Formalizing UML Behavioral Diagrams with B

Hung Ledang, Jeanine Souquières

Formalizing UML Behavioral Diagrams with B

Hung LEDANG and Jeanine SOUQUIÈRES

LORIA - Université Nancy 2 - UMR 7503
Campus scientifique, BP 239
54506 Vandœuvre-lès-Nancy Cedex - France
Email: {ledang,souquier}@loria.fr

Abstract. An appropriate approach for translating UML to B formal specifications allows one to use UML and B jointly in an unified, practical and rigorous software development. We formally analyze UML specifications via their corresponding B formal specifications. This point is significant because B support tools like AtelierB are available. We can also use UML specifications as a tool for building B specifications, so the development of B specifications become easier.

This paper reports our recent results on formalizing UML behavioral diagrams in B notations. We are planning to present automatic derivation schemes from UML behavioral diagrams to B specifications. Our proposal together with the formalization in B of UML structure specifications provide a complete frameworks to translate UML specifications into B. We discuss also the perspectives for analyzing UML behavioral specifications via the derived B specifications.

Keywords: UML, class operation, event, use case, B method, B abstract machine (BAM), B operation.

1 Introduction

The Unifi ed Modeling Language (UML)[25] has become a de-facto standard notation for describing analysis and design models of object-oriented software systems. The graphical description of models is easily accessible. Developers and their customers intuitively grasp the general structure of a model and thus have a good basis for discussing system requirements and their possible implementation. However, the fact that UML lacks a precise semantics is a serious drawback of UML-based techniques.

On the other hand, B[1] is a formal software development method that covers software process from the abstract specifi cation to the executable implementation. A strong point of B (over other formal methods) is support tools like AtelierB [28], B-Toolkit [2]. Most theoretical aspects of the method, such as the formulation of proof obligations, are carried out automatically. The automatic and interactive provers are also designed to help specifi ers to discharge the generated proof obligations. All of these points make B adapted for large scale industrial projects [3]. However, as a formal method, B is still diffi cult to learn and use.

As cited many times in the literature [14, 22, 24, 27, 4], an appropriate combination of object-oriented techniques and formal methods can give rise a practical and rigorous software development. For this objective, we advocate the integration of UML

and B specification techniques. Our approach is to propose derivation schemes from UML into B specifications. This UML-B integration has the following advantages: (i) the construction of UML specifications is rigorously controlled by analyzing derived B specifications; (ii) the construction of B specifications becomes easier thanks to UML specifications. From the informal description of requirements, we successively build the object models with different degrees of abstraction. These models cover from conceptual models through logical design models to the implementation models of the software. This also means that the developed models are successively refined. We verify the consistency of each object model by analyzing the derived B specification. We verify the conformance between object models by analyzing the refinement dependency among them that is formally expressed in B.

In this paper, we present our recent results on formalizing in B the UML behavioral diagrams, which has been so far an open issue. We emphasize on the automatic translation from UML behavioral diagrams into B specifications. Our work combined with the previous works on formalization in B of class diagrams provide a complete frameworks to derive B specifications from UML specifications. Hence, a rigorous analysis on UML specifications via the derived B formal specifications could be realized.

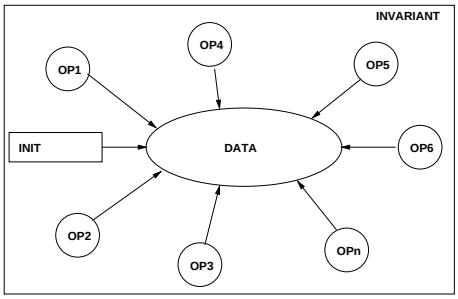
Section 2 gives a brief introduction to the B method. In Section 3 related work on UML-B integration is presented; we also justify why the previous work cannot apply to formalize in B the UML behavioral diagrams. Sections 4-6 detail the principles for formalizing in B the UML behavioral concepts. The possibilities of automatic translation from UML behavioral diagrams to B specifications are also discussed. In Section 7 we discuss the perspectives to analyze UML specifications using the derived B specifications. Finally, some concluding remarks in Section 8 complete our presentation.

2 The B method

B [1] is a formal software development method that covers a software process from specification to implementation. The B notation is based on set theory, the language of *generalized substitutions* and first order logic. Specifications are composed of *B components*. A B component, which is a *B abstract machine* (BAM), a *refinement component* or an *implementation component* of a BAM, is similar to modules or classes. A BAM (Figure 1 right) consists of a set of variables, invariance properties relating to those variables and operations. The variable values are only modifiable by operations which must preserve the invariant (Figure 1 left).

Generalized substitutions are used to express the post-conditions of B operations. We can use generalized substitutions to specify the non-determinism (at the abstract specification level) and also the determinism (at the implementation specification level). This point is a notable difference[1] with respect to Z and VDM, which use only logic expressions. Generalized substitutions provide a more familiar frame to specifiers by integrating the essential methodological aspects like invariant and refinement. Refinement can be seen as an implementation technique but also as a specification technique to progressively augment a specification with more details. At every stage of the specification,

[1] This is seen as a strong point of B over Z and VDM.



MACHINE ...
...
/* *DATA* is declared in SETS, CONSTANTS
and VARIABLES clauses */
SETS ...
CONSTANTS ...
VARIABLES ...
INVARIANT *INVARIANT predicates*
INITIALISATION *INIT substitutions*
OPERATIONS
OP1 = ...
...
OPn = ...
END

Fig. 1. Intuitive view of a BAM

proof obligations ensure that operations preserve the system invariant. A set of proof obligations that is sufficient for the correctness must be discharged when a refinement is postulated between two B components. Hence, by supporting proved refinement, B allows to go progressively from an abstract specification (non deterministic) to a deterministic specification that can be translated into a programming language (ADA, C and C++).

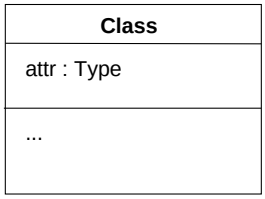
Another characteristic of the B method is that it was designed to be automated easily. The generation of proof obligations (of the invariant preservation and the refinement correctness) obeys the simple rules that can be easily implemented in a piece of software. Furthermore, support tools like AtelierB and B-Toolkit provide utilities to discharge automatically and interactively the generated proof obligations. Analyzing the non-discharged proof obligations with the B support tools is an efficient and practical way to detect errors encountered during the specification development.

Finally, beside the refinement, B provides structuring mechanisms like "IMPORTS", "INCLUDES", "USES", "SEES" so that B components can be composed by various ways. Thus, large systems can be specified in a modular way, possibly reusing parts of other specifications.

3 Formalization of UML specifications in B: state of the art

Meyer [22] and Nguyen [24], based on the previous work of Lano [12], have proposed a set of precise and implementable rules for modeling in B almost the concepts of the class diagrams. Given a class `Class` (Figure 2 left) a BAM *Class* (Figure 2 right) is created by the followed manner: a B deferred set *CLASS*, which models the set of possible instances (instance space) of the class `Class`, is declared. The set of the effective instances of the class `Class` is modeled by a B variable *class* constrained to be a subset of *CLASS*. For each attribute `attr`, a B variable *attr* is created and defined in the INVARIANT clause as a binary relation between the B set *class* and a B set *Type* modeling the type `Type` of `attr`. This binary relation may be refined in a more sophisticated relation, such as a function (as in the current example), a bijection, etc, according to the additional features of `attr`. It is to be noticed that *Type* is declared in a special BAM

called *Types* linked with *Class* by the SEES clause. The reason is that we can reuse *Type* in the other BAMs which also model data of type `Type`. In addition, in the INVARIANT clause, apart from typing predicates there are often predicates modeling the additional constrains of the class and its attributes (the class invariant [21]).



MACHINE *Class*
SEES *Types*
SETS
CLASS
VARIABLES
class, attr
INVARIANT
$class \subseteq CLASS \land$
$attr \in class \rightarrow Type$
...
END

Fig. 2. Formalization in B of class diagrams

An association `ass` between two classes `Class1` and `Class2` is identified by couples of instances. It is naturally expressed in B as a variable *ass* of the type of the binary relation (maybe a more sophisticated relation as noticed earlier) between B variables *class1* and *class2*. If `ass` is a non-fixed association[2] then `ass` gives rise to a BAM, otherwise the B variable *ass* is attached to one of the BAMs *Class1* or *Class2*. Currently, only the inheritance relationship between domain class has been treated. The B variable of a subclass in a specialization hierarchy is a subset of the B variable of its superclass. The BAM of a subclass "USES" the BAM of its superclass. For reason of space, we omit examples of formalization in B of association and inheritance.

The important idea in the work of Meyer and Nguyen in the formalization of class diagrams is that a BAM is created for each class: the attributes are modeled as the data in the BAM; the class operations become B operations in the BAM. At first glance, this seems evident, however, at a closer inspection, the concept of class and the concept of BAM do not coincide with each other. A class operation can affect the data from different classes but a B operation affects only data declared in the same BAM. For this reason, only simple class operations like constructor, destructor or operations that set or query the value of each attributes (selector and mutator), which are local to classes, can be actually modeled. We could not model non basic class operations involving several classes. Consequently, with the current UML-B derivation schemes, we cannot translate interaction diagrams like sequence and collaboration to B. In addition, the attempts in [22, 24, 12, 26] to translate state-chart diagrams into B have two shortcomings: (i) these approaches could not deal with transition having multiple actions; (ii) (cf. the self-evaluation of Meyer [22]), it was difficult and even impossible to automate these rules in a piece of software due to the ambiguity of the translation rule. For this reason, the problem of formalization of UML behavioral diagrams has been so far an open issue.

[2] The association between two classes whose instances are independently created/deleted in comparison with the instances of related classes.

4 Formalizing class operations in B

4.1 General idea

To overcome the shortcoming of Meyer and Nguyen on specifying in B pre-/post conditions of non-basic class operations, we propose to group the class operation and its concerned data in the same BAM. Given an UML specifi cation shown in Figure 3. The class operation `op11` is modeled in a BAM (Figure 4) whose data are derived from both classes `Class1` and `Class2`.

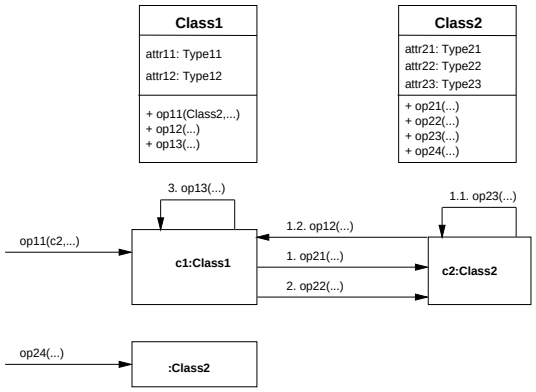


Fig. 3. An UML specification

MACHINE *MachineA*
SEES *Types*
SETS
CLASS1; CLASS2
VARIABLES
class1,attr11,attr12,class2,attr21,attr22,attr23
INVARIANT
$class1 \subseteq CLASS1 \wedge class2 \subseteq CLASS2 \wedge$
$attr11 \in class1 \rightarrow Type11 \wedge attr12 \in class1 \rightarrow Type12 \wedge ...$
...
OPERATIONS
...
op11(*cc1,cc2,...*) =
pre
$cc1 \in class1 \wedge cc2 \in class2 \wedge ...$
then
/* modeling the effect of the non-basic operation `op11`, this time, is similar to modeling the effect of the basic operation `op23` */
end
END

Fig. 4. Formalization in B of the class operation `op11`

In order to conserve the modularity of B specifications we consider the calling-called dependency[3] amongst class operations. This relationship is used to arrange class operations in different BAMs. For each calling-called class operation pair, the B operation of the called operation participates in the implementation of the B operation which model the calling operation[4]. That means that: (i) the BAM for the called operation is imported in the implementation of the BAM for the calling operation and (ii) we use the B implementation operation to model the realization of non basic class operations. Figure 5 shows an example of formalization of the calling-called dependency between `op11` and `op21`, `op22` and `op13`.

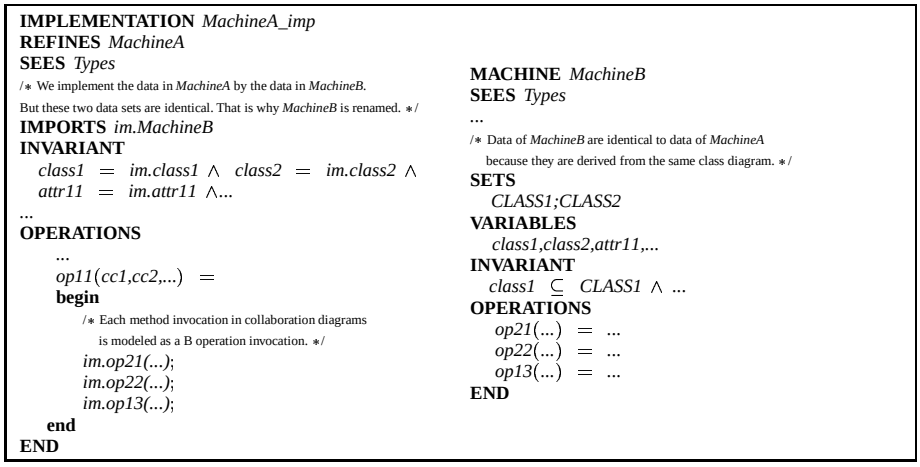


Fig. 5. Formalization in B of the calling-called dependency amongst class operations

From the description above, it is to be noticed that the data of each class could be duplicated in several BAMs where are modeled some operations of the class. The basic class operations are modeled in the BAMs for classes and non-fixed associations. Each non basic class operation `op` is modeled in two stages: (i) modeling the effect of `op` by a B abstract operation *op*; (ii) implementing the B operation *op* in the first step by calling B operations of class operations appearing in the realization of `op`.

4.2 Integrating class and interaction diagrams in the same B specification

Our proposal has been applied to derive automatically B specifications from class and interaction diagrams which realize certain class operations[5]: we use interaction dia-

[3] A calling-called pair relates a class operation - the calling operation - to one of its realization class operations - the called operation.

[4] Recently, we discovered that the BAM of the "realized" operation can be also refined by including the BAM of the "realizing" operations. In general, two possibilities are equal and here afterward we only speak of the implementation/importation dual.

[5] We can also model the activity-based part in the state-chart and activity diagrams.

grams to establish the calling-called dependency amongst class operations. If there is no cyclic calling-called dependency[6] amongst class operations, we are able to arrange class operations into layers (using two procedures: “division” and “dummy-promoting” [16, 15]) such that:

(i) there is no calling-called dependency amongst operations in the same layer;
(ii) the basic operations, which do not have any called operation, are in the bottom layer;
(iii) the system operations, which do not have any calling operation, are in the top layer;
(iv) the operations in a layer differing from the bottom layer only have called operations in the next lower layer. For this purpose, certain operations are duplicated in several layers thanks to the “dummy-promoting” procedure.

Figure 6 represents the layer arrangement of class operations in Figure 3. Each arrowed line represents a calling-called pair or a duplicating-duplicated pair of class operations and the arrow is at the called or duplicated end. After the arrangement, each layer gives rise to a BAM in which model the class operations in the associated layer[7]. A BAM that does not belong to the bottom layer, is implemented by importing the BAM for the next lower layer. Finally, we can decompose the BAM for the bottom layer into BAMs for classes and their non-fixed associations (if any). From layer arrangement in Figure 6, we create three BAMs corresponding to three layers. For reasons of space, we do not show here the derived B code.

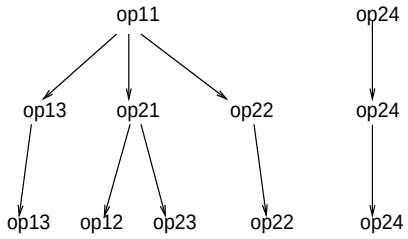


Fig. 6. Layers of class operations

From the description above, the architecture, data and operation skeleton of B specifications are automatically generated. It remains to fill up the body of B operations. In Section 6 we will discuss the way to generate automatically this content. Moreover, we have not yet considered the formalization in B of asynchronous messages. This problem will be briefly mentioned in Section 5 and had been detailed in [17].

[6] This is still an open issue due to technical restrictions of the B language.
[7] Remember that, data in each BAM are derived from the whole class diagram.

4.3 Extension to use cases

Our approach for formalizing class operations can also be extended to deal with use cases as described in [13]. Each use case is also modeled as a B operation. The data in BAMs for use cases are derived from classes related to use cases[8]. We treat "includes" stereotype between use cases as the calling-called dependency between class operations. Due to technical restrictions of the B language, we propose to model a use case and its possible "extends" use cases as distinct B operations in the same BAM.

5 Formalizing events in B

The proposal of Meyer [23, 22] and Sekerinski [26] for modeling in B the events only works in cases without multiple actions related to a single transition. To overcome this shortcoming, we have proposed a two-stages approach [17] for modeling events: (i) modeling the effect of each event as a B abstract operation; (ii) implementing the B operation in the first step by calling B operations for the triggered transition and associated actions.

Given a set of classes and their state-chart diagrams, an integration procedure [17] has been proposed to derive the corresponding B specification:

(i) creating a *System* BAM to model all events in state-chart diagrams. Data in *System* are derived from classes and states; this once again (cf. Section 4) allows us to express easily the pre- and post specification of events;
(ii) creating a *Basic* BAM to model all transitions, actions, state-checking and guard conditions; the data in *Basic* are also derived from classes and states;
(iii) decomposing *Basic* into BAMs for classes and associations;
(iv) implementing *System* by importing *Basic*.

To deal with asynchronous messages amongst state-chart diagrams, a B data structure modeling the signal type and an additional B operation modeling the sending of signal are supplemented for each type of signal. Details are described in [17].

6 Generating automatically the content of B operations

According to Sections 4-5, at present we can only automatically derive the architecture of B specifications from UML specifications. The data, the skeleton of B operations in the B specification are also automatically derived. In order to complete B specifications, we must fill up the body of B operations. For the purpose of a complete automation of transformation, we propose to attach to each element like use cases, events, actions, guard condition and class operations, an OCL-based pre/-post specification. Hence, the abstract content of B operations can be derived by using OCL-B rules of Marcano [19]. The implementation content of B operations for use cases, events and non-basic class operations can be derived from corresponding diagrams (use case diagrams for use cases; collaboration or activity diagrams for class operations; state-chart diagrams for events). The precise rules will be proposed at a later stage.

[8] According to Kilov et al.[11] and Glinz [10], use case and class are complementary views of a requirements specification.

7 Analyzing UML specifications via B specifications

Our experience shows that many defects inside UML specifications can be detected even during the formalization step. For instance a class operation has appeared in the interaction diagrams but was not declared in the class diagrams or there is a mismatch between an operation call in interaction diagrams regarding its declaration in class diagrams. A list of those defects and an automatic support tool for certain defects can be found in the Egyed's dissertation [7]. However, this section presents the perspectives of the using the derived B specifications to detect the more sophisticated semantic defects inside UML specifications.

By using B support tools to analyze the proof obligations generated for initialization substitutions in the INITIALIZATION clause invariant predicates in the INVARIANT clause, the defects involving object diagrams such as the object initialization can be detected. In addition, the cardinality and class invariant of class diagrams could also be analyzed at the same time. Then we can analyze the conformance of behavioral diagrams regarding the class diagram as described below.

Consider the class operation `op11` in Figure 3, the consistency of `op11` regarding the classes `Class1` and `Class2` is expressed by the fact that the B operation *op11* preserve the invariant on the data derived from both classes. This point can be analyzed by considering B proof obligations related to the abstract content of *op11*. If any proof obligations cannot be discharged (automatically and manually using B support tools utilities) there would be something wrong in the description of the specification of `op11` (Assuming that the class invariant of `Class1` and `Class2` is consistent).

The calling-called dependency amongst class operations is analyzed by considering B proof obligations of the B implementation components. Consider the `op11`-`op21` pair in Figure 3 in which `op21` appears in the realization of `op11`. We can analyze the context (regarding `op11`), in which `op21` is called, by considering the B proof obligations generated from the implementation of *op11*. If there are some non-discharged proof obligations, there would be mismatch between `op11` and `op21`.

Similarly, the consistency of use cases with respect to their related classes is expressed by the fact that the B operations of use cases preserve the invariant on the data derived from classes. The structuring of use cases into sub use cases can be analyzed by considering the implementation of the B operation for super use cases.

Finally, concerning the state-charts. We can use B proof obligations of the B specification derived from state-charts to verify the "invariants de liaison" [20] : (i) if the action sequence related to a transition can be executed when we are in its source state; and (ii) if the effect of the transition bring us to its destination state.

Related work about analyzing UML specifications via their derived formal specifications. There has been a lot of work on integrating UML and Z specifications such as the work by France et al. [8, 9] and by Dupuy [6]. Dupuy centered mainly on the formalization in Z and Object-Z of the class diagram and state-chart diagrams of the information systems, therefore, only the consistency inside class diagrams and between state-chart and class diagrams have been attached. France et al. supplemented the formalization in Z of the interaction diagrams, however, only the consistency of class operations regarding the class diagrams has been analyzed. They have not yet attached the

conformance between class operations as ours. In addition, the limit of Z tool supports in generating and discharging the proof obligations imply the hand generation and analysis of the Z proof obligations for the consistency of UML specifications mentioned in the work of France et al and Dupuy. This is error-prone and is a big drawback of using Z with respect of B.

8 Concluding remarks and further work

Bruel [4] has pointed out three considerations of an approach for integrating between informal and formal specifications techniques: (i) the preservation of the intuitive-held interpretation of the informal specifications; (ii) the level of automated support for moving between formal and informal specifications and (iii) the degree of integration.

For the first consideration, our experiences shown that B notations are almost adapted to express the semantics of UML specifications. For example, the object encapsulation is naturally expressed in B. Other aspects of the object orientation like the polymorphism, the object interactions can be simulated. The concurrence can be implicitly expressed thanks to concepts of B operation and B system.

Regarding the second and the third considerations, our approach provide a complete framework for deriving B specifications from UML structure and behavioral diagrams. Hence, the conformance between two aspects (the structure and the behavior) of UML specifications can be formally verified by analyzing the corresponding B specification. Analyzing derived B specifications (thanks to B powerful support tools) is a practical and rigorous way to improve initial UML specifications.

We have validated our formalization proposals by non trivial case studies. The formalization of use cases in B (Section 4.3) has been experimented with a case study on a controlling system for accessibility of buildings [18]. The approach for formalizing in B class operations has been tried with several case studies: the pump component of a controlling system for petrol dispensing [5], the patterns like Client-Server, Broker. The formalization in B of events has been applied for an elevator controller system.

Case studies for analyzing UML specifications will be envisaged. The support tools for translating class diagrams to B [22] will be extended to take into account our translating rules for UML behavioral diagrams. In addition, the formalization in B of the UML refinement dependency is also our study objectives [13, 14].

References

[1] J.R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-521-49619-5.

[2] B-Core(UK) Ltd, Oxford (UK). *B-Toolkit User's Manual*, 1996. Release 3.2.

[3] P. Behm, P. Desforges, and J.-M. Meynadier. MÉTÉOR: An Industrial Success in Formal Development, April 1998. An invited talk at the 2nd Int. B conference, LNCS 1939.

[4] J.M. Bruel. Integrating Formal and Informal Specification Techniques. Why? How? In *the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques*, pages 50–57, Boca Raton, Floria (USA), 1998. Available at http://www.univ-pau.fr/ bruel/publications.html.

[5] D. Coleman, P. Arnold, St. Bodoff, Ch. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development : The Fusion Method*. Prentice Hall, 1994.

[6] S. Dupuy. *Couplage de notations semi-formelles et formelles pour la spécification des systèmes d'information*. PhD thesis, Université Joseph Fourier - Grenoble 1, Grenoble (F), semtembre 2000.

[7] A.F. Egyed. *Heterogenous View Integration and Its Automation*. PhD thesis, University of Southern California, USA, August 2000.

[8] R.B. France and J.M. Bruel. Rigorous analysis and design with the unified modeling language. http://www.univ-pau.fr/ bruel/Tutorials/etapsTut.html, 2001. ETAPS 2001 Tutorial Proposal.

[9] R.B. France, J.M. Bruel, M. Larrondo-Petrie, and E. Grant. Rigorous Object-Oriented Modeling: Integrating Formal and Informal Notations. In *6th International AMAST Conference*, LNCS 1349, Sydney (A), December 1997. Springer-Verlag.

[10] M. Glinz. A Lightweight Approach to Consistency of Scenarios and Class Models. In *the 4th International Conference on Requirements Engineering*, Illinois (USA), June 10-23, 2000.

[11] H. Kilov, H. Mogill, and I. Simmonds. Invariants in the trenches. In H. Kilov and W. Harvey, editors, *Object-Oriented Behavioral Specifications*, chapter 6, pages 77–100. Kluwer Academic Publishers, 1996.

[12] K. Lano. *The B Language and Method : A Guide to Practical Formal Development*. FACIT. Springer-Verlag, 1996. ISBN 3-540-76033-4.

[13] H. Ledang. Des cas d'utilisation à une spécification B. In *Journées AFADL'2001 : Approches Formelles dans l'Assistance au Développement de Logiciels*, Nancy (F), 11-13 juin, 2001.

[14] H. Ledang. Formal Techniques in the Object-Oriented Development: an Approach based on the B method. In *the 11th PhDOOS Workshop: PhD Students in Object-Oriented Systems*, Budapest (Hu), http://www.st.informatik.tu-darmstadt.de/phdws/wstimetable.html, June 18-19, 2001.

[15] H. Ledang and J. Souquières. Integrating UML and B Specification Techniques. In *the GI2001 Workshop: Integrating Diagrammatic and Formal Specification Techniques*, Universität Wien, Österreich, September 26, 2001. http://www.pst.informatik.uni-muenchen.de/GI2001/index.html.

[16] H. Ledang and J. Souquières. Modeling class operations in B : a case study on the pump component. Technical Report A01-R-011, Laboratoire Lorrain de Recherche en Informatique et ses Applications, March 2001. Available at http://www.loria.fr/∼ledang/publications/UML01.ps.Z.

[17] H. Ledang and J. Souquières. New Approach for Modeling State-Chart Diagrams in B. Available at http://www.loria.fr/∼ledang/publications/state-chart-modeling.ps.gz, July 2001.

[18] Y. Ledru, G. Padiou, and J. Jaray. Étude de cas: Système de contrôle d'accès. http://www-lsr.imag.fr/afadl2000/EtudeDeCas/, 2000.

[19] R. Marcano and N. Lévy. Transformation d'annotations OCL en expressions B. In *Journées AFADL'2001 : Approches Formelles dans l'Assistance au Développement de Logiciels*, Nancy (F), 11-13 juin, 2001.

[20] O. Maury, C. Oriat, and Y. Ledru. Invariants de liaison pour la cohérence de vues statiques et dynamiques en UML. In *Journées AFADL'2001: Approches Formelles dans L'Assistance au Développement de Logiciels*, 11-13 juin, 2001.

[21] B. Meyer. *Reusable Software*. Prentice Hall, 1994.

[22] E. Meyer. *Développements formels par objets: utilisation conjointe de B et d'UML*. PhD thesis, LORIA - Université Nancy 2, Nancy (F), mars 2001.

[23] E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. In *FM'99 : World Congress on Formal Methods in the Development of Computing Systems*, LNCS 1708, Toulouse (F), September 1999. Springer-Verlag.

[24] H.P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles*. PhD thesis, Conservatoire National des Arts et Métiers - CEDRIC, Paris (F), décembre 1998.

[25] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998. ISBN 0-201-30998-X.

[26] E. Sekerinski. Graphical Design of Reactive Systems. In D. Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method - 2nd International B Conference*, LNCS 1393, Montpellier (F), April 1998. Springer-Verlag.

[27] C. Snook and R. Harrison. Practitioners Views on the Use of Formal Methods: An Industrial Survey by Structured Interview. *Information and Software Technology March 2001*, 43:275–283, 2001.

[28] STERIA - Technologies de l'Information, Aix-en-Provence (F). *Atelier B, Manuel Utilisateur*, 1998. Version 3.5.