An Automatic Generation of B Specifications from Well-Defined UML Notations for Database Applications

Régine Laleau and Amel Mammar

{laleau, mammar}@iie.cnam.fr

Address: CEDRIC-IIE(CNAM)

18 Allée Jean Rostand, 91025 Evry, France

Tel / Fax: 00.331.69.36.73.{47, 73}/ 00.331.69.36.73.05

Abstract:

Our project aims at providing users with a complete formal environment for the development of database applications. The article describes the different steps for deriving a formal specification from a subset of well-defined UML notations: class diagrams give the static part of the specification and basic operations, state and collaboration diagrams give the application-dependent operations. The different kinds of proofs, automatically generated by the prover, are also presented. Finally, the prototype support tool features are outlined.

Keywords: Database Applications, UML, B, Automatic Generation, Proofs.

1. Introduction

At present, two ways for developing software are used. On one hand, semi-formal methods provide diagrammatic notations which are intended to be intuitive, that is close to the typical mental processes people use in initial modelling of an application domain. On the other hand, formal methods allow absolute precision, consistency verification, proof of properties and refinement to a provably correct code. Up to now, they are mainly used for the development of critical systems.

Database systems are now basic components of most of software systems: e-business, financial and stock-exchange systems,….Even if they are not truly critical systems, they require a certain degree of reliability which can be achieved by using formal methods. Our project aims at formally designing database applications. However, we have to take into account the usual analysis and design methods (such as UML, SA) employed in this domain. That is why, we propose to combine such a classical method with a formal one, as it is proposed in [2], [9], [15].

UML [17] is now standardised by the OMG. In spite of real efforts, it isn't sufficiently formal, specially if proof of consistency between the different components has to be achieved. B is a formal method developed by Abrial [1]. It is a complete method and supports a large segment of the development life cycle: specification, refinement and implementation. It ensures, thanks to refinement steps and proofs, that the code satisfies its specification. It has been used in significant industrial projects and commercial case tools are available in order to help the specifier during the development process. These are the main arguments to use B rather than Z or another formalism. Indeed, in Z, a type checker and tool assistant proof are available, but no support tool for refinement and no automatic prover are provided. VDM provides a standardised semantics and a good support tool, but the structuration of the formal presentation remains rather primitive. So, we can think that the B method is the best suitable formal method for the moment.

In this framework, our objective is to provide designers with a tool that automatically derives B specifications from UML specifications. As UML becomes more mature and is taken up by a wider range of practitioners, the need for focused variants of the approach, targeted to specific types of system has been recognised by OMG. However, these variants need clear semantics and, potentially, specialised tool support. A preliminary step of our work is to develop a version of a well-defined UML adapted for data intensive applications. The main characteristic of such systems is that they manage massive amounts of highly structured and inter-related data that are subject to numerous rules of consistency. Thus the complexity of the design is to ensure that these data are handled safely and efficiently whereas operations that act upon are rather simple and generic. Hence, in contrast to applications like process control, the emphasis is on the preservation of data consistency.

Some research works have already studied this kind of systems. In [10], the derivation of database programs is presented. From a conceptual modelling language, which is already formal, B is used as an intermediate language to ensure a formal refinement. In [20], they used a specific formal language TM to design transactions, then the prover Isabelle/HOL is used to make the proofs. The main drawback is that two distinct formal languages are used, which doesn't facilitate the work of the designer. The work [5] consists of translating an UML application(described with the Rose toolkit) into Z specification. It is focused on the generation of basic operations and their preconditions from a class diagram. More complex operations are not considered. [3] uses the Fusion object-oriented method. This method isn't dedicated to a specific domain. Thus, the definition of generic operations is not possible. Only Z specification skeletons are generated, which must be completed by the designer.

Up to now, among UML concepts which have a rather well defined semantics accepted by the UML community, we have extracted a subset adapted to our domain. This excludes concepts such as aggregation/composition and inheritance. Indeed, for example, inheritance of associations, aggregation or state diagrams have not been clarified yet. For this subset, a formal semantics has been proposed [12]. Then, a set of rules which translate them into B specifications have been defined [7].

The aims of this paper are to describe what can be automatically generated from well-defined UML diagrams (Section 3) and the prototype tool which supports this automation (Section 4). Section 2 gives a concise overview of the B method.

2. Overview of B

In B, specifications are organised in Abstract Machines. Each machine encapsulates state variables, an invariant constraining the state variables and operations on the state variables. The invariant is a predicate in a simplified version of ZF-set theory, enriched by many relational operators. It is possible to declare in an Abstract Machine abstract sets by giving their name, without further details. This allows the actual definition of types to be deferred to implementation. Operations are specified in the Generalised Substitution Language which is a generalisation of Dijkstra's guarded command notation. Hence operations are defined using substitutions. A substitution is like an assignment statement. It allows us to identify which variables are modified during an operation, while avoiding to mention which ones are not. The generalisation allows the definition of non-deterministic and preconditioning substitutions. The preconditioning substitution is of the form **PRE** *p* **THEN** *s* **END** where *p* is a predicate and *s* a substitution. When *p* holds, the substitution *s* is executed, otherwise the execution aborts. Note that, in B, loop statements and sequential composition are not allowed at the specification level. They are only allowed in refinement and implementation ones. Proof of internal consistency of an Abstract Machine requires that each machine operation on the state variables maintains the invariant. Operations may be parameterised. Large machines are constructed using smaller ones through various access links. A machine Ma may use another machine Mb: the variables of Mb can be used but not modified in Ma. A machine Ma may include another machine Mb : the variables of Mb can be read in Ma, but its modification is possible, only by using the operations of Mb. The include relationship is not transitive for operations: operations of Mb are not themselves operations of Ma. A machine may be included (directly or transitively) only one time in the same machine. The aim of these mechanisms and restrictions is to achieve incremental specification with separate consistency proofs.

We assume that the reader is familiar with basic formal expressions in the style of B, which is similar to Z.

3. Generating B Specifications from well-defined UML Diagrams

UML provides a graphical object-oriented modelling language. In this paper, we focus on the analysis level, thus the UML diagrams relevant to describe database applications, are :

- class diagrams representing the static structure of the system,
- state diagrams describing the local behaviour of objects,
- collaboration diagrams which show how objects collaborate to perform functions of the system.

In this section, the different diagrams are described. For each diagram, we specify how it needs to be completed by formal notations in order to increase the amount of specification generated automatically.

3.1 Class Diagrams

In database application design, class diagrams are the basis of the software architecture. At the analysis level, a class diagram contains classes describing persistent data of a system (in UML such classes are called entity classes). UML class diagrams derive from semantic models [18] used in database design for a long time. Thus their semantics is rather well defined. Of course, implementation details (utility and template classes, visibility markers, …) have to be avoided. Up to now, UML ambiguous concepts such as inheritance and aggregation/ composition are not considered, because they need firstly to be precisely defined. Several research groups are working on this area, in particular the pUML group [6]. In [8], we have already proposed a solution for the derivation of a B specification from a OMT specification. The translation of the static part is similar in UML, whereas the translation of dynamic part differs. In this paper we just sum up the main results of our proposition for UML. For more details, interested readers can see [7].

a) Description

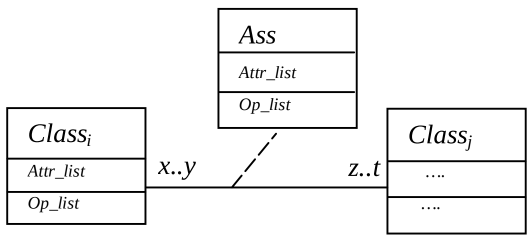

Figure 1. A Class Diagram

A class is a descriptor for a set of objects with similar structure represented by a set of typed attributes (*Attr_List*) and common behaviour expressed by a set of operations (*Op_List*). An attribute has a name, a type which may be atomic (NAT, String, …) or complex (Set of string, Sequence of NAT), a multiplicity specifying the number of values that the attribute may hold and a mutability property that indicates if the value of this attribute may change or not. We impose that UML basic types correspond to B basic types. A label {*K*} may be attached to an attribute (or a set of attributes) to indicate the key of a class. Classes are linked by 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. Each end of an association is a role. A role is characterised by its multiplicity (*x ..y*) indicating the number of class instances that can be related to one instance of the other class. A role may also be named. An association may have a frozen role. It means that a link can be created only at the creation of an object of the class associated to the role. It is never changed for its lifetime and is deleted when the corresponding object is deleted.

b) B formalisation

The first step consists of describing the static part of the B specification which mainly derives from the class diagram (the operations are not considered). Each class $Class_i$ is mapped into a B machine, called $BasicClass_i$ which contains an abstract set $CLASS_i$ of all possible instances and a class variable $Class_i$ typed by $Class_i \subseteq CLASS_i$ and representing the set of existing instances.

SETS $CLASS_i$
VARIABLES $Class_i$
INVARIANT $Class_i \subseteq CLASS_i$

Each attribute *Att* is translated into a relation *Att_var* defined from $Class_i$ to the attribute type. This relation becomes a function depending on the multiplicity of the attribute. A key is translated into an injection.

VARIABLES Att_var , Att_var_{key}
INVARIANT
$Att\_var_{key} \in Class_i \rightarrowtail Type\_Att_{key} \wedge Att\_var \in Class_i \leftrightarrow Type\_Att$

A binary association *Ass*, defined between two classes $Class_i$ and $Class_k$, is translated into a variable which is defined as a relation between the two class variables associated to the classes. As for attribute, the relation becomes a function, injection, … depending on the multiplicity of its roles. An association attribute is modeled like a class attribute.

VARIABLES Ass, Att_var
INVARIANT $Ass \in Class_i \leftrightarrow Class_k \wedge Att\_var \in Ass \leftrightarrow Type\_Att$

If an association has a frozen(or mandatory monovalued) role then it is specified in the same machine containing the class corresponding to the role. Otherwise, a new machine, called *BasicAss*, is defined which uses the machines corresponding to the two classes involved in the association. The structure of the obtained specification corresponding to Figure 1 is described in Figure 6.

c) Generating Basic Update Operations

If operations are described in a class and if they are specified using B notations then they are included in the machine corresponding to the class, otherwise the designer needs to intervene. In order to help the designer in the specification of its operations, we have defined a process to automatically generate basic operations for each class and association. In data-intensive information systems, basic operations are divided into two groups : update operations, which modify the data by inserting, deleting or updating objects or links, and read-only operations, which do not modify the data and thus are not addressed in this paper. Update operations are application independent and have always the same structure. Hence they can be automatically generated from the specification of the class diagram. They are called in more application specific operations which are described with other UML diagrams.

The main idea is to determine the strongest precondition such that the invariant of the machine where the operation is generated will be satisfied after its execution, even if certain proofs may be achieved in other including machines. Thus, it allows incremental proofs and helps the designer to build the precondition of the calling operations. For the moment we only consider multiplicity constraints but other constraints could be considered and an ultimate aim should be to analyse all the conjuncts of the invariant to determine the strongest precondition. The detail of the generation is given in [16], where operations and their preconditions are presented for each type of multiplicity. Here we sum up the main results :

- $BasicAddClass_i$ adds an instance in $Class_i$. The inputs of this operation are the instance to be added and the values of the different mandatory attributes. This operation takes into account possible keys and frozen associations. It is formalised as follows:

$BasicAddClass_i(cl, val\_att_{key}, \ldots, val\_att_n, SubsetClass_j, \ldots)$
PRE
$cl \in CLASS_i - Class_i \wedge$
$val\_att_{key} \in Type\_Att_{key} - \mathrm{ran}(Att_{key}) \wedge$
…..
$val\_att_n \in Type\_Att_n \wedge$
$SubsetClass_j \subseteq Class_j$
THEN
$Class_i := Class_i \cup \{cl\} \parallel$
$Att\_var_{key} := Att\_var_{key} \cup \{cl \mapsto val\_att_{key}\} \parallel$
…..

$Att\_var_n := Att\_var_n \cup \{cl \mapsto val\_att_n\}\} \parallel$
$Ass_{ij} := Ass_{ij} \cup \{cl\} \times SubsetClass_j$
END

where:

- $\parallel$ is the parallel operator. If S and P are two B-substitutions, ($S||$P) represents the simultaneous execution of S and P.
- cl is the instance of the $Class_i$ to be added,
- val_att_{key} is the value of the key attribute Att_var_{key},
- the different parameters val_att_n are the values of the mandatory attributes of the class,
- Ass_{ij} is an association defined between $Class_i$ and $Class_j$, with a frozen role for $Class_i$,
- $SubsetClass_j$ is a subset of $Class_j$ which will be related to the instance cl.

- *BasicDelClass*$_i$ deletes a set of given instances from the existing ones, and also from the domain or range of all relations (representing attributes or frozen roles) defined over this class. It is specified as follows:

BasicDelClass$_i$(SubsetClass$_i$)
PRE $SubsetClass_i \subseteq Class_i$
THEN
$Class_i := Class_i - SubsetClass_i \parallel$
/* the class is the source set of var$_j$ */
$var_j := SubsetClass_i \triangleleft var_j \parallel$
/* the class is the target set of var$_k$ */
$var_k := var_k \triangleright SubsetClass_i$
END

where $\triangleleft$ and $\triangleright$ are the anti-restriction and coanti-restriction operators defined by:

$$A \triangleleft r = \{(x \mapsto y) \in r \wedge x \notin A\}$$
$$r \triangleright A = \{(x \mapsto y) \in r \wedge y \notin A\}$$

where r denotes a relation and A a subset of the source set of r.
For example, the class *Book*, represented in following figure, is translated by:

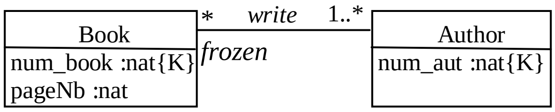

Figure 2. Example of a Frozen Role

SETS *BOOK*
VARIABLES *Book, num_book, pageNb*
INVARIANT $Book \subseteq BOOK \wedge num\_book \in Book \rightarrowtail NAT \wedge pageNb \in Book \rightarrow NAT$

The role is frozen since the authors of a book are known when the book is created and can never change. The *write* association is translated by:

VARIABLES *write*
INVARIANT $write \in Book \leftrightarrow Author \wedge dom(write) = Book$

where $dom(f) = \{x \mid \exists\, y.\ f(x) = y\}$.

The *BasicDelBook* operation of the class *Book* is specified as follows:

BasicDelBook(book)
PRE $book \subseteq Book$
THEN
$Book := Book - book \parallel$
$num\_book := book \triangleleft num\_book \parallel$
$pageNb := book \triangleleft pageNb \parallel$
$write := book \triangleleft write$
END

This operation takes into account the *write* association, because the association is frozen and then is defined in the same machine as the class Book.

- *BasicChangeAtt* changes the value of a mutable attribute *Att*. Its inputs are the instance (of the class or association) whose attribute is to be updated and its new value. This operation is specified as follows:

BasicChangeAtt(cl, val_att)
PRE $cl \in E_i \wedge val\_att \in \text{Type\_Att}$
THEN $Att\_var(cl) := val\_att$
END

where E_i corresponds to the class or the association variable of the attribute *Att*.

Each operation *Op* gives raise to one proof obligation. This proof obligation states that the operation preserves(doesn't violate) the invariant. It is expressed by:

$$(INV \wedge PC) \Rightarrow ([Op]\, INV)$$

where:

- *INV* is the invariant of the machine,
- *PC* is the precondition of the operation *Op*.
- [*Op*]*INV* is the predicate obtained by replacing, in *INV*, each variable with its new(if it is changed) value after the execution of the operation *Op*.

In practice, one proof obligation is generated for each part of the invariant. For example, the *BasicDelBook* operation generates the following proof related to: *dom*(*write*)=*Book*.

$$(dom(write)=Book \wedge book \subseteq Book) \Rightarrow$$
$$[Book:=Book\text{-}\ book \parallel num\_book:=book \lhd\ num\_book \parallel pageNb:=\text{book} \lhd\ pageNb \parallel write:=book \lhd\ write]$$
$$dom(write)=Book$$

which can be simplified further to:

$$(dom(write)=Book \wedge book \subseteq Book) \Rightarrow (dom(book \lhd\ write)=Book\text{-}\ book)$$

This proof is simply achieved by applying properties associated to the anti-restriction operator:

$$dom(book \lhd write)=dom(write)\text{-}book\ =Book\text{-}book$$

As we have already noted, the basic database operations are not very complex. Most proofs are based on the application of relational operator properties. All the basic machines are automatically proved using AtelierB [4].

3.2 State Diagrams

Data behavior is described by transactions triggered by a message called transaction message. At the analysis level, it is sufficient to describe the effect of the message on the objects of the system. It means to specify which basic operations are involved and under which conditions. These are described using UML state and collaboration diagrams. We begin by state diagrams, collaboration diagrams are discussed in 3.3.

a) Description

In UML, state diagrams may have different semantics, according to the kind of software which are modelled. In our case, we suggest that a state diagram is attached to a class. It describes the states of objects of the class. Each state is an abstraction of the value of attributes and/or links of an object. An event occurs if a certain condition, called a guard, is satisfied and triggers transitions on the object which is a parameter of the event. An action may be executed during a transition. All the transitions leaving the same state are either associated to different events or guarded by disjoint conditions. A generic representation of transitions triggered by an event *ev* is:

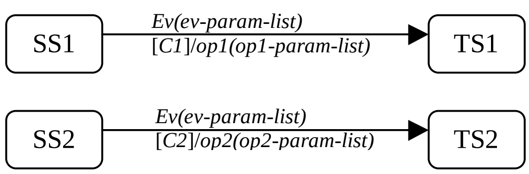

Figure 3. Transitions Triggered by the Event *ev*

where C_i and op_i are the guards and the actions respectively.

In order to make the generation more complete, some rules need to be followed :

- both states and guards are defined by predicates using B notations,
- actions involved in transitions correspond to calls of operation of the related class.

b) Translation into B

To each state diagram, attached to *Class*$_k$, an abstract machine *StateClass*$_k$ is created. It includes machine *BasicClass*$_k$ and may use others machines if necessary. Each state predicate is translated into a B definition. Each event is translated into a B operation which calls the relevant operations of the class. The state diagram of Figure 3 is translated as follows:

ev(ev-param-list) =
PRE $(SS_1 \wedge C_1 \wedge PC\_op_1) \vee (SS_2 \wedge C_2 \wedge PC\_op_2)$
THEN
Select $(SS_1 \wedge C_1)$ **Then** $Op_1(op_1\text{-}param\text{-}list)$
When $(SS_2 \wedge \mathrm{C}_2)$ **Then** $op_2(op_2\text{-}param\text{-}list)$
END
END

where:

- SS_i is the predicate of the state i,
- C_i is the guard of the transition i,
- PC_op_i is the precondition of the operation op_i,
- op_i is an operation of $Class_k$.

Recall that it has been already proved that each op_i establishes the invariant of its machine. As there is no additional invariant in this machine, there is no more proofs associated to op_i in this machine.

Each B operation associated to an event generates as many proofs as transitions triggered by the event. Each proof checks that the operation related to the transition is called under its precondition. Let j be the number of transitions triggered by ev. The proof associated to the i^{eme} transition is:

$$\vee_{p=1..j} (SS_p \wedge C_p \wedge PC\_op_p) \wedge (SS_i \wedge C_i) \Rightarrow PC\_op_i$$

If the guards are disjoint, the expression is simplified:

$$(SS_i \wedge C_i \wedge PC\_op_i) \Rightarrow PC\_op_i$$

In addition to these proofs, we must be sure that, for each transition, the object is in the target state after the execution of the operation. A proof obligation is thus automatically generated:

$$(SS_i \wedge C_i \wedge PC\_op_i) \Rightarrow [op_i]TS_i$$

TS_i is the predicate of the target state associated to the transition.

3.3 Collaboration Diagrams

In UML, collaboration diagrams are used at any level of the development process to describe how objects participate to accomplish a specific purpose such as a scenario or the implementation of an operation. At the analysis level, we suggest to use them in a very simplified way: if a transaction message concerns several classes, a collaboration diagram elaborates the decomposition of this message into internal messages on each class concerned.

a) Description

Message exchanges must be from a special class System to objects of the class diagram. An internal message is either a call to an operation of a class or a state diagram event. A generic collaboration diagram can be represented as in Figure 4.

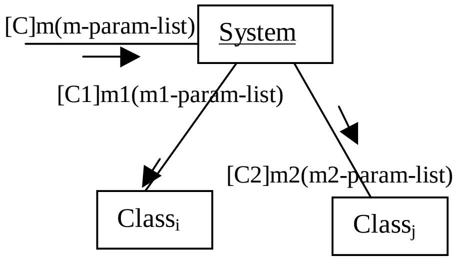

Figure 4. A Generic Collaboration Diagram

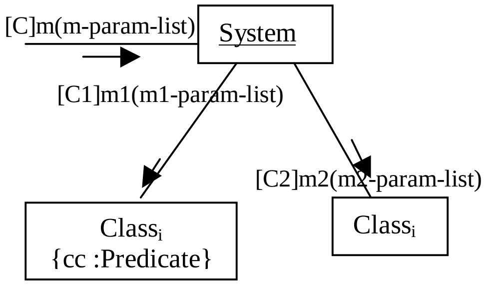

Figure 5. A Specific Case of Collaboration Diagram

m(…) is the transaction message and m_i(…) are internal messages.

Each message may be guarded. The guard C of the transaction message has the same meaning as a guard in state diagrams: the message may occur only if the guard condition holds. The meaning of guards related to internal messages is quite different: if the guard is true then the message is sent otherwise nothing is done.

In some cases, the objects on which the internal messages act, are dynamically determined during the transaction message processing. For this, we propose a special notation(See Figure 5). The semantics of this notation is: when the message m occurs, System takes an object (or a set of objects) cc of $Class_i$ satisfying a given condition Predicate and then, depending on the guards C1 and C2, the messages m1 and m2 are sent or not. Of course, cc might be a parameter of m_1 and/or m_2.

b) Translation into B

Collaboration diagrams are all gathered and translated into one abstract machine called *System* which includes all the state diagram machines and the basic machines corresponding to classes which have not an associated state diagram. There is one B operation for each transaction message. Figures 4 and 5 are automatically translated into the following operations:

```
m(m-param-list) =
PRE     C ∧ PC_m1 ∧ PC_m2
    IF C1 THEN m1(m1-param-list)  END||
    IF C2 THEN m2(m2-param-list)  END
END
```

```
m(m-param-list)
PRE  C ∧ PC_m1 ∧ PC_m2
 ANY  cc WHERE Predicate THEN
  IF  C1 THEN m1(m1-param-list) END ||
  IF C2 THEN m2(m2-param-list) END
 END
END
```

where PC_m_i is the precondition of the operation associated to the internal message m_i. As for state diagrams, an automatic translation is possible only if all the conditions of messages are written in B notations. Proof obligations are generated by the B prover, which ensure the consistency of the invariants of all the related machines. Establishing these proofs allows the consistency of the whole application to be checked.

Let us explore in more details the proofs generated for each operation specified in the *System* machine:

- Proofs checking the precondition of a called operation m_i:

$$C \wedge (\wedge_{p=1..2}\ PC\_m_p) \Rightarrow PC\_m_i$$

Notice that these proofs are all trivial. Indeed, the goal is included in the premises.

- Proofs related to the invariant of the System machine: we must ensure that the execution of the operation m establishes the invariant of the machine. The generated proof is of the form:

$$C \wedge (\wedge_{i=1..2}\ PC\_m_i \wedge INV) \Rightarrow [||_{i=1..2}\ (IF\ C_i\ THEN\ m_i\ END)]\ INV$$

For each possible combination of values of C_i (true or false) a proof is generated. The number of the proofs is thus equal to 2^k(k is the number of guards).

- Proofs related to the invariant of the included machines: in B, if an invariant of a given machine A contains variables of used machines, no proof is generated. The checking of this invariant is postponed into the machine which includes A. So, in the System machine , we check the invariants of the basic machines which are constructed upon shared variables.

c) Discussion

By applying the rules defined above, we always obtain the same three level generic architecture of the formal specification which reflects the three UML description views. Each machine corresponds to a unique component of the analysis UML model: either a class for the basic machines or a state diagram for a state machine or all the collaboration diagrams for the system machine. Graphically, it is represented in Figure 6:

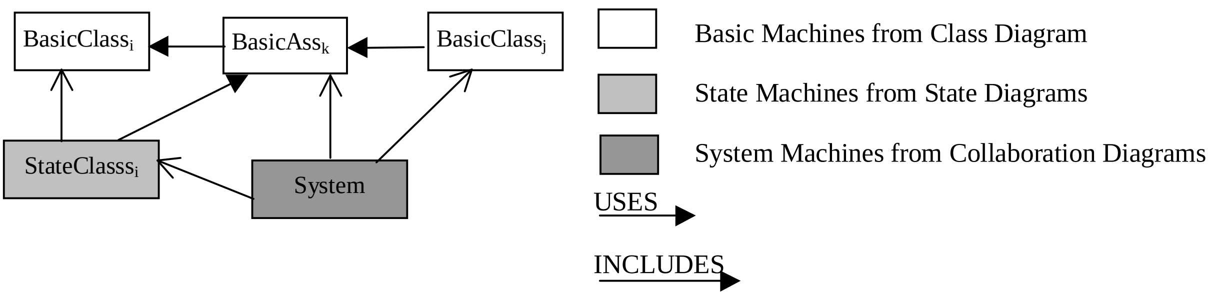

Figure 6. The 3–levels Architecture of the B Specification

Notice that when inheritance and composition concepts will be considered, the architecture will certainly contain more levels to reflect the different abstraction levels of class diagrams.

This architecture makes easier the understanding of the formal specification with respect to the informal one and improves evolution of the system. Moreover, it allows a best localization of errors detected in the proof phase. Indeed, a bijection may be defined between the UML and the B elements. Thus, if a failure occurs during the proof phase it is possible to point out the relevant UML element. For example, if the proof corresponding to an operation of a state diagram fails, the relevant transition may be exhibited because one proof is attached to one transition. In other words, this architecture permits a best debugging of the specification.

Formal annotations in state and collaboration diagrams are expressed in B rather than OCL which is the "formal" language used in UML to describe constraints. In fact it is only partially formalized and no support tools exist.

Up to now, general constraints on data have not been considered for the generation of basic update operations. In the UML toolkit, they could be specified in the documentation field of the associated element using B or OCL languages. In the current solution and if they are formally described, new invariants are added in the relevant state machines or system machine, depending on the classes involved in the constraints. Therefore proving operations of these machines ensures the constraints are satisfied. An interesting work, as done in [13],would consist of finding for the relevant operation the condition, induced by the constraints, to be added to its precondition such that this operation may be called safely.

4. Automatic Translator

Thanks to the precise definitions we have proposed to enrich UML and the rules described above, an automatic translator can be built. It is implemented in OCaml language [14] (the Object French version of ML language) which is well suited for implementing translation between different languages.

4.1 Functional Architecture of the Tool

The derivation of B specifications from different UML diagrams is performed into six steps (Figure 7):

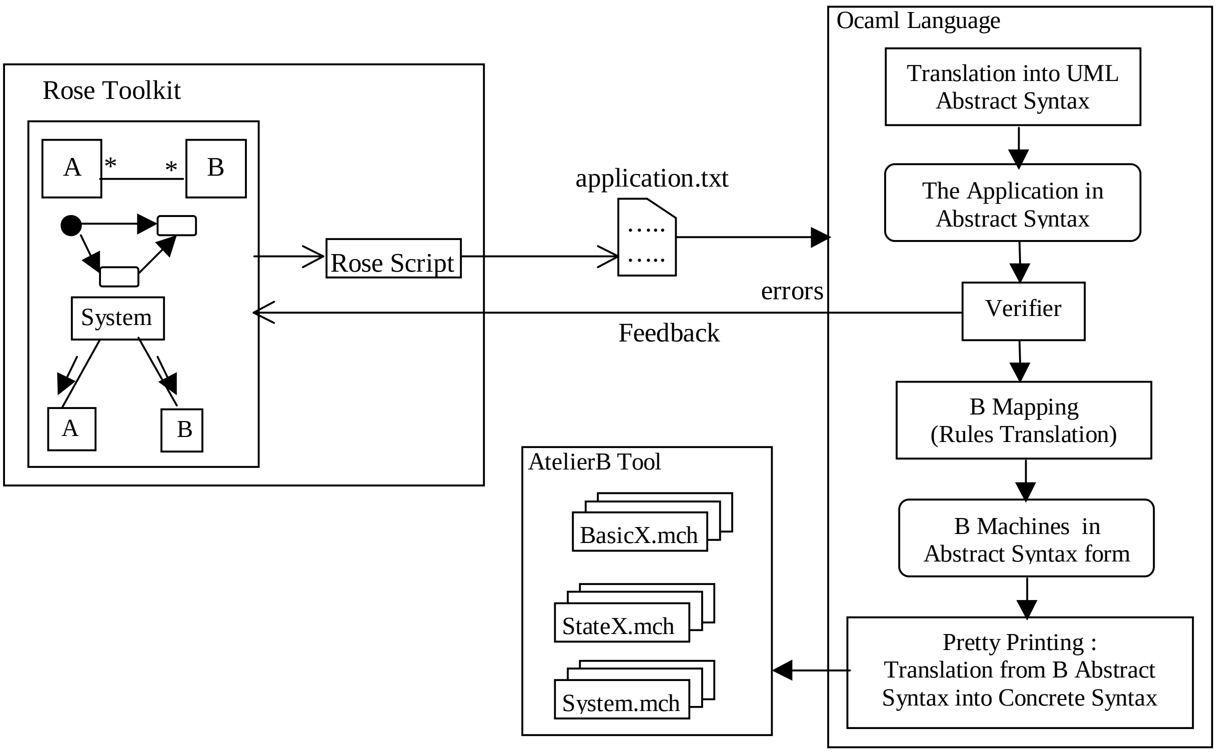

Figure 7. From UML Models to B Specifications

1. The diagrams are edited under the Rose environment. Information not graphically expressed, such as annotations in state diagrams, is put (in B notations) in the documentation field of the relevant element.
2. From these diagrams, a Rose script produces a textual form (because it is not possible to call a OCaml function from the Rose environment).
3. A syntactic analysis generates from text files the abstract syntax Consistency checks between diagrams are carried out by the verifier. It generates a feedback to the designers when errors are detected. Up to now this

feedback is not automated, just an error message is reported. This step is the core of the process. It consists of applying the set of translation rules in order to of UML elements.
4. translate the UML abstract syntax into B abstract syntax.
5. The pretty printing of B abstract syntax gives B machines, suitable for rigorous analysis in AtelierB.

Let us detail the role of the verifier. It allows us to check part of the consistency inside a diagram but also between the different diagrams. For example, the following constraints can be verified:
- each key of a class is composed of attributes of the class.
- each action mentioned in a state diagram corresponds to an operation of the considered class.

The complete consistency of the whole application will be verified when proofs of the generated B machines will be achieved.

4.2 Implementation

The prototype support tool covers the last three steps of the derivation of B specification from UML diagrams. In order to build a reliable tool, a formal semantics for UML models has been defined [12]. The basic semantics is specified by metamodels (as in UML [17]), expressed either in a diagrammatic form or with B notation sets (see Figure 8). Additional semantics used to clarify the constraints among metamodel components are specified in the form of B invariant clauses.

For example, an additional constraint in the metamodel would specify that the attributes of a key belong to the same class as the key.

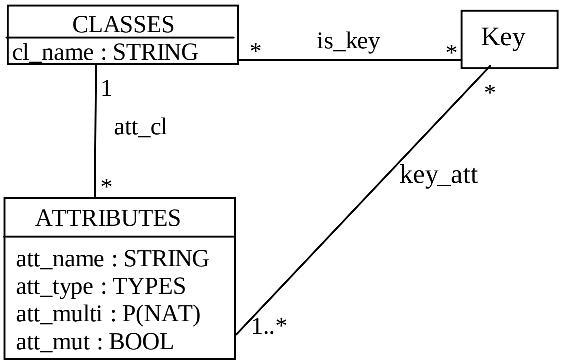

Figure 8.Part of the UML Class Diagram Metamodel

Using these formal metamodels, the tool consists of four logical parts:
a) An abstract syntax of UML: each kind of diagrams is described in an abstract syntax form expressed in OCaml. The OCaml representation is a straightforward translation of the relevant formal UML metamodel. For example, the metamodel of Figure 8 is translated by:

// the names of classes or attributes
type name= *Var* of string
// the type of an attribute
type att_type= Nat| String| Bool|...;;
// only the common multiplicities are given
type att_multi=Zero_One|Zero_N| One_One| One_N|...;;
// the value may be changed or not
type att_mut= True| False;;
// the definition of an attribute
type attribute= *Att* of name* att_type * att_multi * att_mut;;
// the structure of a key
type key= *Key* of attribute list;;
// the definition of a class
type class= *Class* of name* key list* attribute list

For example, the Book class is represented by:
Class(*VAR* "Book",
[*Key*[*Att* (*VAR* "num_book", String,One_One, False)]]
[*Att* (*VAR* "num_book", String,One_One, False); *Att*(*VAR* "pageNb", Nat,One_One, False)])

b) An abstract syntax of B language: it represents the structure of a B machine and its different clauses. The abstract syntax of a B machine is of the form:

```
type B_Machine= Machine of
                    name*
                    Uses of name list*
                    Includes of name list*
                    Sets of name list*
                    Variables of name list*
                    Invariant of predicate list*
                    Initialisation of substitution list*
                    Operation of operation list;;
```

c) A set of checking functions: these functions implement the verifier module of the tool. For example, the above constraint on keys and attributes of a class is checked by a function Verify_Key. This function takes two parameters key and Att representing respectively the list of keys and attributes associated to the class:

```
let rec Verify_Key key Att =
                match key  with
                [] → True |
        (Key Att1)::key' →(intersect Att1 Att=At1) and (Verify_Key key' Att);;
```

where **intersect** is a built-in OCaml list function.

This rule means: the attributes of the first key (head of the list) *Key* Att_1 are attributes of the related class and then we repeat the same verification on the rest of the list (key').

According to the three kinds UML diagrams we consider, the tool provides three levels of verification:

- Class Diagrams: to check the consistency of a class diagram itself.
- State Diagrams: to check the consistency of a state diagram itself and also with respect to the associated class diagram.
- Collaboration Diagrams: to check the consistency of a collaboration diagrams itself and also with respect to the state and class diagrams.

d) A set of generating functions: each translation rule is implemented by a OCaml function working only on the two abstract syntax. For example, the rule associated to the mapping of a class is:

```
let translate_class Class(Var  Cl_name)=
                Machine(
                VAR BasicCl_name,
                Sets [ Var (uppercase Cl_name)],
                Variables Cl_name,
                Invariant  Inclus(Var Cl_name, Var(uppercase Cl_name)))
```

Where **uppercase** is a built-in OCaml function. *Inclus* is the abstract form of the set inclusion operator of B. The obtained B concrete syntax corresponds to the first B notations described in Section 3.1.b

4.3 Benefits and Limits of the Prototype Tool

The tool permits the automatic generation of B specifications of an application. The main benefits are:

- the inputs and outputs of the tool are represented in abstract syntax form, thus the tool is independent of toolkits for UML and B,
- the generated machines are directly usable in the AtelierB tool. Thus the proof phase can be directly started in order to establish the correctness of the specifications,
- OCaml has allowed a rapid development of the prototype and the resulting code is rather concise: 1800 lines are enough for the automation of all diagrams. Furthermore, the OCaml functions are a straightforward translation of the derivation rules.

Nevertheless, up to now our tool is not complete. Indeed, some deep semantic analysisare missing on the diagrams. For example, the predicate of a source state can be always false or contradictory which implies that precondition of the generated B operation is always false and thus the associated proof theorem is discharged(always true). Such bad specification can't be detected by the B prover and so another kind of checker is needed. This kind of properties correspond to liveness properties. The fail of the B prover to detect such error isn't a limit of the B method. Indeed, the B method aims at developing systems such that if they run then we are sure they give good results. This corresponds to safety properties.

5. Conclusions

In this paper, we have presented a prototype tool which generates B specifications from an application described using well-defined UML notations. The obtained B machines are directly usable in AtelierB. Thus proofs can be performed. In general, it is not the easier activity of the specification. AtelierB generates proof obligations and try to automatically demonstrate them. In case of failure, proofs must be achieved manually. In the context of database applications, it turns that proof obligations are often long but simple to demonstrate. Indeed, on one hand, such applications are caracterised by a great amount of inter-related data (variables and invariant conjuncts), but in other hand, theirs operations are rather relatively simple and generic. Thus the definition and the reusability of proof schemas enable to reduce the cost of the proof phase. In the student projects and case studies we have carried out [12], [16], more than 90% of proofs was automatically achieved. The exceptions of the automatic prover are mainly due to the lack of rules in the prover rule database, concerning some operators and also the lack of proof tactics. We think this last point could be improved by taking into account the specific context of our approach.

Further works include :

- extending the tool to automatically generate database implementations (data and programs). We have already defined a set of generic rules for relational implementations [11], based on the refinement process of B. Thus the design and coding phases will be largely automated.
- extending the subset of well-defined UML concepts by considering inheritance and aggregation/ composition.

References

[1] J. R. Abrial. The B-Book. Cambridge University Press, 1996.
[2] J. Armstrong. Industrial Integration of Graphical and Formal Specification. Journal of Systems and Software, 40(3), March 1998.
[3] J. M. Bruel and R.B. France. A Formal Object-oriented CASE Tool for the Development of Complex Systems. 7th European Workshop on Next Generation of Case Tools, Crete, Greece, May 1996.
[4] DIGILOG groupe STERIA. AtelierB, Manuel de Référence. DIGILOG, BP 16000, 13791 Aix-en-Provence Cedex 3 France, 1996.
[5] S. Dupuy, Y. Ledru, and M. Chabre-Pecoud. An Overview of Roz: a Tool for Integrating UML and Z Specification. 12th Int. Conf CAISE'00, Stockholm, Sweden, June 2000.
[6] A. Evans and S. Kent. Core Meta-Modeling Semantics of UML: The pUML Approach. 2nd Int. Conf. On the UML, LNCS 1723, Springer-Verlag, Colorado, USA, 1999.
[7] P. Facon, R. Laleau and A. Mammar. Combining UML with the B formal Method for the Specification of Database Applications. Research Report, CEDRIC Laboratory, Paris, September 1999.
[8] P. Facon, R. Laleau and P. Nguyen. Mapping Object Diagrams into B Specifications. In M. Frappier and H. Habrias, editors, Software Specification Methods. Springer, 2000.
[9] R.B. France, A. Evans, K. Lano, and B. Rumpe. The UML As a Formal Modelling Notation. Computer Standards and Interfaces, 19, 1998.
[10] T. Günther, K.D. Schewe and I. Wetzel. On the Derivation of Executable Database Programs from Formal Specifications. Int Symp. FM'93, Odense, Denmark, April 1993.
[11] R. Laleau and A. Mammar. A Generic Process to Refine a B Specification into Relational Database Implementations. Int. Conference ZB2000, LNCS 1878, Springer-Verlag, York, UK, September 2000.
[12] R. Laleau and F. Polack. Metamodels for Static Conceptual Modelling of Information Systems. Workshop "Defining Precise Semantics for UML", ECOOP 2000, Cannes, France, June 2000.
[13] Y. Ledru. Identifying Pre-conditions with the Z/EVES Theorem Prover. 13th IEEE Automated Software Engineering Conf, 1998.
[14] X. Leroy. The Objective Caml System, Documentation and User's Manual. INRIA, France, 1999.
[15] K.C Mander and F. Polack. Rigorous Specification Using Structured Systems Analysis and Z. Information and Software Technology, 37(5), 1995.
[16] P. Nguyen. Dérivation de Spécification Formelles B à partir de Spécifications Semi-Formelles. PHD thesis, CEDRIC Laboratory, Paris, France, Decembre 1998.
[17] OMG. Unified Modelling Language Specification. BetaR1 Release, available on line, www.rational.com/uml, April 1999.
[18] J. Peckam, Semantic Data Models. ACM Computing Survey, Vol 20, N°3, September 1998.
[19] Relational Software Corporation. Using Rational Rose 98. Rational Rose, 1998.
[20] D. Spelt and H. Balster. Automatic Verification of Transactions on an Object-Oriented Database. 6th Workshop on Database Programming Language, 1998.