Translating B machines into UML diagrams

Houda Fekih, Stephan Merz

Translating B machines into UML diagrams

Houda Fekih
Faculté des Sciences de Tunis
Universit de Tunis
Tunis, Tunisia
Houda.Fekih@fst.rnu.tn

Stephan Merz
INRIA Lorraine
LORIA
Nancy, France
Stephan.Merz@loria.fr

December, 2003

Abstract

This report describes transformations of B abstract machines into UML class and state-transition diagrams. The basic goal of this work is to produce translation rules for deriving UML class diagrams and state machines from sets, variables and operations in a B model. Our translation is interactive and does not necessarily produce a unique UML model, but it takes into account the use of the elements in the B model as a whole to constrain the translation into UML. We also consider how the refinement relationship of B machines maps to relations between the resulting UML models.

Keywords: UML, class diagram, state-transitions diagram, B, refinement

1 Introduction

The Unified Modeling Language [4] (UML) is nowadays the de-facto standard language used for the design of object-oriented software and systems. It is semi-formal, because it has a formally defined syntax, but no formal semantics, but it has been found extremely useful to specify, visualize, and document models of software systems. Offering various types of diagrams to describe static, dynamic, and architectural aspects of systems at different levels of abstraction and during different phases of system design, it is particularly appropriate for communicating ideas between clients and system engineers.

B [2] is a formal method and language for constructing and proving mathematical models of systems. It emphasizes a process of successive refinements from the abstract specification to the concrete model of the system. The event-driven B approach [1] is based on the B notation. It extends the methodological scope of basic concepts of the B method such as generalized substitutions. In this formalism, a formal model is described by a (finite) set of *state variables* that are modified by a (finite) set of *events*; an invariant I states some properties that must always be satisfied by the variables and that must be *maintained* by the execution of the events. An event essentially consists of two parts: a *guard*, which is a predicate built from the state variables, and an *action*, written as a generalized substitution. The refinement of a formal model allows to enrich a model in a *step by step* approach. Refinement provides a way to strengthen invariants and also to add details to a model. It is also used to transform an abstract model

into a more concrete version by modifying the state description. More precisely, a refinement may be based on a different set of variables; a glueing invariant formalizes the relationship between the two state representations. Moreover, events that exist in the abstract model may be refined, and new events may be added, provided they do not modify the high-level state.

The rôles of the two languages are therefore complementary: UML is more accessible to the untrained user and provides a rich array of concepts that clarify the structure of a system. The B method, on the other hand, is based on relatively few and precisely defined mathematical concepts, and is very adequate for formal verification. This distinction is also reflected in existing tool support: tools for UML are centered around graphical editors, they often allow for simulation and code generation, but otherwise offer rather limited (mostly syntactic) analysis techniques. The B method is supported by tools such as Atelier B [12] and B-Toolkit [7] that prominently include a theorem prover to ensure the correctness of a development. It therefore appears desirable to be able to use both languages in a single development. Ideally, one would be able to seamlessly go back and forward between B and UML, using whatever language is more convenient for the problem at hand.

Previous work [9, 8] has mostly focused on formalizing UML models in formal methods such as B, aiming at the verification of UML designs and, consequently, at the elimination of inconsistencies and ambiguities. One problem with this approach is that such translations, aiming to be as comprehensive as possible, tend to result in unnatural and cluttered B specifications that are hard to understand and reason about. Another problem is the traceability of errors detected at the B level back to the UML model.

Although perhaps less natural at first sight, we believe that it is also interesting to derive UML models from B specifications. For example, the additional structure afforded by UML can help to clarify the B specification better than the "flat" set-theoretic language on which B is based. Moreover, users of the B formalism could have access to simulation and code generation provided by B tools. (Atelier B also includes a module for code generation, but only for low levels of specification, which may be less natural than, say, a UML state machine.) There has been less work on this question, with the exception of Tatibouët's work on the B2UML tool [11]. Unfortunately, this work imposes rather stringent assumptions (e.g., a B specification may represent only a single class) that do not seem to be met in practice. In this context, we can cite CEDRIC-IIE work [10, 5] which focuses specially in derivating formally relational database implementations from the B specification. CEDRIC-IIE outline in citelal the needs for a tool to assist in IS development and propose global meta-structures used in linking IS UML concepts without giving translation rules or a fully-conformants IS UML metamodels.

In this paper, we present a first step in this direction. Because UML offers many concepts (e.g., classes, attributes, associations, generalizations, compositions, states, and transitions) that do not have a direct counterpart in B, we do not believe that it is reasonable to hope for a fully automatic translation. Rather, the user will have to intervene to indicate the rôle of different entities such as constants and variables that appear in the B specification. On the other hand, we have identified a number of possible alternatives, as well as guidelines that help choosing between them.

Concretely, we propose a method to interactively produce UML diagrams (focusing on class and state-transition diagrams) from abstract machines of the B notation. The diagrams are complemented by annotations expressed in the Object Constraint Language [14] (OCL) that specify invariants on classes, attributes, and associations, and to describe pre- and post-conditions on operations and methods. We also discuss how the mapping interacts with a development of a B model across successive refinements.

MACHINE Building
SETS
$BUILDING; PERSON$
CONSTANTS
aut, com
VARIABLES
sit
INVARIANT
$aut \in PERSON \leftrightarrow BUILDING \wedge$
$com \in BUILDING \leftrightarrow BUILDING \wedge$
$sit \in PERSON \rightarrow BUILDING \wedge$
$sit \subseteq aut \wedge com \cap \mathrm{id}(BUILDING) = \{\}$
OPERATIONS
$pass \mathrel{\widehat{=}}$ ANY p, b
WHERE $(p,b) \in aut \wedge (sit(p), b) \in com$
THEN $sit(p) := b$
END
END

Figure 1: Abstract specification of access control.

Longer-term goals of this work are to arrive at an integration of both types of languages and methods and to define a refinement relationship on UML models, inspired from the concepts that have already been developed in the B method.

2 Example : access control

We illustrate our approach to translating B models into corresponding UML diagrams at the hand of the well-known specification of a system to control the access of persons to buildings. This example develops and illustrates many translation rules that will be introduced more formally in section 3. Following the philosophy of the B method, we consider two specifications of access control: the first, abstract model introduces the basic entities, and moves between buildings are modeled as atomic actions. The second model refines the first one by introducing the concepts of doors and authentification. For both models, we first focus on the static aspects of the models, introducing UML classes, their attributes, and associations, before moving on to consider dynamic aspects such as operations, states, and transitions.

2.1 Abstract model

Figure 1 reproduces the abstract specification of the access control system [3]. It introduces two abstract sets *BUILDING* and *PERSON*, that represent the basic entities of the model. Next, the model declares two constants *aut*, *com*, and a variable *sit* whose types are given in the *INVARIANT* clause. The relation *aut* indicates which persons are allowed to enter which buildings. Similarly, the relation *com* models which buildings communicate. Finally, *sit* associates to each person the building he or she is currently in, mathematically modeled as a (total) function. The remaining conjuncts of the invariant assert that at each state, persons can only be in buildings they are allowed to

enter and that *com* is irreflexive (i.e., no building communicates with itself). The model introduces a single operation (or events) *pass* that represents a move of a person *p* to a building *b*, provided that *p* is allowed to enter *b* and that *b* can be accessed from the building that *p* is currently in.

We will now develop a UML model corresponding to the B specification of Fig. 1, initially concentrating on deriving a class diagram. When encountering abstract sets such as *BUILDING* and *PERSON*, there is a choice in UML between representing them as basic types or as classes, and we inspect the use of such sets in the B specification in order to help us resolve this choice. For example, consider the set *PERSON*: it appears as the domain of the relation *aut* and of the function *sit*, indicating that it "governs" relationships with other entities. This observation suggests that *PERSON* represents a class. Similarly, *BUILDING* appears as the domain of the relation *com* and will therefore also be translated into a class. On the other hand, a set that occurs only as the range of functions or relations in the B specification can be represented in UML as either a type or a class.

Next, we consider the variable *sit* and the constants *aut* and *com*. Variables are the only means offered by the B language to describe entities whose values may change during system execution. Constants describes entities whose values can not be modified. In UML, we have a large choice of distinct concepts that variables and constants can be mapped to, including associations, attributes or states.

Again, the invariant clause of the B specification offers some clues on how to resolve this choice. For example, *aut* describes a relation between two sets that we have decided to represent as classes; we therefore find it most natural in UML to model *aut* as an association of multiplicity (*,*) between these classes. It would also be possible to represent *aut* as a set-valued attribute in either class *Person* (indicating the set of building the person is allowed to enter) or *Building* (describing the set of persons that are allowed to enter the building). Analogously, we will represent *com* as a reflexive association of multiplicity (*,*) of class *Building*, although it could also be represented as a set-valued attribute of that class, with two different interpretations. Constraints will be added to *com* and *aut* associations. The value of these constraints is "frozen" and will be drawn in braces in the UML class diagram. It asserts that once a link *com* or *aut* is added, it cannot be modified.

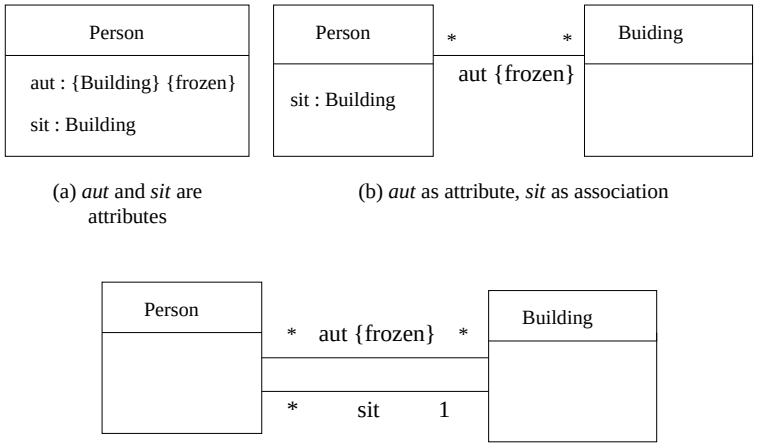
The variable *sit* represents a function whose domain and codomain are again represented by classes in the UML model. Using similar reasoning as above, it can be represented as either an association of multiplicity (*,1) or as an attribute of type *Building* in class *Person*. Figure 2 summarizes some of these choices. For the remainder of this paper, we choose the model shown in Fig. 2(b). Since *sit* is variable, its value may change over the run of a system; it is a "changeable" attribute. Since this is the default in UML, no declaration needs to be added.

We have now successfully identified the classes, associations, and attributes of a UML model corresponding to the original B specification. The conjuncts of the invariant that have not been expressed as the types of attributes and associations can be translated as OCL constraints, as follows:

context *Person* **inv :**
$self.aut \rightarrow includes(self.sit)$

context *Building* **inv :**
$not(self.com \rightarrow includes(self))$

We now turn to the operation *pass* declared in the B specification of Fig. 1. In gen-



(a) *aut* and *sit* are attributes

(b) *aut* as attribute, *sit* as association

(c) *aut* and *sit* are associations

Figure 2: Alternative representations of *aut* and *sit*.

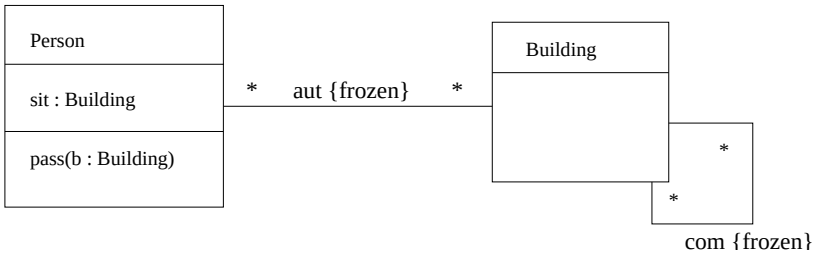


Figure 3: Class diagram for the access control system.

eral, operations will be mapped to methods of classes. They may give rise to transitions between states in UML state machines, and may be further described by interaction diagrams, in particular when instances of several classes are affected by the operation. In our case, we observe that there are two participating objects, a person p and a building b (the types are given implicitly by the pre-condition $(p,b) \in aut$). The effect of the operation is to update the function *sit* at argument position p, represented in UML as an attribute of class *Person*. Therefore, it appears most natural to map the operation *pass* to an instance method of class *Person*, taking a parameter b of type *Building*. The method can be described as follows, based on OCL's notation for pre- and post-conditions:

context $Person :: pass(b : Building)$
$pre : self.aut \rightarrow includes(b)$ *and* $self.sit.com \rightarrow includes(b)$
$post : self.sit = b$

Summing up, we obtain the UML class diagram of Fig. 3 to represent the B specification of the access control system shown in Fig. 1. This first B model does not contain enough structure to derive a non-trivial state machine. We will therefore delay our discussion of that aspect of translation to the refined B specification of the same system discussed in the following section.

2.2 A refined specification of the access-control system

Figure 4 introduces a refinement of the access control system. Specifically, it explicitly models (one-way) doors, represented by the set *DOOR*, that connect the buildings, and that must be unlocked prior to a person moving from one building to another one. The functions *orig* and *dest* associate the origin and destination building with each door; the relation *com* of the previous model can now be computed from these functions as follows :

context *Door* **inv :**
$self.orig.com \rightarrow includes(self.dest)$

Before a person can use a door to move between two buildings, it must be unlocked for that person. The variable *unl* records which door, if any, is currently unlocked for a given person. Its value is a partial injection, expressing the idea that at any time at most one door can be unlocked for a person and, conversely, a door can be unlocked for at most one person. As long as a door is unlocked, a green indicator is lit at that door. The set *green* of the B specification is simply defined as the range of the function *unl* and contains the doors that are currently unlocked. On the other hand, a red light indicates that a person is not allowed to pass through the door; the subset *red* of *DOOR* contains the doors whose red indicator is lit. This description implies that the red and green indicators can never both be lit for a door, hence the invariant asserts the sets *red* and *green* to be disjoint. The invariant moreover asserts that a door will only be unlocked for a person situated at the origin of the door, and that the person must be authorized to enter the destination building.

The refined model redefines the event *pass* that was already present in the abstract model. It is interesting to note that the new event takes only one formal parameter q representing an unlocked door, since the person and the destination building can be inferred via the functions *unl* and *dest*. Besides modeling the person crossing the door, the event also changes the function *unl* by removing all pairs of the form (p,q). In particular, the green indicator is no longer lit.

The model also adds a number of new events, which (in line with the requirements of the B method) do not change any of the variables of the abstract model. The events *unlock* and *refuse* model the attempt of a person to unlock a given door. This can only be successful if the person is allowed to cross a door, as defined by the predicate *admitted*: the person is currently in the origin building of the door, there is no other door unlocked for that person, and the person is authorized to enter the destination building. These events are responsible for lighting the green or red indicators. The remaining events *lock* and *free* model the spontaneous locking of an unlocked door and the reset of a red indicator. Presumably, these events are intended to be activated after the green or red indicators have been lit for suitable timeout periods, but this cannot be expressed formally at this level of abstraction, which does not include real time.

We now consider how to transform this B specification into a corresponding UML model, at first again focusing on the static system model. The newly introduced abstract set *DOOR* occurs as domains of functions. Following our argumentation of section 2.1, it will therefore be represented by a UML class *Door*. Similarly, the functions *orig* and *dest* will be represented by "frozen" attributes of type *Building*. Concerning the translation of *unl*, the type invariant tells us that it is a partial injection from *PERSON* to *DOOR*, both of which sets are represented by classes. In UML, we have a choice between representing this function as an association of multiplicity ((0..1),(0..1)) be-

REFINEMENT Building1
REFINES
Building
SETS
$DOOR$
CONSTANTS
$orig, dest$
VARIABLES
unl, red
DEFINITIONS
$admitted(p,q) \mathrel{\widehat{=}} orig(q) = sit(p) \wedge p \notin \mathrm{dom}(unl) \wedge (p, dest(q)) \in aut$
$green \mathrel{\widehat{=}} \mathrm{ran}(unl)$
INVARIANT
$orig \in DOOR \rightarrow BUILDING \wedge$
$dest \in DOOR \rightarrow BUILDING \wedge$
$unl \in PERSON \rightarrowtail\!\!\!\!\!\!\rightarrow DOOR \wedge$
$red \subseteq DOOR \wedge green \cap red = \{\} \wedge$
$com = (orig^{-1}; dest) \wedge (unl; orig) \subseteq sit \wedge (unl; dest) \subseteq aut$
OPERATIONS
$pass \mathrel{\widehat{=}}$ ANY q
WHERE $q \in green$
THEN $sit(unl^{-1}(q)) := dest(q)$
$unl := unl \rhd \{q\}$
END
$unlock \mathrel{\widehat{=}}$ ANY p, q
WHERE $p \in PERSON \wedge q \in DOOR \wedge$
$q \notin green \cup red \ \wedge admitted(p,q)$
THEN $unl := unl \cup \{p \mapsto q\}$
END
$refuse \mathrel{\widehat{=}}$ ANY p, q
WHERE $p \in PERSON \wedge q \in DOOR \wedge$
$q \notin green \cup red \ \wedge \neg admitted(p,q)$
THEN $red := red \cup \{q\}$
END
$lock \mathrel{\widehat{=}}$ ANY q
WHERE $q \in green$
THEN $unl := unl \rhd \{q\}$
END
$free \mathrel{\widehat{=}}$ ANY q
WHERE $q \in red$
THEN $red := red - \{q\}$
END
END

Figure 4: Refinement of abstract specification of access control.

tween these classes, or as possibly null-valued attributes in either class *Person* or *Door*. Invariants related to *unl* will have to be expressed by an OCL constraint as follows :

context *Person* **inv :**
if $self.unl \rightarrow notEmpty$
then $(self.sit = self.unl.orig)$ *and* $(self.aut \rightarrow includes(self.unl.dest))$

The representation of the sets *green* and *red* is more interesting. Both are subsets of *DOOR*, which we have already identified as a class. Set inclusions of the form $e \subseteq C$, where C represents a class, can have a number of different interpretations. For example, e could denote a subclass of C provided that e itself denotes a class. It could also represent the set of existing instances of class C; in fact, Meyer [9] and Ledang [8] systematically introduce such variables when translating from UML to B. Observing that both sets are changing over time and that they are required to be disjoint by the invariant, the most natural interpretation in our example is to take these sets as denoting distinct states of an object. We therefore propose to introduce an attribute *state* of class *Door* that can take values in the set $\{green, red, neutral\}$, the third value modeling doors neither of whose indicator is lit. (Formally, the introduction of a third value is justified by the absence of a conjunct asserting that the union of the sets *green* and *red* equals *DOOR*.) Because *green* is a defined entity in the B model, this introduces a redundancy in the UML model, which can be expressed by an OCL class invariant asserting that the state of a door is green if and only if there is a person related by the *unl* relation.

We introduce a further redundancy by explicitly introducing an association *admitted* of multiplicity (*,*) between the classes *Person* and *Building* in the UML model. Another choice would be to represent *admitted* as an operation in *Person* or *Building* class. The OCL expression corresponding to the *addmited* association invariant can be expressed as follow :

context *admitted* **inv :**
$self.Door.orig = self.Person.sit$ *and* $not(self.Person.unl \rightarrow isEmpty)$
and $self.Person.aut \rightarrow includes(self.Door.dest)$

We now turn to the translation of the operations of the B specification. Again, we follow the idea to represent an operation by a method located at the class whose attributes are modified by the operation. (In general, an operation may modify the attributes of several classes. It may therefore be associated with an association or even with a set of classes at this level of abstraction.) Therefore, we still choose to map the *pass* operation to a method of class *Person* rather than of class *Door*. The pre-condition of *pass* is strengthened in this refinement. The new *pass* method can now be described as follows, in OCL's notation :

context $Person :: pass(q : Door)$
pre : $q.state = \#green$
post : $q.unl.sit = q.dest$
$self.unl \rightarrow excluding(q)$

All the other operations are mapped to methods of class *Door*. The class diagram shown in Fig. 5 summarizes the transformations discussed so far.

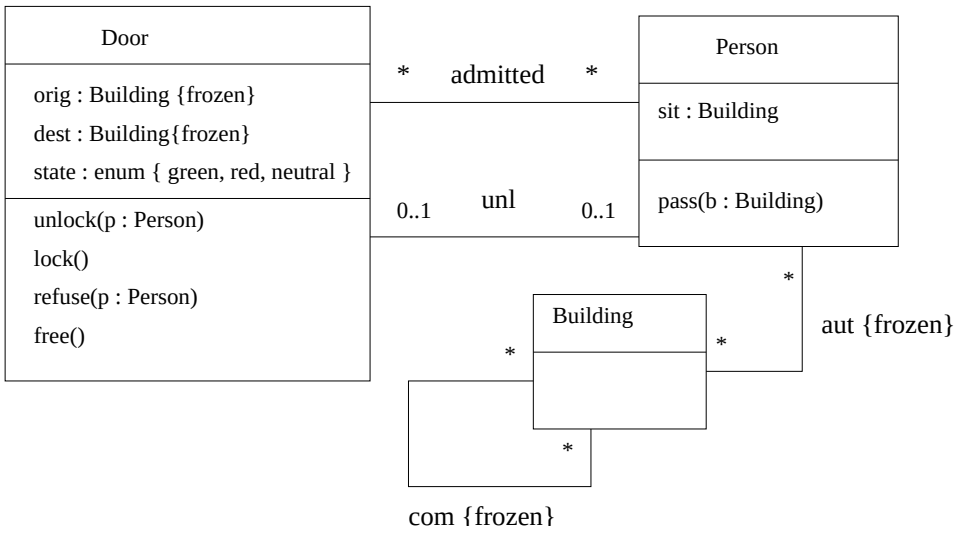


Figure 5: class diagram

The refined B specification of Fig. 4 contains enough information to produce a non-trivial state machine for instances of class *Door*. We have already found that a door can be in one of the three possible states *neutral*, *green*, and *red*, so it only remains to identify the transitions between these states. The syntactic presentation of events, consisting of a guard and a generalized substitution, helps us to identify the source and target states of the associated transition(s) as well as their activation guards. For example, the guard of the operation *unlock* requires the door to be in its neutral state and imposes the additional condition that person p (the parameter of the method) be admitted for the door. The body of the operation clearly establishes the postcondition $q \in green$, identifying the target state of the transition. Applying similar reasoning to the remaining methods of class *Door*, we arrive at the state machine of Fig. 6, which nicely summarizes the possible behavior of a door.

3 Rules for translation

We have applied similar transformations to the remaining refinements of Abrial's model of the access control system [3], as well as to several other case studies that are available in the literature. According to this experience, we do not believe that it is reasonable to attempt to a single, mechanized translation procedure for translating B into UML. Human assistance will be necessary in order to guide the translation towards obtaining models that are natural for both formalisms. Some decisions will have to be justified by theorem proving that will not generally be supported by automated reasoning. Decisions must take into account the entire B specification and they are mutually dependent. However, we have found that there exist a number of guidelines to identify the available alternatives and to help in choosing one or another, and that are embodied in the following rules.

- Rule 1 : an abstract set T that occurs as the domain in relations with other sets in a B machine is very probably a class *T*.

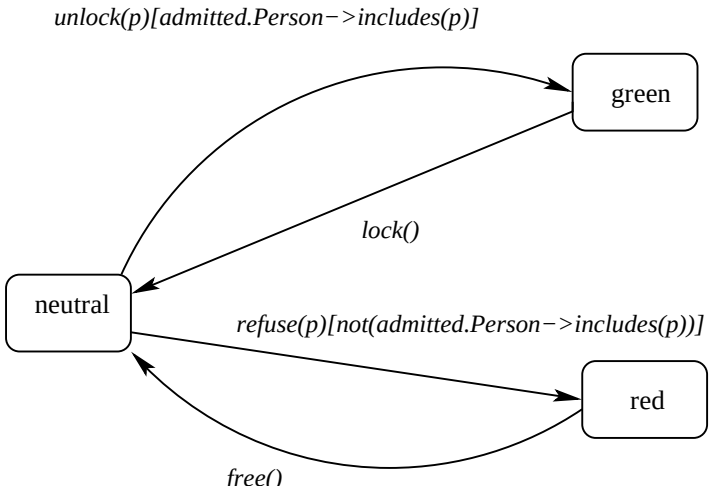


Figure 6: state-transition diagram

Example : The set *DOOR* in B machine is translated to a class *Door* in UML.

- Rule 2 : an abstract or enumerated set T in a B machine that occurs as the range in relations with other sets without occuring as a domain in any relation can be translated to the type *T* of an attribute.

  Example : if there was no invariant specifying relation between buildings, the set *BUILDING* could be translated into a type *Building* of an attribute.

- Rule 3 : the inclusion relation between two sets S and E can represent different concepts in UML :

  – generalization concept between two classes S (subclass) and E (superclass). The invariants related to the set E and specifiying attributes and associations for the corresponding class (*E*) will be inherited by the subclass (*S*) represented by the set S.
  Example : an invariant $MAN \subseteq PERSON$ linking two sets *PERSON* and *MAN* can show a generalisation relation between two classes *Man* and *Person*.

  – a possible state S of an instance of a class E.
  Example : the invariant $red \subseteq DOOR$ in this case study describe a state *red* of *Door* object.

  – the set S of effectives instances of a class E.
  Example : if we have an invariant $product \subseteq PRODUCTS$ where *product* is a variable and *PRODUCTS* is an abstract set, *product* represents all effective instances of *Product* class whereas *Products* represents all the possible instances of this class.

Relation Type	A multiplicity	B Multiplicity
relation : $A \leftrightarrow B$	*	*
partial function : $A \nrightarrow B$	*	0..1
total function : $A \rightarrow B$	*	1
partial injection : $A \rightarrowtail B$	0..1	0..1
total injection : $A \rightarrowtail B$	0.. 1	1
partial surjection : $A \twoheadrightarrow B$	1..*	0..1
total surjection : $A \twoheadrightarrow B$	1..*	1
partial bijection : $A \rightarrowtail\!\!\!\!\!\twoheadrightarrow B$	1	0..1
total bijection : $A \rightarrowtail\!\!\!\!\!\twoheadrightarrow B$	1	1

Table 1: Translating functions to associations

Relation	Partial function (injection and surjection)	Total function (injection and surjection)
set-valued	optional + mono-valued	mandatory + mono-valued

Table 2: Translating B machine functions to attributes

- Rule 4 : a relation r : $A \leftrightarrow B$ between two sets A and B representing classes *A* and *B* can be translated to an association *r* between these classes (Table. 1), or to attribute *r* of type *B* in class *A* (Table. 2). In the first case, we distinguish two multiplicities : one is associated to *A*, i.e. the number of *A* instances associated to each *B* instance and the other is associated to *B*. In the second one, attributes could be mandatory or optional, they can also be set-valued or mono-valued. If r is a constant, It indicates that the value of the corresponding link cannot be modified (*r* is an association) or that values of the corresponding attibutes could not change after the object is initialized (*r* is an attribute). Note that in this case, injective function will introduce an additional constraint asserting that for each value of the attribute *r*, will be associated only one instance of the class *A*. Surjective function will also introduce an additional constraint asserting that each value of the attribute *r* will be affected to at least one class *A*.

- Rule 5 : the operations can describe state-transition diagram of an object and/or operations in classes and link creation, modification or remove between objects. The pre-conditions specify class or association constraints, they can describe initial state of objects and guards of transition pre-conditions.The post-condition can specify operations that modify attributes and links within other classes. They can also specify final states for transitions.

4 Conclusion

Several authors have worked on combining B and UML. For example, Laleau et al. [10, 5] focus on formally deriving relational database implementations from B specifications. In [6], they outline the need for a tool to assist in development of information systems, and propose global meta-structures used to relate UML concepts relevant for information systems, but without giving translation rules or a fully-conformant IS UML metamodel. The aim of work by Treharne [13] is also to use UML and B methods in a development process but focusing on mapping UML classes into B rather than translating B notation into UML.

We have reported on some methodological rules, derived from carrying out a number of case studies, on how to represent a B specification as a UML model. Our intent has been to obtain models that are natural from the point of view of the designer and that can be useful for communication with the client, for visualisation, and for animation. Therefore, the system engineer plays an essential rôle by assisting the translation, which is not automatic. In the future, we intend to implement our approach. Our envisaged tool would document the decisions taken during the translation and check them for coherence. This experience will undoubtedly lead to a clarification of our method and to the discovery of additional rules. In the longer term, we want to be able to integrate the two languages more fully, enabling the user to go back and forth between descriptions in B and in UML.

Because refinement is central to the B method, it will have to be taken into account more rigorously by comparing the UML models that correspond to different refinements of the B specification. We hope that ultimately this will clarify ideas about refinement relations defined on UML models.

References

[1] J.-R. Abrial. Event driven sequential program construction. In *AFADL01: Approches Formelles dans l'Assitance au Dveloppement des Logiciels*, Nancy, June 2001.

[2] J.R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996.

[3] J.R. Abrial. Etude système: méthode et exemple. internal report, November 1998.

[4] I. Jacobson J. Rumbaugh and G. Booch. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[5] R. Laleau and A. Mammar. A generic process to refine a b specification into a relational database implementation. In *ZB2000:Formal Specification and Development in Z and B*, volume 1878 of *Lecture Notes in Computer Science*, pages 22–41. Springer Verlag, 2000.

[6] R. Laleau and F. Polack. Coming and going from UML to B: A proposal to support traceability in rigorous IS development. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 517–534. Springer-Verlag, 2002.

[7] B-Core(UK) Ldt. *B-Toolkit Users Manual*. Oxford (UK), 1996. Release 3.2.

[8] H. Ledang. *Traduction Systematique*. PhD thesis, LORIA-Universite Nancy 2, Nancy (F), 2002.

[9] E. Meyer. *Développement formel par objets : utilisation conjointe de B et dUML*. PhD thesis, LORIA-Universite Nancy 2, Nancy (F), mars 2001.

[10] H.P. Nguyen P. Facon, R. Laleau and A. Mammar. Combing uml with the b formal method for the specification of database applications. Technical report, CEDRIC laboratory, CNAM, Paris, September 1999.

[11] B. Tatibouet and J.C. Voisinet. jBtools and B2UML : a plateform and a tool to provide a UML class diagram since a B specification. In *ICSSEA : 14th International Conference on Software and Systems Engineering and Their Applications*, volume 2, Paris, France, december 2001.

[12] STERIA technologies de l'information. *Atelier B, Manuel Utilisateur*. Aix-en-Provence (F), 1998. version 3. 5.

[13] H. Treharne. Supplementing a UML development process with B. In *FME 2002: Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe*, volume 2391 of *Lecture Notes in Computer Science*, Copenhagen, Denmark, July 2002. Springer Verlag.

[14] J. Warmer and A. Kleppe. *The Object Constraint Language : Precise Modeling with UML*. 0201379406. Addison Wesley, 1998.