

Transformation of B Specifications into UML Class Diagrams and State Machines

Houda Fekih, Leila Jemni Ben Ayed, Stephan Merz

▶ **To cite this version:**

Houda Fekih, Leila Jemni Ben Ayed, Stephan Merz. Transformation of B Specifications into UML Class Diagrams and State Machines. 21st Annual ACM Symposium on Applied Computing - SAC 2006, Apr 2006, Dijon, France, pp.1840-1844. inria-00001269

HAL Id: inria-00001269
https://inria.hal.science/inria-00001269

Submitted on 27 Apr 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Transformation of B Specifications into UML Class Diagrams and State Machines

Houda Fekih
Dépt. Informatique
FST, Tunis, Tunisia
Houda.Fekih@loria.fr

Leila Jemni Ben Ayed
Dépt. Informatique
FST, Tunis, Tunisia
Leila.Jemni@fsegt.rnu.tn

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

ABSTRACT

We propose a rule-based approach for transforming B abstract machines into UML diagrams. We believe that important insight into the structure underlying a B model can be gained by representing it in UML, for example in order to explain the model to stakeholders that are not experts in the B formalism. We focus on the generation of class diagram and state machines. Our approach does not prescribe a mechanic algorithm for translation, giving the modeler choices to adapt the resulting UML models as appropriate.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements / Specifications; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software / Program Verification

Keywords

B method, UML, class diagram, state machine

1. INTRODUCTION

Formal and semi-formal methods and notations for software development are advocated by different schools and exhibit complementary characteristics. Semi-formal notations such as the Unified Modeling Language UML [9] emphasize graphical, intuitive representations of the object structure of systems and their components. On the other hand, formal notations and methods, including the B method [1, 2] that we consider here, are based on a small number of basic concepts that have a precise semantics and therefore enable the formal verification of correctness properties. Central to the B method is a formally defined concept of refinement by which two models written at different levels of abstraction can be compared. In particular, the refinement relation ensures that all properties of interest that hold of the abstract model are preserved by the refined one. Formal and semi-formal methods typically differ in the kind of tools supporting the methods: 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. Tool support for formal methods such as B is centered around automatic and interactive verification tools. All these observations indicate that it would be desirable to use formal and semi-formal methods side-by-side in system development.

Previous work, including [6, 7, 8, 11], has mostly focused on formalizing UML models in formal methods such as B [1], with the aim of verifying properties or eliminating inconsistencies and ambiguities of UML designs. However, it quickly becomes apparent that UML and B are based on quite different concepts. Therefore, schematic translations that attempt to cover a broad class of UML models usually result in B models that are hard to read and quite unnatural. The traceability of B model elements back to the original UML model becomes a serious problem, and failures to verify properties can be difficult to understand. For this reason, Okalas et al. [10] propose a process that aims to construct and maintain two views (UML and B) of a system.

In this paper, we consider the inverse problem and propose to construct UML class diagram and state machines from B specifications. Although perhaps less intuitive at first sight, we believe that such transformations can clarify the roles and the relationships of B model elements, which can be hard to discover in the "flat" set-theoretic language of the B method. For example, UML representations of B models can be explained to and discussed with clients and users of the system under development. We aim at intuitive and natural representations and therefore do not wish to constrain the modeler to a specific style of translation. Our transformations are therefore interactive and guided by heuristics, which can be overridden by the modeler, subject to integrity constraints that ensure the coherence of the translation.

The structure of our paper is as follows: Sect. 2 gives a general overview of our approach. Sections 3, 4, and 5 show the derivation of UML models corresponding to a B model of an access control system. Finally, Sect. 6 concludes with a discussion of our approach, future and related work.

2. OVERVIEW

Given a formal model in the form of a B machine, we aim at producing class and state diagrams. The transformation process is guided by heuristic rules and proceeds in three steps, as illustrated in Fig. 1. The initial step consists in representing the static structure of the B model as a UML

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'06 April 23-27, 2006, Dijon, France
Copyright 2006 ACM 1-59593-108-2/06/0004 ...$5.00.

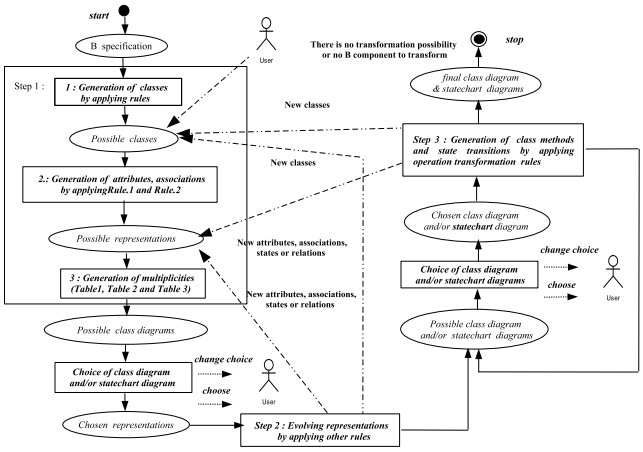


Figure 1: Overview of the translation process.

class diagram. Subsequently, the class diagram is enriched by identifying subclasses and object states. Finally, operations of the B model are translated into methods of classes, and possibly induce transitions of state machines. Each step may lead to the discovery of new classes, attributes, associations, or states, and thus enrich the diagram. Our rules are heuristics and are designed to help identify concepts such as classes, attributes, and associations. Nevertheless, the modeler may decide to override these rules in order to obtain a more natural UML representation.

As the running example to illustrate our approach we use a model of a system for controling the access of persons to a building, a fragment of which appears in Fig. 2. At first, we concentrate on the static structure of the model that is represented by the underlying sets, constants, variables, and definitions. We also take into account the assumptions on the constant parameters of the models (clause PROPERTIES) and the invariant governing the model's variables. The operations describing the dynamics of the model appear in Fig. 6 in Sect. 5.

3. INITIAL CLASS DIAGRAM

In the object-oriented approach to system modeling, a class represents a set of objects that share meaningful properties such as attributes, associations, and states they can be in. The first step of our translation is to identify those entities of the B model that represent classes. Prime candidates are those entities (appearing as SETS, CONSTANTS, VARIABLES, and DEFINITIONS) that are identified as having set type by the B type system and that appear as domains of relations or functions: such occurrences indicate that the entity has attributes or associations with other entities. Similarly, an entity occurring as a superset in an inclusion relation is likely to indicate a class (see also Sect. 4). Other sets, in particular those that appear solely as codomains of relations and functions, may be identified as being classes by the user. In particular, high-level models often do not contain enough information about relationships between sets to reliably detect classes. We formulate the following heuristic rule that helps us to identify classes.

RULE 1 (IDENTIFICATION OF CLASSES). *Let* T *be an entity of set type in the B model.*

MACHINE *AccessControl*
SETS
  PERSON; DOOR; STATE $= \{on, off\}$
CONSTANTS
  $aut, orig, dest, n$
VARIABLES
  $sit, red, unl, state$
DEFINITIONS
  $admitted(p,q) \;\widehat{=}\; orig(q) = sit(p) \wedge p \notin \mathrm{dom}(unl) \wedge (p, dest(q)) \in aut$
  $green \;\widehat{=}\; \mathrm{ran}(unl)$
  BUILDING $\widehat{=}\; 1..n$
  $empty \;\widehat{=}\;$ BUILDING $- ran(sit)$
PROPERTIES
  $aut \in$ PERSON $\leftrightarrow$ BUILDING $\wedge$
  $orig \in$ DOOR $\rightarrow$ BUILDING $\wedge$
  $dest \in$ DOOR $\rightarrow$ BUILDING $\wedge\; n \in NAT$
INVARIANT
  $sit \in$ PERSON $\rightarrow$ BUILDING $\wedge$
  $unl \in$ PERSON $\rightarrowtail$ DOOR $\wedge$
  $red \subseteq$ DOOR $\wedge\; \mathrm{ran}(unl) \cap red = \{\} \wedge (unl; orig) \subseteq sit \wedge$
  $(unl; dest) \subseteq aut \wedge sit \subseteq aut \wedge state \in$ STATE
END

Figure 2: Running example.

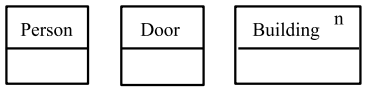


Figure 3: Identification of classes.

- *If* T *appears as the domain of a relation, it is likely to be represented by a UML class* T.
- *If* T *appears as the super-set in an inclusion relation (*$s \subseteq$ T*), it is a candidate for being represented as a UML class* T.
- *Otherwise,* T *can be represented as a UML class* T *upon the user's request.*

Classes corresponding to constant entities (i.e. SETS, CONSTANTS, *and* DEFINITIONS *that do not contain any* VARIABLES*) will be designated as «static».*

Applying Rule 1 to the example of Fig. 2, we infer (static) classes *Person* and *Door* corresponding to the sets PERSON and DOOR. Although the entity BUILDING does not immediately verify the first two conditions of Rule 1 we can infer the subset relationship *empty* $\subseteq$ BUILDING from the definition of *empty*, justifying its status as an UML class. Even without theorem proving support, the user can intervene to promote BUILDING to being a class (with a fixed number n of instances). We obtain the classes shown in Fig. 3.

The next step is to identify the attributes and associations for these classes. We use the following rule; the idea is to derive attributes and associations from relations that exist in the B model. Observe in particular that functions (including partial functions, total functions, injections etc.) are just special relations in set theory, and therefore Rule 2 applies to functions as well. However, the type of relation considered induces further constraints on the multiplicities

Relation type	A multiplicity	B multiplicity
relation : $A \leftrightarrow B$	0..*	0..*
partial function : $A \nrightarrow B$	0..*	0..1
total function : $A \rightarrow B$	0..*	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 : $A \rightarrowtail\!\!\!\!\!\rightarrow B$	1	0..1
total bijection : $A \rightarrowtail\!\!\!\!\!\rightarrow B$	1	1

Table 1: Multiplicities of associations.

Relation type	Attribute type
$A \leftrightarrow B$	set-valued
$A \nrightarrow B$, $A \rightarrowtail B$, $A \nrightarrow B$	optional, mono-valued
$A \rightarrow B$, $A \rightarrowtail B$, $A \twoheadrightarrow B$	mandatory, mono-valued

Table 2: Constraints for attributes.

of the association or the type of the attribute; these are indicated in the Tables 1, 2, and 3.

RULE 2 (ATTRIBUTES AND ASSOCIATIONS). *Assume that $r \in A \leftrightarrow B$ is a relation-valued entity of a B specification. If both A and B are represented as classes, r can be transformed into an association or an attribute. If only A is represented as a class, r will become an attribute of that class. If r is a constant entity, the association or attribute will be marked as «frozen» to indicate that its value does not change at runtime. The multiplicities and type constraints are shown in Tables 1, 2, and 3.*

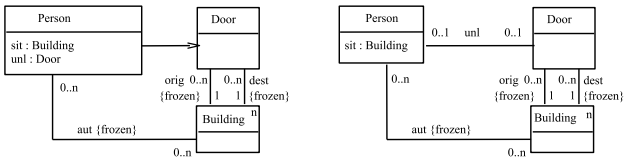
Concerning the constraints for associations and attributes, tables 1 and 2 apply when B is not a powerset, whereas table 3 governs functions of the form $A \rightarrow \mathbb{P}(B)$, which are isomorphic to relations $A \leftrightarrow B$. We also add a dependence relation from class A to class B if the model contains some relation $A \leftrightarrow B$, and all such relations have been translated to attributes.

The application of Rule 2 to our running example presents the user with a choice of possible representations. In Fig. 4, we assume that *aut* is represented as an association between the classes *Person* and *Building*, and that *sit* is represented by an attribute of the class *Person*. Similarly, *orig* and *dest* have become associations (they could also be represented as attributes). The partial injection *unl* can be transformed into an attribute of class *Person* as in Fig. 4(a)) or into an association between these classes, as shown in Fig. 4(b).

Other possible representations of relations in the B model will be considered in Rule 5 below.

Relation type	Association multiplicities	Attribute type
$A \nrightarrow \mathbb{P}(B)$	(0..*, 0..*)	optional + set-valued
$A \rightarrow \mathbb{P}(B)$	(0..*, 0..*)	mandatory + set-valued
$A \rightarrowtail \mathbb{P}(B)$	(0..1, 0..*)	optional + set-valued
$A \rightarrowtail \mathbb{P}(B)$	(0..1, 0..*)	mandatory + set-valued
$A \twoheadrightarrow \mathbb{P}(B)$	(1..*, 0..*)	optional + set-valued
$A \twoheadrightarrow \mathbb{P}(B)$	(1..*, 0..*)	mandatory + set-valued
$A \rightarrowtail\!\!\!\!\!\rightarrow \mathbb{P}(B)$	(1, 0..*)	optional + set-valued
$A \rightarrowtail\!\!\!\!\!\rightarrow \mathbb{P}(B)$	(1, 0..*)	mandatory + set-valued

Table 3: Constraints for set-valued functions.



(a) Attribute *unl*. (b) Association *unl*.

Figure 4: Associations and attributes.

4. STEP OF ENRICHMENT

Having identified the basic classes of the UML model, we will now consider some additional structure that can be inferred from the static part of the B model (i.e., constant and variable declarations, and the INVARIANT and PROPERTIES sections), and in particular identify subclass relationships and object states. We start by interpreting subset relations $A \subseteq B$ where B is represented as a class. Because sets in B can be used to represent conceptually different entities in UML, there is some ambiguity in interpreting subset relationships. The following rules 3, 4, and 5 list three different possible interpretations as subclasses, Boolean attributes or as object states. Note that rule 3 may introduce new classes.

RULE 3 (SUBSET AS SUBCLASS). *An inclusion relation $A \subseteq B$ appearing in the* PROPERTY *or* INVARIANT *sections can be interpreted as representing a subclass relationship between classes* A *and* B *provided that B has been transformed into a class.*

RULE 4 (SUBSET AS ATTRIBUTE). *An inclusion $A \subseteq B$ appearing in the* PROPERTY *or* INVARIANT *sections can be interpreted as representing a Boolean attribute* IS_A *of class* B *provided that B has been transformed into a class.*

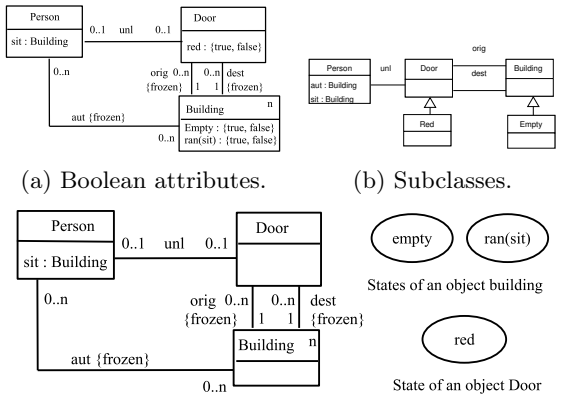
RULE 5 (IDENTIFICATION OF STATES).

- *An inclusion $A \subseteq B$ appearing in the* PROPERTY *or* INVARIANT *sections where B has been transformed into a class can be interpreted as representing a possible state* A *of objects of the class* B.
- *If $r \in A \rightarrow B$ is a variable entity where A is a class and B is an enumerated set then B can be interpreted as the set of possible states of class A.*

Starting from the class diagram of Fig. 4(b), we find that the variable entity *empty* and the set *ran*(*sit*) could alternatively be interpreted as subclasses, as (Boolean) attributes, or as possible states of class BUILDING. Similarly, the rules apply to the set *red* in relation to class DOOR, and we obtain the representations shown in Fig. 5, from which the user can choose. Subsequent transformations may indicate whether the choices were appropriate. For example, the transformation of operations will suggest that *red* is a state and not a subclass.

The analysis of rules 3–5 can be refined when a single set B has several disjoint subsets $A_1, \ldots, A_n$. All subsets A_i should then be interpreted in the same way, and we can infer some additional constraints. These ideas are formalized in Rule 6.

RULE 6 (DISJOINT SUBSETS). *In case the B model contains n disjoint subsets $A_1, \ldots, A_n \subseteq B$ of a set B that is interpreted as a class* B*, the interpretation of the A_i according to rules 3–5 can be refined as follows:*



(a) Boolean attributes. (b) Subclasses.

(c) Identifying states.

Figure 5: Interpreting subset relations.

- *If rule 3 was applied to some A_i, all sets A_j will be interpreted as subclasses, which are marked with the constraint «disjoint».*
- *In the case of rule 4, class* B *may be interpreted as having an attribute* A *of type* $\{A_1, \ldots, A_n\}$.
- *In the case of rule 5, all A_i will be interpreted as states of class* B.

If, moreover, the union of the subsets A_i can be inferred to equal B, we obtain the following information: in the first case, the generalization relation is marked with the stereotype «complete». In the second case, the attribute A becomes mandatory, and in the third case we have identified all possible object states.

The definition $empty \mathrel{\widehat{=}} \text{BUILDING} - ran(sit)$ in our B model implies that class BUILDING can be partitioned into the subsets *empty* and $ran(sit)$. Similarly, *red* and $ran(unl)$ are disjoint subsets of class DOOR, although there may be DOORs that do not belong to either subset. We therefore complete Fig. 5(c) by introducing the states IN_RAN_UNL and NOT_RED_NOT_IN_RAN_UNL of the class DOOR. These two states correspond, respectively, to the sets $ran(unl)$ and $\text{DOOR} \setminus (red \cup ran(unl))$.

The above rules need not define translations for all constants and variables of the B model. In our running example, we find that the set STATE has not been represented in our UML model so far. In these cases, we propose to add a (singleton) class that represents the B machine itself, and to represent the remaining variables or constants as attributes of this class.

RULE 7 (CONTROLLER CLASS). *A machine M that contains variables or constants that are not translated into attributes, associations or states of classes deduced from the components of the machine will be translated into a (singleton) class* M. *We infer an aggregation relation between class* M *and all classes* A *derived from sets A declared within machine M.*

This step of our transformation ends when there is no rules to apply or when all the B features except for operations have been transformed. The representation obtained will be the basis for transforming the operations of the B specification.

```
pass ≙
  ANY q WHERE q ∈ ran(unl) THEN
    sit(unl⁻¹(q)) := dest(q)
    unl := unl ⩤ {q}
    IF dest(q) ∈ empty THEN
       empty := empty - {dest(q)}
    ELSEIF sit⁻¹(orig(q)) − unl⁻¹(q) = ∅ THEN
       empty := empty ∪ {orig(q)}
    ENDIF
  END
unlock ≙
  ANY p, q WHERE p ∈ PERSON ∧ q ∈ DOOR
       ∧ q ∉ ran(unl) ∪ red ∧ admitted(p, q) THEN
    unl := unl ∪ {p ↦ q}
  END
lock ≙
  ANY q WHERE q ∈ ran(unl) THEN
    unl := unl ⩤ {q}
  END
refuse ≙
  ANY p, q WHERE p ∈ PERSON ∧ q ∈ DOOR
       ∧ q ∉ ran(unl) ∪ red ∧ ¬admitted(p, q) THEN
    red := red ∪ {q}
  END
free ≙
  ANY q WHERE q ∈ red THEN
    red := red − {q}
  END
```

Figure 6: Operations of example system.

5. TRANSFORMING OPERATIONS

The final step of our transformation process concerns the OPERATIONS section of a B specification. Operations can give rise to methods associated with classes, or to transitions of state machines. The OPERATIONS of the access-control example appear in Fig. 6.

We first infer the types of the parameters of the operations, and the entities they modify. For example, the operations unlock and lock take two parameters p (of type PERSON) and q (of type is DOOR). The operations unlock and lock modify the relation *unl*, represented in UML as an association. The operations refuse and free modify the object state of class DOOR, they will therefore clearly give rise to methods of that class. The operations unlock and lock can either be transformed into methods of class PERSON or DOOR (with an object of the other class as parameter), or they can be considered as methods of the (class) association. In general, the most difficult choice is usually that of the class to which the method should belong. All entities accessed by the operation must be linked via paths of navigability. For example, operation pass can be represented as a method of clases PERSON and DOOR but not of class BUILDING. The following rules express these observations. Observe that rule 10 may require modifications of the class diagram, possibly by creating new classes.

RULE 8. *An operation that modifies only variables translated into attributes or states of a single class will be transformed into a method of this class.*

RULE 9. *An operation that updates an association relating two classes can be transformed into a method of one of the two classes that takes the other object as argument, or into an operation of the association, and this association will be transformed into a class-association.*

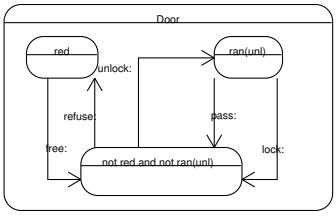


Figure 7: State diagram for class DOOR.

RULE 10. *An operation that modifies attributes of several classes or associations between different classes will be placed in a class from which all necessary attributes and associations can be reached.*

In particular, operations modify entities that represent object states, and these modifications are represented as state transitions in UML state diagrams. For example, we obtain the state diagram shown in Fig. 7 for objects of class DOOR.

6. DISCUSSION

We believe that it may be helpful to represent formal models, such as B machines, as UML diagrams, which are more familiar to non-specialists in formal methods, and which expose interesting information about the structure of a model that is not immediately obvious in the "flat" set-theoretic language on which B is based. For our purposes, we require the resulting UML model to be as "natural" as possible, and therefore consider an interactive approach where a modeler is guided by heuristics but can choose the most appropriate representation that is consistent with the underlying B model. We are currently developing a prototype tool to support the suggested method. Although we have not discussed refinement in this paper, we have extended some of our rules to handle dependencies between models at different levels of abstraction; roughly, choices made for the abstract model are recorded and should be respected during refinement. We hope to adapt and refine our approach in the light of practical experience; in particular, the design of an interface that lets the modeler investigate the choices and their consequences without being overwhelmed by irrelevant detail appears as a non-trivial challenge.

We are aware of some competing approaches, but who are mainly interested in automatic transformations from B models to UML representations. For example, Idani and Ledru [4] propose a two-step translation from B to class diagrams. An application of this work rules to the example model considered in this paper introduces a subclass RED of class DOOR as well as a subclass EMPTY of class BUILDING. Although this translation is allowed by our rules, we believe that an alternative representation based on object states is more natural. They also considers the generation of state machines from B models [5], but this step appears to be independent of the generation of the class diagram from the static structure, which appears questionable to us.

Tatibouet et al. [12, 13] have also considered the problem of deriving separately UML class diagram and state machines from B specifications. The approach described in [13] is restricted to specifications having scalar variables typed with enumerated sets and can therefore not accomodate typical object-oriented structures.

7. REFERENCES

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

[2] J.-R. Abrial. Event driven sequential program construction. In *Approches Formelles dans l'Assistance au Développement des Logiciels (AFADL 2001)*, Nancy, 2001.

[3] J.-R. Abrial. Etude de cas: Le contrôle d'accès aux bâtiments. June 2000.

[4] A. Idani and Y. Ledru. Object oriented concepts identification from formal B specifications. In *9th Intl. Workshop Formal Methods for Industrial Critical Systems (FMICS 2004)*, ENTCS 133, pp. 159–174, 2005.

[5] A. Idani and Y. Ledru. Dynamic graphical UML views from formal B specifications. In *Information and Software Technology*, to appear.

[6] K. Lano, D. Clark, and K. Androutsopoulos. UML to B: Formal verification of object-oriented models. In *Proc. 4th Intl. Conf. Integrated Formal Methods (IFM2004)*, LNCS 2999, pp. 187–206. Springer, 2004.

[7] H. Ledang. *Traduction Systématique de spécifications UML en B*. PhD thesis, Université Nancy 2, Nancy, France, 2002.

[8] E. Meyer. *Développement formel par objets : utilisation conjointe de B et d'UML*. PhD thesis, Université Nancy 2, Nancy, France, 2001.

[9] Object Management Group. Unified Modeling Language Specification, Version 2.0.

[10] D. Okalas Ossami, J.-P. Jacquot, and J. Souquières. Consistency in UML and B multi-view specifications. In *Proc. 5th Intl. Conf. Integrated Formal Methods (IFM2005)*, LNCS, Springer, 2005 (to appear).

[11] C. Snook and M. Butler. UML-B : Formal modelling and design aided by UML. Technical Report, Electronics and Computer Science, University of Southampton, September 2004.

[12] B. Tatibouet, A. Hammad, and J.C. Voisinet. From an abstract B specification to UML class diagrams. In *2nd IEEE Intl. Symp. Signal Processing and Information Technology (ISSPIT'2002)*, Marrakech, Morocco, 2002.

[13] B. Tatibouet and J.C. Voisinet. Generating statecharts from B specifications. In *16th Intl. Conf. Software and Systems Eng. and Their Applications (ICSSEA)*, Paris, France, 2003.