation. The appropriate research papers that describe more precisely specific points are referenced throughout the paper.

The paper is organized as follows. Section 2 starts with a brief description of the B method and the relational database model. Section 3 gives an overview of our approach for the development of database applications. The different steps constituting our approach are described, through a running example, in Sections 4, 5 and 6. A

discussion of the benefits of the proposed approach and a comparison with other similar approaches is given in Section 7. Finally, Section 8 concludes and outlines some future work.

2 The B method and the relational database model

This section briefly presents the main concepts of B and the relational model to help the reader understand our approach.

2.1 The B method

Introduced by Abrial (1996), B is a formal method dedicated to the development of safe systems. B specifications are organized in abstract machines. Each machine encapsulates state variables on which operations act, and an invariant constraining the state variables. The invariant is a predicate in a simplified version of ZF-set theory, enriched with relational operators. It allows us to type the variables and to specify some constraints that must always be verified by the state of the system. Operations are specified in the Generalized Substitution Language which is a generalization of Dijkstra's guarded command notation. A substitution is like an assignment statement. It allows us to identify which variables are modified by the operation, while avoiding mentioning those which are not. The generalization allows the definition of non-deterministic and pre-conditioned substitutions. A pre-conditioned substitution is of the form **PRE** P **THEN** S **END** where P is a predicate, and S a substitution. When P holds, the substitution is executed, otherwise nothing can be ensured: for instance, the substitution S might not terminate or might violate the invariant. A brief description of the B substitutions used in this paper is provided in the appendix. The B method is supported by commercial tools like AtelierB (Clearsy, 2003), BToolkit (B-CORE, 1996), and recently by a free tool called *B4free and Clik'n'prove* (Clearsy, 2004). B Refinement is the process of transforming one specification into a less abstract one. A refinement can operate on an abstract machine or another refinement component. In B, we distinguish two kinds of refinement:

- Behavioral refinement: this refinement aims at eliminating non-determinism and coming close to the control structures used in the chosen target programming language. Behavioral refinement includes, for example, weakening of preconditions, replacement of a parallel substitution by a sequence substitution, etc.
- Data structure refinement: an abstract data structure D is replaced by a concrete data structure D' that must be close to the data structure allowed in the target implementation language. For example a set, which is not available in standard programming languages, is often refined by an array available in most programming languages. In this kind of refinement, a predicate J, called the gluing invariant, must be defined. It states the existing relation between D and D'.

Both specification and refinement steps give rise to proof obligations. At the abstract level, proof obligations ensure that each operation maintains the invariant of the system, whereas at the refinement level, they ensure that the transformation preserves the properties of the abstract level. To carry out these proofs, AtelierB (Clearsy, 2003)

includes two complementary provers. The first one is automatic, implementing a decision procedure that uses a set of deduction and rewriting rules. The second prover allows the users to enter into a dialogue with the automatic prover by defining their own deduction and/or rewriting rules that guide the prover to find the right way to discharge the proofs. To ensure the correctness of the interactive proofs, each added rule must be proved by the prover of the AtelierB before it can be used.

2.2 Overview of the relational database model

The relational model was defined by Codd (1970). Relations, which we call tables according to SQL (Melton and Simon, 1993) terminology to avoid confusion with the B concept, are specified as sets of tuples (SQL bags without redundant tuples). A tuple is an element of a cartesian product. The formal definition of a table contains two parts: the table intension and the table extension. The intension is a tuple type that defines the table attributes, each of which must be of a valid type. The extension is the set of the instances of the type tuple existing at a given moment. A full description of the relational model can be found in Elmasri and Navathe (2003). The integrity constraints that can be defined on tables are as follows:

- NOT NULL constraint: defined on an attribute (or a group of attributes), this constraint specifies that the attribute must be valued.
- UNIQUE constraint: defined on an attribute (or a group of attributes), this constraint specifies that: if the attribute is valued, its value is unique in the table extension. This means that such an attribute may be null.
- Key constraint: each table has at least one key. By definition, a table is a set of tuples and thus does not contain duplicate tuples. Thus, a table has at least the key composed of all its attributes. However, in most cases it is possible (and strongly recommended) to determine a key composed of a subset of all the attributes. If several keys are possible, one of them is defined as the PRIMARY KEY of the table. The other candidate keys are defined by using NOT NULL and UNIQUE constraints.
- Referential constraint: defined on an attribute (or a group of attributes), denoted Att_1, of a table T_1 towards a key of a table T_2, denoted K_2, it specifies that the set of values of Att_1 is included in the set of values of K_2.

The remainder of the paper describes a formal approach and its support tool that allows the generation of a relational implementation from a B specification describing a database application. The B specifications we consider are derived from well-defined UML diagrams. We assume that the reader is familiar with UML notations.

3 Overview of the approach

Our approach to develop database applications can be summarized by the following steps (see Fig. 1).

1. The data structure and the transactions are described using *IS-UML* diagrams. *IS-UML* notations are derived from UML 1.4 notations, dedicated to the

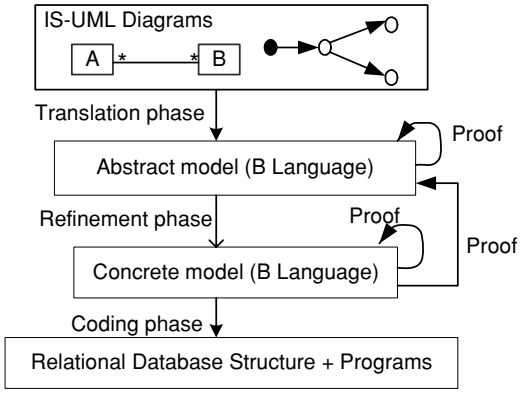

Fig. 1 From IS-UML diagrams to a relational implementation

specification and design of database applications. They are given a formal semantics (see Laleau and Polack, 2001a,b). Data structures are specified by class diagrams with a semantics of standard ER models. Class diagrams we consider include classes, attributes, associations, class associations and simple inheritances without cycles. A given class may also inherit from different classes that share the same superclass. Transactions are described by state and collaboration diagrams supplemented with precise annotations described in the B notations.

2. According to translation rules defined in Laleau (2000), Laleau and Mammar (2000b), B specifications are generated from the previous *IS-UML* diagrams. These specifications are structured into a three-level architecture that reflects the *IS-UML* specification structuring. This architecture gives a better readability to the B specifications and lightens the proof phase cost (see Sections 5.1.3 and 5.2.3).
3. The next step of our approach deals with the refinement of the generated B specifications. Applying the generic refinement rules described in detail in Laleau and Mammar (2000a) and Mammar (2002), the B specifications are successively transformed until they become close enough to a B formalization of a relational implementation. The last B specification, produced in the B implementation phase, reflects the architecture of the database which consists of both the database schema and the transactions.
4. From the last refinement step, the definition of the database schema is automatically derived (Mammar and Laleau, 2006; Mammar, 2002). We plan to extend this automation to derive the programs corresponding to the transactions.

Our approach has several benefits. The most important among them are the following:

– Having a precise matching between the abstract B specification and the *IS-UML* description, on the one hand, and between the concrete B specification and the relational database, on the other hand, allows a better understanding of the developed application. Moreover, the resulting *IS-UML* model is as formal as the B specification, but in a form which is more friendly and familiar to IS designers. The translation to textual B specification does not add anything to the *IS-UML* specification: it merely provides an alternative form and, above all, allows existing B tools to be used to reason about the specification.

– Allowing the designer to concentrate more on the analysis phase (elaboration of *IS-UML* diagrams) since the specification and design phases are largely automated thanks to the building of a tool called *UB2SQL*.
– Providing the database engineers with a method which ensures the complete consistency of applications: by construction, the generated programs still verify all constraints specified at the abstract level that are easier to verify at the design stage in order to discover possible inconsistencies.

The following subsections illustrate the different development steps through a simplified information system of a university. The system manages a set of teachers and students that are gathered into the concept of *Person*. Each of them, described by a name ($NamePers$), subscribes to an insurance company. Each teacher may intervene in one to several courses. Similarly, several teachers may be involved in the same course. Each course is described by the number of students (*FreePlaces*) that can be enrolled in the future. When a course is full ($FreePlaces = 0$), all the forthcoming enrollments are set to state "Wait" (see Fig. 2). We have chosen a simplified example in order to facilitate the presentation of our approach. Real-life sized systems may contain a great number of classes and associations, whereas transactions, most often, involve few classes and associations. We would obtain a larger B specification, which implies a great number of refinement proofs. However, the forms (schemas) of these refinement proofs remain the same.

4 Specification of the static aspect of an application

4.1 Elaboration of IS-UML class diagrams

The class diagram is the first one to be elaborated. Indeed, the other diagrams use not only information related to this diagram but also to its corresponding B specification. For example, we need to know the name of the operations derived for each class and

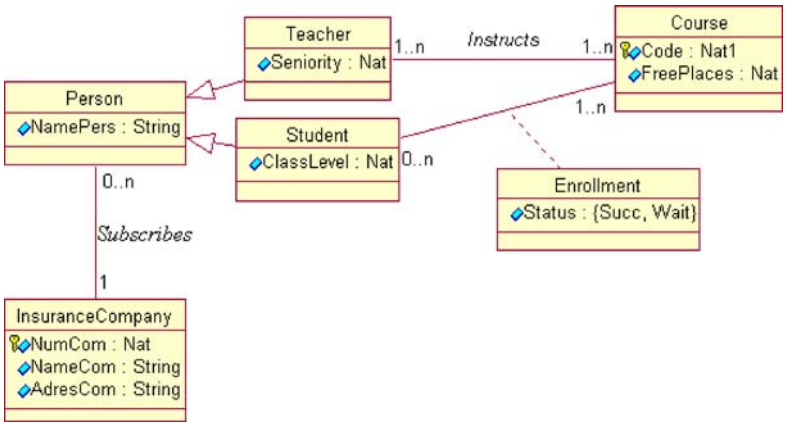

Fig. 2 An example of class diagram

association. Figure 2 shows the class diagram of the running example. To generate a complete B specification, users specify some additional information about:

- Attribute multiplicity: mandatory-monovalued (1..1), optional-monovalued (0..1), optional-multivalued (0..n), mandatory-multivalued (1..n). Only these common multiplicities are considered. The extension to arbitrary multiplicities doesn't raise any difficulty.
- Attribute mutability: the value of an attribute may be modified or not.
- The frozen characteristic of each association end: an association with a frozen association end doesn't have its own operations. For instance, the association end of *Subscribes* related to *InsuranceCompany* is frozen since each link of this association is created (resp. deleted) when a new person is created (resp. deleted) and doesn't change (in general) during the person's life.

4.2 B translation of IS-UML class diagrams

To translate a class diagram into a B specification, the formal rules described in detail in Laleau (2000) and Laleau and Mammar (2000b) are applied. The main rules are summed up as follows:

- With each class A (resp. no frozen association As) a B machine M_A (M_{As}) is associated.
- For each class A which has no superclass, an abstract set S_A of all possible instances is declared.
- For each class A, we define a variable v_A representing the set of existing instances.
- Each mandatory-monovalued attribute is modeled as a function ($\rightarrow$) from the variable v_A to the type of the attribute. A key is translated into a total injective function ($\rightarrowtail$). For each UML type which does not match a B type, an abstract B set is created.
- If a class A inherits directly from an other class B, then machine M_A uses machine M_B in order to have access to its data. The variable v_A, associated with A, must be a subset of v_B.
- A UML association, involving two classes C and D, is translated into a B relation ($\leftrightarrow$) between the two sets v_C and v_D. This relation becomes a function, an injection, etc. depending on the multiplicity of the roles (association ends).
- When an association is frozen, its specification is included in the B machine translating the class related to the frozen association end.
- For each inheritance structure involving a set of classes S, a B machine M_S that includes all the machines associated with classes S is defined. Machine M_S gathers all information related to the different classes involved in the related inheritance. Moreover, it is in this machine where constraints involving different classes S are defined.

For instance, three B machines *Basic_Person*, *Basic_Student* and *Basic_Teacher* are defined for the classes involved in the inheritance structure of Fig. 2. These machines contain the following B specifications:

- Machine *Basic_Person*

  Sets *PERSON*
  Variables *Person*, *NamePers*
  Invariant $Person \subseteq PERSON \wedge NamePers \in Person \rightarrow STRING$

- Machine *Basic_Student*

  USES *Basic_Person*
  Variables *Student*, *ClassLevel*
  Invariant $Student \subseteq Person \wedge ClassLevel \in Student \rightarrow NAT$

- Machine *Basic_Teacher*

  USES *Basic_Person*
  Variables *Teacher*, *Seniority*
  Invariant $Teacher \subseteq Person \wedge Seniority \in Teacher \rightarrow NAT$

  In addition to these machines, a fourth machine *Person* which includes the previous three ones is defined. This machine defines the basic operations related to the subclasses involved in the considered inheritance structure. Moreover, it is in this machine where we can state, for instance, that classes *Teacher* and *Student* are exclusive: $Teacher \cap Student = \emptyset$.

Similarly, the association class *Enrollment* is translated into a B machine *Basic_Enrollment* whose static part is as follows (*Course* is the variable translating class *Course*):

Sets $Type\_Status = \{Succ, Wait\}$
USES *Person*, *Basic_Course*
Variables *Enrollment*, *Status*
Invariant
$Enrollment \in Student \leftrightarrow Course \wedge$
//translation of multiplicity 1..*n*
$dom(Enrollment) = Student \wedge$
$Status \in Enrollment \rightarrow Type\_Status$

- A set of basic operations is automatically derived for each class and association. These operations include insertion, deletion of objects and the update of the value of each mutable attribute. For instance, the operation which adds a new link to association *Enrollment* is specified as follows:

  BasicAddEnrollment(stu, cou, sta) $\triangleq$
  PRE $stu \in PERSON \wedge cou \in COURSE \wedge sta \in Type\_Status \wedge$
  $stu \mapsto cou \notin Enrollment$
  THEN
  $Enrollment := Enrollment \cup \{stu \mapsto cou\} \parallel$
  $Status := Status \cup \{stu \mapsto cou \mapsto sta\}$
  END

  This operation takes as parameters the two instances to be linked and a value of each mandatory attribute of this association. The precondition of this operation consists in specifying the type of each input parameter and checking that the enrollment

we would like to create does not exist yet ($stu \mapsto cou \notin Enrollment$). Under this precondition, the operation updates the set of the existing links and the values of the attributes. Recall that all the parts of this operation are automatically generated from the class diagram. Let us remark that the precondition of the previous operation refer to sets *PERSON* and *COURSE*, representing the set of all possible instances, instead of variables *Student* and *Course* that represent the existing instances. Such a relaxed typing permits to specify for instance an operation that enrolls, in a course, a new student at the same time that this latter is created. Moreover, such typing doesn't violate the invariant related to the variable *Enrollment*. In fact, as this invariant refers to variables defined in the used machines (*Person* and *Basic_Course*), the B prover will check this invariant in all other operations that call *BasicAddEnrollment*.

To illustrate our approach, we also consider the following operations:

- *BasicAddPerson*: this operation adds a new person. It is specified in the machine *BasicPerson*.

  BasicAddPerson(pr, na, ic)$\stackrel{\triangle}{=}$
  PRE $pr \in PERSON - Person \wedge na \in STRING \wedge ic \in InsuranceCompany$
  THEN
  $Person := Person \cup \{pr\} \|$
  $NamePers := NamePers \cup \{pr \mapsto na\} \|$
  $Subscribes := Subscribes \cup \{pr \mapsto ic\}$
  END

- *BasicAddStudent*: this operation adds a new student without considering him as a person. It is specified in the machine *BasicStudent*.

  BasicAddStudent(st, cl)$\stackrel{\triangle}{=}$
  PRE $st \in PERSON - Person \wedge cl \in NAT$
  THEN
  $Student := Student \cup \{st\} \|$
  $ClassLevel := ClassLevel \cup \{st \mapsto cl\}$
  END

- *AddStudent*: this operation actually adds a new student by adding him as a new person. Hence, it calls the previous two operations *BasicAddPerson* and *BasicAddStudent*. It is specified in the machine *Person*.

  AddStudent(st, na, cl, ic)$\stackrel{\triangle}{=}$
  PRE $st \in PERSON - Person \wedge na \in STRING \wedge cl \in NAT \wedge$
  $ic \in InsuranceCompany$
  THEN
  //adding a new instance to *Person*
  $BasicAddPerson(st, na, ic) \|$

  //adding a new instance to *Student*
  $BasicAddStudent(st, cl)$
  END

- *BasicUpdateFreePlaces*: this operation is specified in the machine *Course*.

BasicUpdateFreePlaces(co, fp)$\stackrel{\triangle}{=}$
PRE $co \in Course \wedge fp \in NAT$
THEN
$FreePlaces(co) := fp$
END

Let us remark, since the association end of *Subscribes* is frozen, at the creation of each new instance of *Person*, this latter is linked to an instance of *InsuranceCompany*. For that, association *Subscribes* is defined in the same B machine corresponding to *Person* that uses the machine related to *InsuranceCompany*.

4.3 Correctness of basic machines

In order to ensure the correctness of the B basic machines generated from the *IS-UML* class diagrams, a set of proof obligations is automatically calculated by the proof obligations generator (GOP) of AtelierB. From a theoretical point of view, one proof obligation is generated for each B operation. This proof verifies that the execution of the operation maintains the invariant. In order to facilitate the establishment of these proofs, GOP operates simplification rules that lead to a great number of proofs which are easier to achieve than the initial sub-proof. For instance, to prove the correctness of the operation *BasicAddEnrollment*, 4 sub-proof obligations are defined. Note that all the proofs related to the B machine translating classes are automatically discharged by the automatic prover of AtelierB. However, proving the correctness of operations that delete links from associations requires defining new deduction rules in the B prover. More details can be found in Mammar (2002).

5 Specification of the behavioural aspect of an application

Once the class diagram is translated into a B specification, transactions (behavioral aspects) of the application are described by using state and collaboration diagrams. The aim of these diagrams is to specify the effect of transactions on the objects of the system. It is important to note that the term transaction, used in databases, has several meanings. From a system point of view, it refers to the three-phase processing of acquiring locks on some resources, performing a sequence of operations and releasing the locks. From a functional point of view, it means an interrupted execution of a sequence of operations that corresponds to the achievement of a functionality of the information system. In this paper we adopt the functional point of view. The description of state and collaboration diagrams and their B translation follows.

5.1 IS-UML state machines (diagrams)

5.1.1 Elaboration of IS-UML state machines

Generally, a state machine describes the behaviour of objects of the system. The notations of these diagrams are similar to those of Harel (1987) for which we have defined a particular semantics dedicated to the database domain. In our case, a state

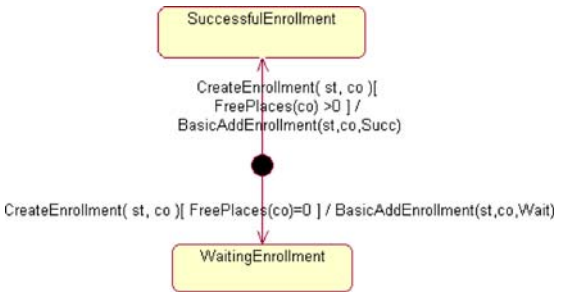

Fig. 3 The partial state diagram of association class *Enrollment*

diagram is associated with a single class or association class. It describes the state changes of a single object. These changes are triggered by events. Also, only basic states are considered for this purpose. The different states in which an instance of the related class may be in are described by predicates denoting properties that the instance must verify. A transition, triggered by an event, may be guarded. To ensure a deterministic behavior, we make the assumption that at the reception of an event, at most one transition is fireable. To avoid deadlock, we assume that at the reception of an event, at least one of these guards is satisfied. When a transition is fired, one basic operation of the related class or association class may be called. Figure 3 (partially) shows the state diagram of an instance of the association class *Enrollment*. We associate with this diagram the following semantics: when event *CreateEnrollment* occurs, depending on the number of free places of the considered course, the enrollment is either created as successful by calling the basic operation *BasicAddEnrollment* with the value "Succ", or in state "Wait" by calling the basic operation *BasicAddEnrollment* with the value "Wait". Each state of such a diagram is described by a B predicate. For instance, the *initial* and *WaitingEnrollment* states are described by the following two predicates: $(initial(st, co) \triangleq (st \mapsto co) \notin Enrollment)$ and $(WaitingEnrollment(st, co) \triangleq Status(st, co) = Wait)$ respectively. The definition of the initial state means that a given link, defined by a couple of objects (st, co), is in the initial state if and only if these objects satisfy the related predicate. Moreover, event *CreateEnrollment* happening on such a link will modify its state according to the conditions represented by guards.

5.1.2 B translation of IS-UML state machines

The translation into B consists in generating a B operation for each event. In other words, a single operation translates all the transitions triggered by the same event. The following B operation translates event *CreateEnrollment*:

CreateEnrollment(st, co) $\triangleq$
PRE $st \in PERSON \wedge co \in COURSE \wedge st \mapsto co \notin Enrollment \wedge$
$((initial(st, co) \wedge FreePlaces(co) > 0) \vee (initial(st, co) \wedge FreePlaces(co) = 0))$
THEN
SELECT $FreePlaces(co) > 0$ **THEN** $BasicAddEnrollment(st, co, Succ)$
WHEN $FreePlaces(co) = 0$ **THEN** $BasicAddEnrollment(st, co, Wait)$
END
END

Let us give some explanations about the way whereby each part of this operation is generated:

1. *Signature*: the signature of a B operation translating an event exactly corresponds to the signature of this event: same name and same parameters.
2. *Precondition:* the precondition includes several conjuncts. The last conjunct is derived from the transitions triggered by the related event. It is a disjunction of predicates, each of them is associated with a transition and is the conjunction of the predicate of the source state and the guard of the transition: $((initial(st, co) \wedge FreePlaces(co) > 0) \vee (initial(st, co) \wedge FreePlaces(co) = 0))$. The first two preconditions concern the typing of the parameters and the preconditions of the called operations: $(st \in PERSON \wedge co \in COURSE \wedge st \mapsto co \notin Enrollment)$. This predicate is automatically inferred from the way whereby these parameters are used in the basic operation calls. From a practical point of view, we proceed as follows. For each call $op(EParam_1, \ldots, EParam_n)$ to an operation *op* whose formal parameters are $FParam_1, \ldots, FParam_n$, we substitute, in the precondition of *op*, each formal parameter by its corresponding actual parameter. Then, the precondition of the operation translating the event denotes the conjunction of all these sub-preconditions. Doing this, some conjuncts may be redundant, consequently, simplification rules must be applied. An example showing how such preconditions are calculated is given in Section 5.2.2.
3. *Postcondition (Body)*: the effect of the event is translated by a *SELECT* substitution. Each branch of this substitution is related to a single transition. It is guarded by the guard of the transition, and executes the action specified for it.

5.1.3 Proofs for state diagrams

As for the B specification translating class diagrams, proof obligations are generated for the B specifications translating state diagrams. Since no additional invariants are added in this step, these proof obligations concern exclusively two points:

1. *Preconditions of the called operations:* for each operation *op* called in an operation *ev* whose precondition is $Prec_{ev}$, a proof obligation checking the precondition $Prec_{op}$ of *op* is generated. These proof obligations are automatically discharged since, as mentioned in the previous section, the precondition $Prec_{ev}$ is obtained by taking the precondition of each called operation into account.
2. *Target state of a transition:* let *A* and *B* be two states of a state diagram. Let Pr_A and Pr_B the predicates describing the states *A* and *B* respectively. For each transition *tr*, from *A* to *B*, for which a guard *gr* and an action *op* are specified, a proof obligation, ensuring that Pr_B is verified after executing *op*, is generated. For instance, the transition relating the *initial* and *WaitingEnrollment* states raises the following proof obligation:

$$(initial(st, co) \wedge FreePlaces(co) = 0) \Rightarrow (Status \cup \{st \mapsto co \mapsto Wait\})(st, co) = Wait$$

which is true. It is worth noting that no new proof obligations are generated for the invariants specified in the basic machines since we have already proved that each basic operation preserves its invariant when it is called under its precondition.

Since the operation translating a state diagram specifies just additional conditions for calling one of the basic operations, all the proofs are still valid.

5.2 IS-UML collaboration diagrams

5.2.1 Elaboration of IS-UML collaboration diagrams

To describe transactions that involve several classes and associations, we use collaboration diagrams. In our method, a collaboration diagram shows how the effect of a transaction is decomposed into internal messages sent to each class or association. Figure 4 shows the UML collaboration diagram associated with transaction *AddEnrollment* that actually registers a student into a course. This transaction consists in simultaneously executing the following actions:

- Triggering event *CreateEnrollment*.
- Decreasing the number of the free places of the related course if there is at least one available place.

5.2.2 B translation of IS-UML collaboration diagrams

A collaboration diagram is translated into a single B operation whose body consists in simultaneously calling the operations corresponding to the messages. The operation generated for the collaboration diagram of Fig. 4 is as follows.

AddEnrollment(st, co) $\stackrel{\triangle}{=}$
PRE $st \in Student \wedge co \in Course \wedge st \mapsto co \notin Enrollment \wedge$
$((initial(st, co) \wedge FreePlaces(co) > 0) \vee (initial(st, co) \wedge FreePlaces(co) = 0))$
THEN
$CreateEnrollment(st, co) \|$
IF $FreePlaces(co) > 0$ **THEN**
BasicUpdateFreePlaces(co,FreePlaces(co)-1)
END
END

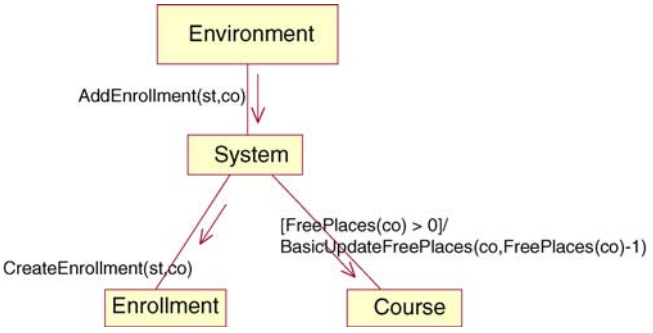

Fig. 4 The collaboration diagram of transaction *AddEnrollment*

It is important to note that, contrary to other generated operations, the operations generated from collaboration diagrams define the interface of our information system. In other words, the state of the system evolves through the execution of these operations. This is why the typing of the input parameters of these operations is not inferred only from operations calls but also from the substitutions that compose these operations. So, to generate the preconditions of these operations, we have to substitute the formal parameters, in the body of the called operations, with the actual ones. Taking operations' bodies into account permits, for instance, to resolve the relaxed typing used in the operations generated for associations. For instance, precondition P of operation *AddEnrollment* is generated as follows:

1. From the call to operation *BasicUpdateFreePlaces*, we deduce that:
$$P_1 \triangleq (co \in Course \wedge (FreePlaces(co) - 1) \in NAT)$$

2. From the call to operation $CreateEnrollment$, we deduce that:
$$P_2 \triangleq (st \in PERSON \wedge co \in COURSE \wedge st \mapsto co \notin Enrollment$$
$$\wedge ((initial(st, co) \wedge FreePlaces(co) > 0) \vee (initial(st, co) \wedge FreePlaces(co) = 0))))$$

The predicate P is then equal to:

$$P_1 \wedge P_2 \triangleq (co \in Course \wedge (FreePlaces(co) - 1) \in NAT$$
$$\wedge st \in PERSON \wedge co \in COURSE \wedge st \mapsto co \notin Enrollment$$
$$\wedge initial(st, co) \wedge FreePlaces(co) > 0) \vee (initial(st, co) \wedge FreePlaces(co) = 0))$$

On this predicate, simplification rules must be applied in order to optimize it. To do this, we have defined several simplification rules. For instance, to simplify the predicate P, the following simplification rules, proved by the prover of the AterlierB, must be applied:

1. $\frac{B \in NAT}{((A-B) \in NAT) == (A \in NAT)}$: this rule means that if the second operand of the minus operator is natural, and the result of the operation is natural, we can deduce that the first operand is natural too. Applying this rule rewrites $((FreePlaces(co) - 1) \in NAT)$ into $(FreePlaces(co) \in NAT)$.
2. $\frac{f \in A \rightarrow B}{(f(a) \in B) == (a \in A)}$: this rule means that expression $f(a)$ is defined if only and only if a belongs to the domain of f. Applying this rule rewrites $(FreePlaces(co) \in NAT)$ into $(co \in Course)$.
3. $\frac{C_2 \subseteq C_1 \quad \wedge not(c \in (C_1 - C_2))}{(c \in C_1) == (c \in C_2)}$: this rule means that if there is no constraint saying that an instance does not belong to a given subset C_2, then it is. Applying this rule rewrites $co \in COURSE$ into $co \in Course$. But, the same rule applied on $(st \in PERSON)$ gives three possible rewritings: $(st \in Person)$, $(st \in Teacher)$ and $(st \in Student)$. This is why we have to consider the body of operation *CreateEnrollment* too, which comes down to consider the body of *BasicAddEnrollment*. We obtain the following substitutions ("_" denotes either "Succ" or "Wait" value):

$$Enrollment := Enrollment \cup \{st \mapsto co\}$$
$$Status := Status \cup \{st \mapsto co \mapsto \_\}$$

Replacing the after-value of *Enrollment* in invariant ($Enrollment \in Student \leftrightarrow Course$) gives: ($Enrollment \cup \{st \mapsto co\} \in Student \leftrightarrow Course$). On this last formulas, we have just to apply the simplification rule 4.

4. $\frac{f \in A \leftrightarrow B}{(f \cup \{a \mapsto b\}) \in A \leftrightarrow B == (a \in A \wedge b \in B)}$: this rule means that the type of a relation f is maintained by adding links whose components are well-typed.
5. $\frac{c \in C_1 \ \wedge \ c \in C_2 \ \wedge (C_1 \subseteq C_2)}{(c \in C_1 \ \wedge \ c \in C_2) == (c \in C_1)}$: this rule means that we keep only the strongest typing.

Finally precondition P of operation *AddEnrollment* is equal to:

$$P \triangleq (st \in Student \wedge co \in Course \wedge st \mapsto co \notin Enrollment \wedge$$
$$((initial(st, co) \wedge FreePlaces(co) > 0) \vee (initial(st, co) \wedge FreePlaces(co) = 0))))$$

As we can see, this predicate can be further simplified. Indeed, the predicate $initial(vc)$ is redundant with the predicate ($st \mapsto co \notin Enrollment$). However, we keep it in order to maintain a correspondance between the generated B specification and the *IS-UML* diagrams.

5.2.3 Proofs for collaboration diagrams

Since no additional invariants are added in this step, for each operation *op* translating a collaboration diagram, two kinds of proof obligation are generated. The first kind concerns the called operations. One proof obligation is generated for each called operation. This proof obligation ensures that each operation is called under its precondition. For instance, we must verify that the precondition of the operation *BasicUpdateFreePlaces* is verified when it is called. These proof obligations are all automatically discharged since the precondition of the operation *AddEnrollment* was defined by taking the preconditions of all the called operations into account. The second kind of proof obligations concerns the parts of the invariant of the basic machines that depend on variables that are read and modified simultaneously by the called operations. Indeed, although we have proved that each basic operation, executed separately, establishes the invariant, the simultaneous execution of two (or more) basic operations may violate the invariant. These proof obligations must be achieved interactively. For our case study for instance, suppose we have defined in machine $BasicEnrollment$ the following B invariant:

$$INV \triangleq \forall(st, co).(st \mapsto co \in Enrollment \wedge FreePlaces(co) > 0$$
$$\Rightarrow Status(st, co) = Succ)$$

that states that all the enrollments in a given course *co* in which there are free places must be successful. Let *co* be a full course ($FreePlaces(co) = 0$), and *st* be a student which is not enrolled in *co* yet. It is obvious that $BasicAddEnrollment(st, co, Wait)$ maintains *INV*. Similarly, operation $BasicUpdateFreePlaces(co, 1)$ does not violate *INV*. However, the simultaneous execution of these two operations violates *INV*. In fact, there will be free places in *co* while the enrollment of *st* is registered in status "Wait".

6 From a B specification to a relational implementation

Once the B specification corresponding to the different *IS-UML* diagrams is generated, it is successively transformed, using the B refinement technique, until a concrete level which is close to a B representation of the relational database and its corresponding programs (transactions) is obtained. In this way, the last step of coding becomes a straightforward task. Our data structure refinement rules use the algorithm defined in Batini et al. (1992) that derives a relational schema from a class diagram whose semantics is similar to ours. The main idea of this algorithm is to reorganize the class diagram into a set of independent classes. So, all elaborated concepts such as inheritance, association class and complex attributes must be transformed. The contribution of our approach is the joint refinement of data and operations (basic operations and transactions). In this way, we ensure, thanks to the refinement proofs, that the obtained program still verifies constraints specified at the abstract level. The following subsections illustrate the main steps of our refinement process (Mammar, 2002). Note that due to some technical contraints of the B method detailed in Mammar (2002), we have been obliged to consider that the global B specification is contained in a single machine. The operations of the subclasses are expanded by unfolding the operation calls. For instance, operation *AddStudent* is rewritten into:

AddStudent(st, na, cl, ic)$\stackrel{\triangle}{=}$
PRE $st \in PERSON - Person \wedge na \in STRING \wedge cl \in NAT \wedge$
$ic \in InsuranceCompany$
THEN
//adding a new instance to *Person*
$Person := Person \cup \{st\} \|$
$NamePers := NamePers \cup \{st \mapsto na\} \|$
$Subscribes := Subscribes \cup \{st \mapsto ic\} \|$

//adding a new instance to *Student*
$Student := Student \cup \{st\} \|$
$ClassLevel := ClassLevel \cup \{st \mapsto cl\}$
END

6.1 Data structure refinement

The objective of this phase is to transform all the UML data concepts that are not allowed in the relational model. For this purpose, these rules are applied:

i. Elimination of inheritance links: our approach supports only simple inheritance without cycles. Inheritance links can be removed according to three different strategies:

- removing the superclasses and keeping the subclasses.
- removing the subclasses and keeping the superclasses.
- keeping the super and subclasses and replacing the inheritance links with inclusion constraints.

In this paper, we give an illustration of the first case. By removing superclass *Person*, i.e variable *Person*, its attributes and associations must be redefined on subclasses $Teacher$ and $Student$. For instance, attribute *NamePers* (resp. association *Subscribes*) is split into two attributes (resp. associations) *NameTeach* and *NameStu* (resp.*SubscribesTeach* and *SubscribesStu*) which are defined as follows:

Invariant
//typing the added variables
$NameTeach \in Teacher \rightarrow STRING \wedge NameStu \in Student \rightarrow STRING \wedge$
$SubscribesTeach \in Teacher \rightarrow InsuranceCompany \wedge$
$SubscribesStu \in Student \rightarrow InsuranceCompany \wedge$

//the gluing invariant expressing the added variables with respect to the removed ones
$NamePers = NameTeach \cup NameStu \wedge$
$Subscribes = SubscribesTeach \cup SubscribesStu$

To establish the above invariant, *AddStudent* is refined by removing the substitution related to variable *Person* and replacing variables *NamePers* and *Subscribes* by *NameStu* and *SubscribesStu* respectively.
To keep the concept of *Person* in the system, we associate with the removed variable *Person* the following B definition: ($Person \triangleq Student \cup Teacher$) that is translated into a SQL view.

ii. Transition to the first normal form: in a relational database, all the attribute domains are atomic. According to its cardinality, a multivalued attribute of a class is replaced either by a new class or by several new atomic attributes.

iii. Transition from an object-based model to a value-based model: in a relational database, each table must have a key: it is a value-based model, that means that each tuple is identified by a set of attributes. On the contrary, the UML model is an object-based model: each object is identified by an object identifier that is independent of the value of its attributes. Thus, for each class, either a key already exists or the algorithm adds a new one. The selected key will be mapped to a primary key when generating the database schema. For example, as class *Student* has no key, this step adds a key variable *CardStu* defined by the following conjunct:

$$CardStu \in Student \rightarrowtail NAT$$

To establish this conjunct, operation *AddStudent* is refined by adding a substitution that updates variable *CardStu*.[1] Note that the precondition has been weakened (removed precisely):

AddStudent(st, na, cl, ic)$\triangleq$
BEGIN
ANY *no* **WHERE** $no \in NAT - ran(CardStu)$ **THEN**
$CardStu := CardStu \cup \{st \mapsto no\} \|$

[1] $ran(f) = \{y \mid \exists x.(f(x) = y)\}$.

$NameStu := NameStu \cup \{st \mapsto na\} \|$
$SubscribesStu := SubscribesStu \cup \{st \mapsto ic\} \|$
$Student := Student \cup \{st\} \|$
$ClassLevel := ClassLevel \cup \{st \mapsto cl\}$
END
END

iv. Transformation of monovalued associations: an association (or association class without attributes) between classes C and D with at least one monovalued role (attached for example to class C) is replaced by a new attribute in C. This new attribute is linked to the key of D by a referential constraint. In B, applying this rule replaces variable *SubscribesStu* by a new variable *SubscribesStuMig* defined by the following invariant:

$$SubscribesStuMig \in Student \rightarrow NAT \wedge$$
$$SubscribesStuMig = (NumCom \circ SubscribesStu)$$

The first conjunct types the added variable *SubscribesStuMig*, the second is a gluing invariant that relates *SubscribesStuMig* to *SubscribesStu*. Intuitively, this refinement replaces each instance *ic*, of $InsuranceCompany$, by the value of its key (*NumCom*). Therefore, operation *BasicAddStudent* is refined by replacing *ic* with *NumCom*(*ic*).

iv. Transformation of multivalued associations: multivalued associations (resp. association classes) become classes with two additional attributes linked by referential constraints to the keys of the classes involved with the association. The tuple of these two attributes denotes a key for the created class. Applying this rule adds two variables *Enrollment_Stu* and *Enrollment_Cou* defined by[2,3,4,5]:

$$Enrollment\_Stu = Enrollment \lhd (prj_1(Student, Course); CardStu)$$
$$Enrollment\_Cou = Enrollment \lhd (prj_2(Student, Course); Code)$$
$$(Enrollment\_Stu \otimes Enrollment\_Cou) \in Enrollment \rightarrowtail NAT \times NAT$$

The first two predicates are the gluing invariant. It states how the two added variables are related to the abstract ones. The first (resp. second) predicate states that variable *Enrollment_Stu* (resp. *Enrollment_Cou*) associates each tuple (st, co) with the key value of its first (resp. second) element. The last predicate specifies that tuple (*Enrollment_Stu*, *Enrollment_Cou*) constitutes a key for the class *Enrollment*.

In order to preserve these invariants, we refine all the operations referring to variable *Enrollment*. For instance, the operation *BasicAddEnrollment* is refined by adding the following two substitutions that update the added variables *Enrollment_Stu* and *Enrollment_Cou*:

$$Enrollment\_Stu := Enrollment\_Stu \cup \{(st \mapsto co) \mapsto CardStu(st)\}$$
$$Enrollment\_Cou := Enrollment\_Cou \cup \{(st \mapsto co) \mapsto Code(co)\}$$

[2] $X \lhd f = \{x \mapsto y \mid x \mapsto y \in f \wedge x \in X\}$.
[3] $prj_1(X, Y) = \{(x \mapsto y) \mapsto x \mid x \in X \wedge y \in Y\}$.
[4] $prj_2(X, Y) = \{(x \mapsto y) \mapsto y \mid x \in X \wedge y \in Y\}$.
[5] $f \otimes g = \{x \mapsto (y \mapsto z) \mid x \mapsto y \in f \wedge x \mapsto z \in g\}$.

v. Definition of the database schema: the idea is to gather under a single variable all the characteristics related to a given class. This single variable corresponds to a relational table. By this rule, the functions *Enrollment_Stu, Enrollment_Cou* and *Status*, defined on the variable *Enrollment*, are gathered under the single variable *Table_Enrollment* defined by the following gluing invariant:

$$Table\_Enrollment \in Enrollment \rightarrow (NAT \times NAT \times Type\_Status) \wedge$$
$$Table\_Enrollment = Enrollment\_Stu \otimes Enrollment\_Cou \otimes Status$$

Similarly, the substitutions related to the variables *Enrollment_Stu, Enrollment_Cou* and *Status*, are replaced with a single substitution acting on the single variable *Table_Enrollment*. The basic operation *BasicAddEnrollment* is thus refined into:

BasicAddEnrollment(stu, cou, sta) $\triangleq$
BEGIN
$Enrollment := Enrollment \cup \{stu \mapsto cou\} \|$
$Table\_Enrollment := Table\_Enrollment \cup$
$\{(stu \mapsto cou) \mapsto (CardStu(stu) \mapsto Code(cou) \mapsto sta)\}$
END

vi. Implementation of the data structure and the basic operations: for each cartesian product obtained in the previous refinement step, we specify a B machine, called an SQL machine, that implements it. For example, to implement variable *Table_Enrollment*, we define the SQL machine *Rel_Enrollment* as follows[6]:

MACHINE *Rel_Enrollment*
VARIABLES *Table_Enrollment*
INVARIANT
$Table\_Enrollment \subseteq$ **struct**$(CardStu : NAT, Code : NAT, Status : Type\_Status)$
INITIALISATION *Table_Enrollment*:=$\emptyset$
OPERATIONS
Insert_Enrollment(stu, cou, sta) = /*Adding a new tuple in the *Table_Enrollment**/
PRE $stu \in NAT \wedge cou \in NAT \wedge sta \in Type\_Status$ **THEN**
$Table\_Enrollment := Table\_Enrollment \cup \{rec(stu, cou, sta)\}$
END;
...
END

The implementation of the abstract variables obtained in the previous step by the variables of the imported SQL machines is achieved by expressing the existing relation between them. The main idea of this step is to identify each object by the value of its key: the two elements become the same. For instance, the implementation of variable *T_Enrollment* by variable *Table_Enrollment* is formally specified by the following gluing invariant:

$$\forall(x, y, z) \cdot (((x \mapsto y) \mapsto z) \in T\_Enrollment \Rightarrow rec(x, y, z) \in Table\_Enrollment) \wedge$$
$$\forall(x, y, z) \cdot (x \in NAT \wedge y \in NAT1 \wedge z \in Type\_Status \wedge$$
$$rec(x, y, z) \in Table\_Enrollment \Rightarrow ((x \mapsto y) \mapsto z) \in T\_Enrollment)$$

[6] $struct(A_1 : t_1, \ldots, A_n : t_n)$ denotes the structure of records in B. An element of this structure is denoted by $rec(a_1, \ldots, a_n)$.

The above conjunct defines a bijection relation between instances of the variables *T_Enrollment* and *Table_Enrollment*. Each instance of *T_Enrollment* is associated with exactly one instance of *Table_Enrollment* and vice-versa. This is why we do not have to express, in machine *Rel_Enrollment*, that tuple (*CardStu*, *Code*) is a key. In fact, the bijective feature of the gluing invariant makes each property on *T_Enrollment* also verified on *Table_Enrollment*. In the same way each basic operation obtained in the previous step is implemented by its equivalent ones defined in the SQL machine. For example, the basic operation *BasicAddEnrollment* is implemented by the operation *Insert_Enrollment*. Each of these SQL machines is mapped to a table. Integrity constraints are derived from the invariants defined at the abstract specification level and those added during the different refinement steps.

6.2 Substitution refinement

Up to now, only the data and the basic operations have been refined. The transactions defined in our specification were only rewritten with respect to the concrete variables. The transactions still contain B substitution constructors, such as *parallel* and *SELECT* constructors, which are not allowed at the implementation level. Moreover, the variables which the transitions may refer to are refined by the variables defined in the imported SQL machines. In Mammar and Laleau (2006), we have defined a generic and automatic process to refine transactions. The refinement of transactions consists hence of the following points:

1. We associate with each expression or predicate a B read operation that returns its value. The input parameters of these operations are the free variables appearing in the related expression or predicate. This operation refers to the variables defined in the SQL machines. The specification of these operations requires the definition of formal rules that rewrite the variables of the last refinement level with respect to those defined in the SQL machines. For example, with respect to variable $Table\_Enrollment$, variable $Status$ is rewritten into[7]:

$$Status \triangleq \bigcup x \cdot (x \in Table\_Enrollment | \{(x'CardStu \mapsto x'Code) \mapsto x'Status\})$$

Intuitively, function *Status* is rebuilt from *Table_Enrollment* by generating tuple $((x \mapsto y \mapsto z))$ from each record $rec(x, y, z)$. Abstract variables defined on *Course* being rewritten according to concrete ones, operation *AvailablePlaces* that returns the value of predicate $(FeePlaces(co) > 0)$ checking whether there are available places or not is defined as follows:

val ⟵ **AvailablePlaces(co)** $\triangleq$
PRE $co \in Course$ **THEN**
$val := Bool(FreePlaces(co) > 0)$
END

[7] $x'y$ denotes the value of field y for record x.

2. The *IF* constructor is kept since it is allowed at the B implementation level.
3. The *SELECT* constructor is replaced with the *IF* constructor. This implementation is correct since we have made the assumption that when an event occurs at least one of the guards of the transitions related to the event is satisfied.
4. The *Parallel* constructor is replaced with the *Sequence* one because it is not allowed at the B implementation level. Since by construction S_1 and S_2 do not modify the same variables, two solutions are possible: either $(S_1; S_2)$ or $(S_2; S_1)$. In Mammar and Laleau (2006), we described a formal approach that provides the best solution. The approach is based on the set of modified and read variables of S_1 and S_2. It is inspired by the lazy evaluation technique (Hughes, 1989). The expressions and predicates appearing in a transaction are classified into two sets: dependent expressions (resp. predicates), and mandatory expressions (resp. predicates). An expression (resp. predicate) is dependent if it may be not relevant to the transaction. For example, if we consider the following substitution:

   IF *pred* **THEN** *op*(*exp*) **END**

   the predicate *pred* is mandatory since *pred* is always evaluated, whereas *exp* is dependent since it becomes relevant only when *pred* is true. Having defined these two kinds of expressions/predicates, our solution consists in executing at first the substitutions that modify the fewest number of variables appearing in dependent expressions and predicates.

Applying this strategy, operation *AddEnrollment* is implemented by:

```
AddEnrollment(st, co) ≙
BEGIN
  /*Implementation of the operation CreateEnrollment*/
    VAR availableplaces_1 IN
      /*Evaluation of predicate FreePlaces(co) > 0*/
      availableplaces_1 ← AvailablePlaces(co);
      IF availableplaces_1 = TRUE THEN    Insert_Enrollment(st, co, Succ)
      ELSE
        VAR noavailableplaces IN
          /*Evaluation of predicate FreePlaces(co) = 0*/
          noavailableplaces ← NoAvailablePlaces(co);
          IF noavailableplaces = TRUE THEN
            Insert_Enrollment(st, co, Wait)
          END
        END
      END
    END
  /*Implementation of operation BasicUpdateFreePlaces */
  VAR availableplaces_2, newvalfreeplaces IN
    /*Evaluation of actual parameter (FreePlaces(co) − 1)*/
    newvalnbfree ← GetNewValueFreePlaces(co);
    /*Evaluation of predicate (FreePlaces(co) > 0)*/
    availableplaces_2 ← AvailablePlaces(co);
```

IF $availableplaces_2 = TRUE$ **THEN** $Update\_FreePlaces(co, newvalfreeplaces)$;
END
END
END

6.3 Refinement proofs

To ensure the correctness of the refinement process we defined, we have established the correctness of each refinement rule: about 200 refinement proofs. Proving the refinement of a substitution S by a substitution T consists in: 1. proving that if S terminates then T terminates as well, 2. the execution of S and T yields the same result. With AtelierB (version 3.5), 70% of these proofs have been automatically discharged, but this concerns only the easier proofs related to the first condition. The remaining proofs are rather hard and often very tedious to achieve. Fortunately, the generic feature of the refinement rules made it possible to define proof tactics that enable to automate the refinement proofs. This means that, once the proof of a generic refinement rule has been obtained, it is possible to reuse it in all the instantiations of the rule (Mammar and Laleau, 2003).

Similarly, to prove the refinement of transactions, we have studied and defined sufficient conditions that allow us to reuse the basic operation refinement proofs. We have also pointed out that, in the database domain, these conditions most often hold. Hence, the proof of the transaction refinement has been largely automated. More details can be found in Mammar and Laleau (2003) and Mammar (2002).

6.4 Generation of a relational implementation

The schema of the database and its related SQL statements are generated from the imported machines. With each imported machine *Rel_A*, are associated:

- a relational table *A* which defines the different fields that it contains,
- a JAVA class *A* which defines a set of methods corresponding to the different operations specified in machine *Rel_A*.

In Mammar and Laleau (2006), we have informally presented through an example the generation of these two elements. The objective of this part is to formalise such a generation process.

6.4.1 Generation of database schemas

Each imported machine *Rel_A* defines a relational table $Table\_A$ whose fields are obtained according to the following formal rules

a) Definition of the fields

$$\begin{array}{l}\textbf{Trad\_B\_SQL}(Table\_C \subseteq \textbf{struct}(Att_1 : T_1, \ldots, Att_n : T_n)) = \\ \quad \textbf{CREATE TABLE } C(\\ \quad\quad Att_1\ \textbf{Trad\_B\_SQL}(Att_1 : T_1)\ \textbf{NULL\_Constraint}, \\ \quad\quad \ldots \\ \quad\quad Att_n\ \textbf{Trad\_B\_SQL}(Att_n : T_n)\ \textbf{NULL\_Constraint} \\ \quad\quad \textbf{PRIMARYKEY\_Constraint},\end{array}$$

UNIQUE_Constraint,
REFERENTIAL_Constraint)

b) Definition of the types of fields

Trad_B_SQL($Att : NAT$) = *INT Check Att* ≥ 0
Trad_B_SQL($Att : BOOL$) = *BOOLEAN*
Trad_B_SQL($Att : STRING$) = *CHAR*(n) where n denotes an integer value.
Trad_B_SQL($Att : \{val_1, \ldots val_n\}$) = *CHAR(n) Check Att* **IN** (*"*val_1*"*,…, *"*val_n*"*)
where n denotes an integer value representing the length of the longest identifier val_i.

c) Definition of constraints

//if the B function corresponding to an attribute is total, the field must be not null
Trad_B_SQL($Att_i \in C \rightarrow T_i$) = **NOT NULL**

//The key of a table C corresponds to the injective function defined on C
//(or the functions whose direct product is injective on C)
Trad_B_SQL($Att_i \otimes \cdots \otimes Att_j \in C \rightarrowtail T_i \times \cdots \times T_j$) = **PRIMARY KEY** ($Att_i, \ldots, Att_j$)
//The other injective functions are mapped as UNIQUE constraints
Trad_B_SQL($Att_m \otimes \cdots \otimes Att_n \in C \rightarrowtail T_m \times \cdots \times T_n$) = **UNIQUE** ($Att_m, \ldots, Att_n$)

//The gluing invariant added by the refinement of monovalued associations is translated
//to a referential constraint
Trad_B_SQL($Att_1 = (Key_D \circ Att_2)$) = Att_1 **REFERENCES** $D(Key\_D)$

//The gluing invariant added by the refinement of multivalued associations is translated
//to two referential constraints
Trad_B_SQL($C \in A \leftrightarrow B \wedge Att_1 = C \lhd (prj_1(A, B); Key_A)$) = Att_1 **REFERENCES** $A(Key_A)$
Trad_B_SQL($C \in A \leftrightarrow B \wedge Att_2 = C \lhd (prj_2(A, B); Key_B)$) = Att_2 **REFERENCES** $B(Key_B)$

Applying these rules to the case study generates six tables as depicted in Fig. 5. Roughly speaking, there is one table for each class, multivalued association or association class. Attributes *Key_Student* and *Key_Teacher* are the two keys automatically added to classes *Student* and *Teacher* respectively. Let us notice that during the relational implementation generation, the translation of the B constraints into SQL is restricted to the above constraints (attribute type, primary key, etc.). Of course, the considered B specification may contain other kinds of constraints. However, they may be omitted in SQL since by refinement and its associated proofs, we are sure that the generated transactions will verify them. In this way, the time spent by the DBMS to verify these constraints will be saved. Moreover, one cannot specify additional constraints on the generated SQL tables. In fact, it is not recommended since, after such an adding we cannot ensure that the transactions will verify the added constraints.

6.4.2 Generation of the JAVA classes defining the basic SQL statements

Each operation defined in an imported machine Rel_A is mapped into a JAVA method. These methods are declared in the same JAVA class A. The skeleton of this class is constructed according to the following rule:

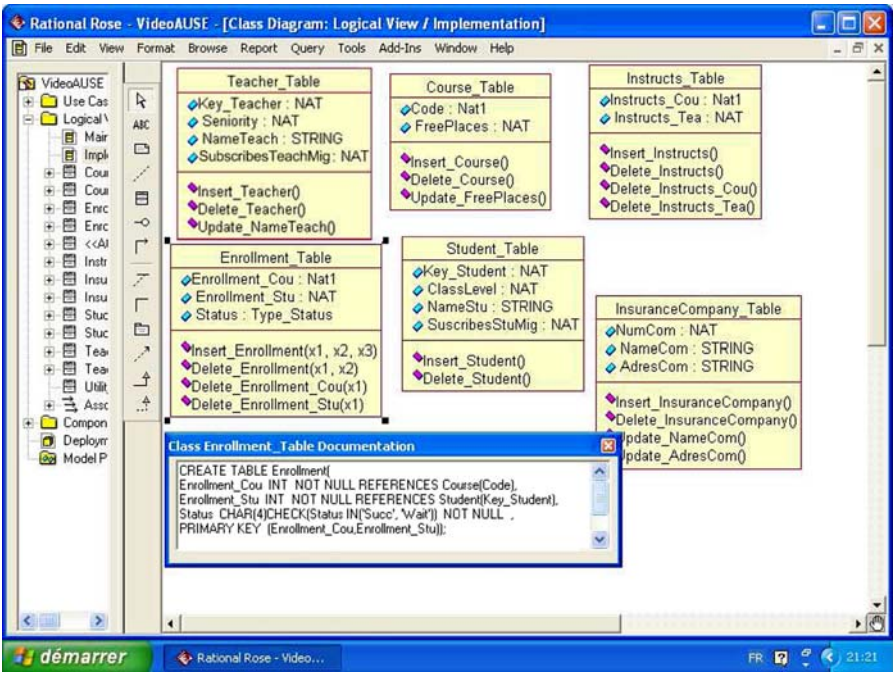

Fig. 5 The UML representation of a database schema

```
Trad_B_JSQL(MACHINE Rel_A ... END)=
    public class A {
          /*as many variables as operations defined in Rel_A*/
                    ...
        /*definition of a connection to table A_Rel*/
            private Connection conn;
        /*creation of an instance*/
        public A (Connection conn) throws SQLException{
            this.conn = conn;
          /*Definition of variables storing the SQL statements*/
                    ...
            /*Declaration of methods*/
                    ...  }}
```

The declaration and the definition of the variables storing the SQL statements are obtained from the operations defined in the imported machines. In this paper, we only give the rule translating an add operation.

Trad_B_JSQL(Insert_A($p_1, \ldots, p_n$) $\triangleq$ **PRE** $p_1 \in T_1 \wedge \cdots \wedge p_n \in T_n$ **THEN** $Table\_A := Table\_A \cup \{\mathbf{rec}(p_1, \ldots, p_n)\}$ **END**) =

```
    /*declaration of the variable storing the statement*/
      private PreparedStatement stmtInsert;
    /*definition of the variable storing the statement*/
    stmtInsert = conn.prepareStatement("insert into A"
        +"(Att1,  ... , Attn) values(?, ... ,?))");
   /*adding a new instance in table A*/
```

```
public void Insert_A(T1 p1, ... , Tn pn) throws SQLException {
        /*assigning values to the fields*/
    stmtInsert.setT1(1, p1); ... ;stmtInsert.setTn(n, pn);
        /*execution of the insertion*/
        stmtInsert.executeUpdate();}
```

For each operation defined in the imported machine, a *PreparedStatement* variable is declared. *PreparedStatement* is a JDBC class that provides a method *prepareStatement* allowing to store a pre-compiled SQL statement. This class also offers methods to assign values to the different parameters of the related statement (*SetTi*, *Ti* denotes a JAVA type), to execute statements (executeUpdate), etc.

6.4.3 Translation of transactions into JAVA

The translation of a transaction into JAVA is based on the different JAVA translation of the operations it calls. Since the control structures (IF, Sequencing) obtained at the implementation level are all supported by the JAVA language, this translation becomes straightforward: we have just to replace each B substitution constructor by its corresponding JAVA control structure, and operation calls by JAVA methods calls. Hence, transaction *AddEnrollment* is translated into:

```
public  void AddEnrollment(int st, int co){
   boolean availableplaces1 = course.AvailablePlaces(co);
   if (availableplaces1==true)
       enrollment.Insert_Enrollment(st, co,"Succ");
   else{
       boolean noavailableplaces= course.NoAvailablePlaces(co);
       if (noavailableplaces==true)
          enrollment.Insert_Enrollment(st, co, "Wait");
       }
   int newvalfreeplaces = course.GetNewValueFreePlaces(co);
   boolean availableplaces2 =course.AvailablePlaces(co);
   if (availableplaces2==true)
     course.Update_FreePlaces(co, newvalfreeplaces);
}
```

7 Discussion

7.1 Benefits and limitations of the approach

Our contribution is twofold: 1. a systematic approach to generate B specifications from *IS-UML* diagrams, and 2. a systematic B refinement process that derives a correct B representation of a relational implementation from B specifications. Translating *IS-UML* diagrams into B specifications assigns them a formal semantics. This helps users remove the ambiguities and alleviate the missing semantics of standard UML diagrams. Furthermore, in Mammar and Laleau (2006) we show that the B implementation of a transaction is in most cases longer than its abstract specification that hides all the implementation concerns that our refinement rules deal with. Moreover, to make our proposal more accessible for database users, we have developed a tool, called *UB2SQL* (Mammar and Laleau, 2005), that automates most of

the tedious tasks. Automating both the generation of B specifications from *IS-UML* diagrams and its refinement into a relational implementation frees engineers from tedious tasks. Indeed, without considering the different refinement steps and the proof phase, handwriting (in ASCII characters) B abstract specifications corresponding to a small diagram composed of one association involving two classes takes at least half a day. Moreover, *UB2SQL* has been integrated into the ROSE environment (Rational, 2003) to facilitate its use as several notations (UML, B and SQL) can be viewed in the same environment.

It is also important to consider the interest of using UML diagrams for building a B model. Actually, the crux of the construction of a formal model is to find appropriate abstractions, that is choosing the objects that make up the formal model (Mandrioli, 2004; Davies et al., 2004; Oliveira, 2004). In the database domain, we are used to constructing data models. After converting them into B specifications, we obtain the complete state of the formal model, that is all the variables. Moreover, the architecture of the formal model is also derived from the UML diagrams. Finally, a state diagram provides a different view of the operations of a class compared to the list of operations of a B machine. It describes ordering constraints between operations that cannot be straightforwardly expressed in B (we can do this in B by adding new state variables and preconditions but it is not a natural way of thinking). In fact state diagrams could be used for the model checking of the formal model, a verification technique presented below.

From a verification point of view, the proposed approach can be further improved. Using AtelierB, only the safety properties can be verified. Indeed, the B method is devoted to develop systems which, when they run, are guaranteed to produce correct results. However, it would be interesting to verify some other kinds of properties such as liveness properties. For instance,we have to check that the different preconditions of operations generated from state and collaboration diagrams are not always false. Similarly, we have to check that the predicate of a source state is not always false. A similar verification must also be achieved for guards that must be satisfiable and disjoint. The first case detects unreachable states. The second detects transitions that can never be fired. Such bad specifications cannot be discovered by the B prover because they make the precondition of the generated B operation always false and thus the associated proof theorem is automatically discharged (always true). For verifying this kind of properties, we think that other provers and/or model checkers should be investigated (Leuschel and Butler, 2003).

Moreover, the annotation of UML with B notations may be improved by considering the OCL language, which is an integral part of the latest UML language versions. For instance, it would be easier for database users to express the previous constraint *INV* of page 13 as OCL expressions instead of B notations. These OCL notations would be translated into B specifications using the *UML2B* tool developed in the context of the Lutin project (Marcano and Levy, 2002).

Another interesting topic that we have not yet considered concerns the link between partial functions in B and the use of the unknown value (NULL) in SQL. In B, an attribute undefined for some objects, such as the maiden name for a person, is represented by a partial function. In SQL, this is defined as an attribute that accepts NULL values. Whereas in SQL a three-valued logic has been explicitly specified to deal with NULL values, the B logic is a two-valued logic. However there exists some

work on the definition of a logic for a language with partial functions, with a three-valued interpretation for logical formulas (Behm et al., 1998; Burdy, 2000). Up to now, our approach is somewhat simple and consists in reserving a given value of each B type to denote the NULL value (see Laleau and Mammar, 2000a for more details).

7.2 Comparison with other similar approaches

Over the last decade, several approaches, based on formal methods, to develop database applications have been proposed. These approaches fall into two categories. The principle of the first category's approaches is to translate graphical notations into formal ones, like B (Marcano and Levy, 2002; Ledang et al., 2003; Treharne, 2002) and Z (Dupuy et al., 2000; Hall, 1990; Kim and Carrington, 1999), in order to obtain formal models on which rigorous reasoning and correctness proofs can be achieved. In Dupuy (2000) a formal approach, supported by the *RoZ* tool (Dupuy et al., 2000), to generate Z specifications from UML class diagrams is presented. Contrary to our approach, basic operations are only generated for classes. Operations on associations are not supported. Moreover, behavioral diagrams are not considered. A similar tool, named *FuZe*, which deals with Fusion class diagrams edited under the *Paradigm Plus* environment, has been developed by Bruel and France (1996). The generation of B formal specifications from UML diagrams has been also investigated by Snook and Butler (2001) and Ledang and Souquières (2001). Their approaches are supported by *U2B* and *ArgoUML* + *B* tools respectively. Both are restricted to class and state/transition diagrams. In Snoock's approach only the insert operation is generated for associations. Operations on attributes are not considered. Moreover, the considered domain being reactive systems, the semantics attached to UML diagrams is rather different from ours. Let us remark that none of these tools includes generation of code. Although such approaches move the development of database applications forward, they do not cover all the development phases. Lano et al. (2004) has developed an approach, supported by the UML-RSDS tool, to generate B, JAVA and SMV specifications from a subset of UML diagrams comprising class diagrams involving inheritance, state diagrams and constraints expressed in OCL. The rules used to translate class diagrams are similar to ours. The generation of SMV specifications permits to detect some intra- and inter-diagram inconsistencies. However no details are given about the formalism and the correctness of the JAVA process generation.

Approaches of the second category use the refinement technique to derive executable code from formal specifications. The most representative approaches are those presented in Matthews and Locuratolo (1999), Edmond (1995), Gunther et al. (1993) and Schewe et al. (1991). Matthews and Locuratolo (1999) proposes a refinement process to develop object-oriented database applications. Based on the ASSO formal method, which is very similar to B, the defined process aims at reorganizing an inheritance hierarchy into independent classes. The application to develop is first described and refined in ASSO, the resulted ASSO specification is then translated into B in order to validate the entire development. The validation is made using B tools since the ASSO method lacks such tools. The main drawback of this approach is that the concepts available in ASSO are rather limited: classes, monovalued attributes, inheritance. These concepts are not sufficient to capture the whole semantics of database

applications. In Gunther et al. (1993) and Schewe et al. (1991), an approach to generate a DBPL code from a B specification derived from TDL specification is presented. The main drawback of this approach is that formal rules are defined only for the refinement of data, transactions are not considered. In Edmond (1995), another method to derive relational implementations from Z specifications is proposed. Nevertheless, this method is only presented through an example, no formal rules are provided. Qian (1993) has defined a logic for an abstract specification of relational transactions. The deductive tableau technique is then used to derive a program from this specification. This approach requires strong mathematical and logical background. Indeed, deduction rules of the deductive tableau technique being a general approach, the author illustrates his approach through practical examples showing that some steps often require non-systematic adaptations to match deduction rules. We think that such non-systematic adaptations do not allow neither the definition of generic rules nor the automation of the method.

Compared to these different approaches, our proposal takes a wider range of concepts into account: classes, attributes and their mutability, associations, inheritance, operations, etc., and it covers the entire development process from design to implementation. Table 1 presents a comparison of these approaches based on seven representative criteria (minus "−" means that the concept is not supported).

8 Conclusion and future work

This paper reports on our experience of developing a formal approach, based on UML and the B formal method to build database applications. This approach includes two main phases: 1. translating UML diagrams into B specifications, and 2. refining the obtained B specifications into a B representation of a relational implementation. The approach we propose permits to cover a wide range of UML concepts that are relevant to design database applications. Moreover, our approach is supported by an automatic tool *UB2SQL* which makes it more accessible for software engineers. This tool allows a better integration of formal methods into the development process. In addition, such a tool would help them detect inconsistencies in UML diagrams from the inconsistencies discovered in the corresponding B specifications as described in Laleau and Polack (2002). We have tried out our approach on academic case studies (much more complex than the one presented here: 20 classes and 15 associations) and on student projects, and promising results have been obtained. Also, it is planned to use *UB2SQL* in the context of the EDEMOI[8] (Ledru, 2003) project which aims at investigating the integration of semi-formal methods (UML) and formal ones (B, Z, etc.) for modeling and verifying airport security.

Even if the approach described in this paper has been specifically defined for the development of relational database applications, we believe that the way it has been defined could be applied to other kinds of application (but always with the B method). Indeed, we always use the following main principles: (1) defining a domain-dedicated semantics of UML diagrams, (2) translating the dedicated semantics into B, and (3)

[8] The EDEMOI project is supported by the French National Action Concertée Incitative "Sécurité Informatique".

Table 1 Comparison of the different approaches

Approaches	Bruel and France (1996)	Dupuy (2000)	Ledang and Souquières (2001)	Marcano and Levy (2002)	Snook and Butler (2001) (Schewe et al., 1991)	Matthews and Locuratolo (1999)	Gunther et al. (1993)	Edmond (1995)	Qian (1993)	UB2SQL.
Criteria										
Formalization degree	High	High	High	High	High	Low	Medium	Low	Medium	High
Graphical notations	Fusion	UML	UML	UML	UML	−	−	−	−	UML
Supported concepts	class diagrams	class diagrams	class diagrams State diagrams OCL expressions	class diagrams State diagrams Collaboration diagrams	class diagrams State diagrams	classes monovalued attributes inheritance	classes attributes associations	classes associations		class diagrams State diagrams Collaboration diagrams
Elements for which operations are generated	classes	classes	−	classes associations	classes	classes	classes associations	classes associations		classes associations
Formal languages	Z	Z	B	B	B	ASSO, B	DBPL, B, TDL	Z	First order logic	B
Code generation	−	−	−	−	−	−	+	−	+	+
Tool support/Development environment	FuZe/ Paradigm Plus	RoZ/ ROSE	ArgoUML+B/ ArgoUML	UML2B/ Objecteering	U2B/ ROSE	−	−	−	−	UB2SQL/ ROSE

defining a refinement process that takes the concepts of the target language into account.

Future work includes the generalization of our approach to a better consideration of integrity constraints during the generation of B operations. It would be interesting to automatically determine the precondition of an operation by just providing its body (B substitution) and the invariant we would like to preserve. Other ongoing work concerns the optimization of the SQL code generated from transactions in order to reduce their execution time and/or the size of buffers used during the evaluation of SQL requests (Mammar and Laleau, 2006). We also plan to augment *UB2SQL* by supporting the generation of JAVA/SQL code from the last B refinement level according to the translation rules we have defined in Mammar and Laleau (2004) and to support inheritance and aggregation/composition concepts. Note that the refinement of an aggregation/composition structure is dealt with as if it were a simple association; such a concept affects only the set of correct transactions. Indeed, for instance, a transaction that deletes a composite without deleting all its parts will not be considered as correct. We also plan to define the concept of B data structure refinements as UML model transformations (Lano, 2005) in order to make our refinement rules more understandable to software developers.

Appendix: B substitutions

This appendix presents the semantics of the main B substitutions used in this paper (see Table 2). The double square brackets denote optional elements.

Table 2 The main substitutions of the B language

Notation B	Semantics
Skip	Does not modify anything
$x := E$	Assigns the value of E to the variable x
PRE P **THEN** S **END**	Executes S under the hypothesis that P is true
IF P **THEN** S **ELSE** T **END**	If P is true, S is executed, otherwise T is executed
SELECT P **THEN** S **WHEN** Q **THEN** T **END**	S or T is executed according to the truth-value of P and Q If P and Q are both true, the substitution to execute is arbitrarily chosen. If P and Q are both false, the substitution is not implementable.
ANY X **WHERE** P **THEN** S **END**	indeterministically selects a value of X verifying P, then executes S
$u \leftarrow \mathrm{Op}(v)$	Calls the operation Op with the parameters v and assigns its result to u
$S \mid\mid T$	Executes simultaneously S and T
$S[]T$	Executes either S or T
$S; T$	Executes S then T

References

Abrial, J. R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)

B-CORE: B-Toolkit Release 3.2. Manual. Oxford, UK (1996)

Barros, R.S.M.: Deriving relational database programs from formal specifications. In: Naftalin, M. Denvir, B.T. Bertran, M. (eds.) Industrial Benifit fo Formal Method, Second International of Formal Methods Europe (FME'94), vol. 873 of LNCS, Springer-Verlag (19941)

Barros R.S.M.: On the formal specification and derivation of relational database application. Ph.D thesis, Department of Computing Science, the University of Glasgow (1994b)

Barros, R.S.M.: On the formal specification and derivation of relational database applications. Elec Notes in Theoretical Comptuer Science **14** (1998)

Batini, C., Ceri, S., Navathe, S.: Conceptual Database Design: An Entity-Relationship Approach. Benjamin/Cummings Publishing Company (1992).

Behm, P., Burdy, L., Meynadier, J.-M.: Well defined B. In: Bert, D. (ed.) B'98: Recent Advances in the Development and Use of the B Method, Second International B Conference, vol. 1393 of Lecture Notes in Computer Science pp. 29–45. Springer-Verlag (1998)

Bruel, J.M., France, R.B.: A formal object-oriented CASE tool for the development of complex systems. In: 7th European Workshop on Next Generation of Case Tools. Available at citeseer.ist.psu.edu/bruel96formal.html (1996)

Burdy, L.: Traitement des expressions dépourvues de sens de la théorie des ensembles: Application à la méthode B. Ph.D. thesis, CEDRIC Laboratory, Paris, France (2000)

Clearsy: Atelier B, Manuel de Référence. available at http://www.atelierb.societe.com (2003)

CLEARSY: (2004) http://www.b4free.com/

Codd, E.: A relational model for large shared data banks. Commun of the ACM **13**(6) (1970)

Davies, J., Simpson, A., Martin, A.: Teaching formal methods in context. In: Neville Dean, C, Boute, R.T. (eds.) Teaching Formal Methods, CoLogNET/FME Symposium (TFM 2004), vol. 3294 of LNCS. Springer (2004).

Dupuy, S.: Couplage de notations semi-formelles et formelles pour la spcification des systmes d'information. Ph.D. thesis, Universit Joseph Fourier (2000)

Dupuy, S., Ledru, Y., Chabre-Peccoud, M.: An overview of RoZ: A tool for integrating UML and Z specifications. In: Wangler, B., Bergman, L. (eds.), 12th International Conference Advanced Information Systems Engineering (CAiSE'00), vol. 1789 of LNCS Springer-Verlag (2000)

Edmond, D.: Refining database systems. In: Bowen, J.P. Hinchey, M.G. (eds.) The Z Formal Specification Notation (ZUM'95), vol. 967 of LNCS Springer-Verlag (1995)

Elmasri, R., Navathe, S.: Fundamental of Database Systems (4th edition). Addison-Wesley (2003)

Gunther, T., Schewe, K.D., Wetzel, I.: On the derivation of executable database programs from formal specifications. In: Woodcock, J.C.P., Larsen, P.G. (eds.) Industrial-Strength Formal Methods, First International Symposium of Formal Methods Europe (FME'93), vol. 670 of LNCS Springer-Verlag (1993)

Hall, A.: Using Z as a specification calculus for object-oriented systems. In: VDM'90: 3rd International Conference, Kiel, Germany, vol. 428 of LNCS Springer-Verlag (1990)

Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8**(3) (1987)

Hughes, J.: Why functional programming matters. Compu J **32**(2) (1989)

Kim, S., Carrington, D.: Formalizing the UML class diagram using OBJECT-Z. In: UML99, vol. 1723 of LNCS Springer-Verlag (1999)

Laleau, R.: On the interest of combining UML with the B formal method for the specification of database applications. In: ICEIS'00: 2nd International Conference on Enterprise Information Systems. Available at http://www.univ-paris12.fr/lacl/laleau/ (2000).

Laleau, R., Mammar, A.: A generic process to refine a B specification into a relational database implementation. In: Bowen, J.P., Dunne, S., Galloway, A., King, S. (eds.) The First International Conference of B and Z Users on Formal Specification and Development in Z and B (ZB'00), vol. 1878 of LNCS Springer-Verlag (2000a)

Laleau, R., Mammar, A.: An overview of a method and its support tool for generating B specifications from UML notations. In: The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00) IEEE Computer Society (2000b)

Laleau, R., Polack, F.: A rigorous metamodel for UML static conceptual modelling of information systems. In: Dittrich, K.R., Geppert, A., Norrie, A.C. (eds.) 13th International Conference on Advanced Information Systems Engineering (CAiSE'01), vol. 2068 of LNCS Springer-Verlag (2001a)

Laleau, R., Polack, F.: Specification of integrity-preserving operations in information systems by using a formal UML-based language. Info Soft Techno **43**(12) (2001b)

Laleau, R., Polack, F.: Coming and going from UML to B: A proposal to support traceability in rigorous IS development. In: Bert, D., Bowen, J.-P., Henson, M., Robinson, K. (eds.), Formal Specification and Development in Z and B (ZB'02), vol. 2272 of Lecture Notes in Computer Science, Springer-Verlag (2002)

Lano, K.: Advanced Systems Design with Java. UML and MDA Elsevier (2005)

Lano, K., Clark, D., Androutsopoulos, K.: UML to B: Formal verification of object-oriented models. In: Boiten, E.A., Derrick, J., Smith, G. (eds.) IFM'04: 4th International Conference on Integrated Formal Methods (2004)

Ledang, H., Souquières, J.: Modeling class operations in B: Application to UML behavioral diagrams. In: The Sixteenth IEEE International Conference on Automated Software Engineering (ASE'01). IEEE Computer Society (2001)

Ledang, H., Souquieres, J., Charles, S.: ArgoUML+B : Un Outil de Transformation Systématique de Spécifications UML vers B '. In: Proceedings of AFADL'2003, INRIA (2003)

Ledru, Y.: http://www-lsr.imag.fr/EDEMOI/ (2003)

Leuschel, M., Butler, M.-J.: ProB: A model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) The 12th International FME Symposium (FME'03), vol. 2805 of LNCS. Springer-Verlag (2003)

Mammar, A.: Un environnement formel pour le développement d'Applications bases de données. Ph.D. thesis, CEDRIC Laboratory, Paris, France (2002)

Mammar, A., Laleau, R.: Design of an automatic prover dedicated to the refinement of database applications. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) The 12th International FME Symposium (FME'03), vol. 2805 of LNCS. Springer-Verlag (2003)

Mammar, A., Laleau, R.: Génération de Code à Partir d'une Spécification B : Application aux Bases de Données'. In: Proceedings of Approches Formelles dans l'Assistance au Développement de Logiciels" (AFADL'2004) (2004)

Mammar, A., Laleau, R.: UB2SQL: An integrated environment based on UML and the B formal method for the development of database applications. Technical report, University of Luxembourg. Available at http://se2c.uni.lu/users/AM (2005)

Mammar, A., Laleau, R.: From a B formal specification to an executable code: Application to the relational database domain. Information & Software Technology **48**(4) (2006)

Mandrioli, D.: Advertising formal methods and organizing their teaching: Yes, but ..... In: Neville Dean, C., Boute, R.T. (eds.), Teaching Formal Methods, CoLogNET/FME Symposium (TFM 2004), vol. 3294 of LNCS. Springer (2004)

Marcano, R., Levy, N.: Transformation rules of OCL constraints into B formal expressions. In: Jurjens, J., Cengarle, M. V., Fernandez, E. B., Rumpe, B., Sandner, R. (eds.) Critical Systems Development with UML – Proceedings of the UML'02 Workshop, Technische Universität München, Institut für Informatik (2002)

Matthews, B., Locuratolo, E.: Formal development of databases in ASSO and B. In: Wing, J.-W., Woodcock, J.-C.-P., Davies, J.-W.-M. (eds.), Proceedings of FM'99: World Congress on Formal Methods, vol. 1709 of LNCS Springer-Verlag (1999)

Melton, J., Simon, A.: Understanding the New SQL: A Complete Guide. Morgan Kaufmann (1993)

NoMagic: http//:www.MagicDraw.com (2005)

Oliveira, J.: A survey of formal methods courses in european higher education. In: Neville Dean, C., Boute, R.T. (eds.) Teaching Formal Methods, CoLogNET/FME Symposium (TFM 2004) vol. 3294 of LNCS Springer (2004)

OMG: http://www.omg.org/ (2005)

Qian, X.: The deductive synthesis of database transactions. ACM Trans Data Syst **18**(4) (1993)

Rational: http://www.rational.com (2003)

Rational: http://www.rational.com (2005)

Schewe, K., Schmidt, J., Wetzel, I.: Specification and refinement in an integrated databaseapplication environment. In: Prehn, S., Toetenel, T. (eds.), VDM'91: Proceedings of Formal Software Development Methods vol. 552 of LNCS Springer-Verlag (1991)

Snook, C., Butler, M.: Using a graphical design tool for formal specification. In: Kadoda, G. (ed.) The 13th Annual Workshop of the Psychology of Programming Interest Group (PPIG'01). Available at http://www.ppig.org/papers/13th-snook.pdf (2001)

Treharne, H.: Supplementing a UML development process with B. In: FME2002:International Symposium of Formal Methods Europe, vol. 2391 of LNCS. Springer-Verlag (2002)