New Approach for Modeling State-Chart Diagrams in 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 integrating UML and B specification techniques allows us to map UML specifications into B specifications. Therefore, we can formally analyze an UML specification via the corresponding B formal specification. This point is significant because B support tools are available. We can also use UML specifications as a tool for building B specifications. Thus, an approach for a practical and rigorous software development, which is based on object and B from the requirements elicitation to the executable code, is proposed.

In this paper, we address the problem of modeling UML state-chart diagrams in B, which has not been, so far, completely treated. We distinguish between event-based and activity-based parts of state-chart diagrams. We propose creating, for each part, a B specification. Because activities relate to class operations, we can use our previous work on modeling class operation for modeling the activity-based part. Hence, we consider here only the event-based part. A new approach for modeling events is proposed. The asynchronous communication amongst state-chart diagrams is also considered.

Keywords: UML, state-chart diagram, event, activity, class operation, B method, B abstract machine (BAM), B operation.

1 Introduction

The Unifi ed Modeling Language (UML)[14] 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 object-oriented techniques based on UML.

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 [18], B-Toolkit [2]. Most theoretical aspects of the method, such as the formulation of proof obligations, are done automatically by tools. Provers are also designed to run automatically and reference a large library of mathematical rules, provided with the system. All of these points make B well adapted in large scale industrial projects [3]. However, as a formal method, B is still diffi cult to learn and to use.

As cited many times in the literature [7, 11, 13, 16, 17], an appropriate combination of object-oriented techniques and formal methods can give a way that is applicable in the software industry. For this objective, we advocate integrating object-oriented and B techniques. Our approach is to propose derivation schemes from UML concepts into B notations. This UML-B integration has following advantages: (i) the construction of UML specifications is formally controlled; (ii) the construction of B specifications becomes easier with the help of 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 among object models by analyzing the refinement dependency among them that is formally expressed in B.

At the present, we only consider the modeling of UML concepts in B. The problem of analyzing the derived B specification remains at a later stage. The works in [5, 11, 12, 13, 15] proposed a set of rules for mapping UML static diagrams into B. However, their approaches for mapping UML state-chart diagrams into B have several shortcomings: (i) they did not consider the concept of activity in state-chart; (ii) they could not work in cases where several actions are sequentially associated to the same individual transition and in addition (iii) only synchronous communication but not asynchronous communication amongst state-charts has been considered.

In this paper, we present a new approach for mapping state-chart diagrams into B. We distinguish between activity-based and event-based parts of state-chart diagrams[1]. We propose deriving for each part a B specification. The activity-based part relates only to class operations, so the activity-based B specification can be developed by using the approach in our previous works [6, 9, 7, 8] for modeling class operations and use cases. Like Meyer and Sekerinski's proposals, we propose modeling each event as a B operations. However, our proposal differs from theirs by a modeling in two stages:

(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.

In our opinion, this two-stages approach allows us to overcome the second shortcoming of existing works. Dealing with asynchronous communication amongst state-chart diagrams is another contribution in this paper. For each type of signal, a B operation modeling the sending is created. Thus, there would be some extra data structure (with respect from classes, attributes and states) associated with this operation.

The remainder of this section presents an example which is used throughout the presentation. In Section 2, remarks about works for modeling in B state-chart diagrams are presented. In Section 3 we detail the way to model events. In Section 4 we go on to present the modeling of the asynchronous communication amongst state-chart diagrams. The integration of state-chart diagrams into B specifications derived from class

[1] We define event-based part the element in state-chart diagrams concerning events. These elements comprise events, state, transition and actions related to transitions. The remainders in state-chart diagrams constitutes activity-based part.

diagrams is discussed in Section 5. Finally, in Section 6, concluding remarks complete our presentation.

1.1 Example

Given an UML specification as shown in Figure 1. The class diagram presents two classes `Class1` and `Class2` linked by an association class `Ass`. Attributes and operations are defined for each class and association. There are two state-chart diagrams associated to `Class1` and `Class2`. Objects of `Class1` have three possible states : `St1`, `St2`, `St3`. The change of the state is triggered by `Evt1`, `Evt2` and `Evt3`. When `Evt1` occurs at `St3`, a signal `Evt4` is sent to the state-chart diagram of `Class2`. There is also a synchronous message `sendc2.op21()` when `Evt2` occurs at `St2`. The state-chart diagram associated to `Class2` is composed of two states `St4`, `St5` and two events `Evt4`, `Evt5`.

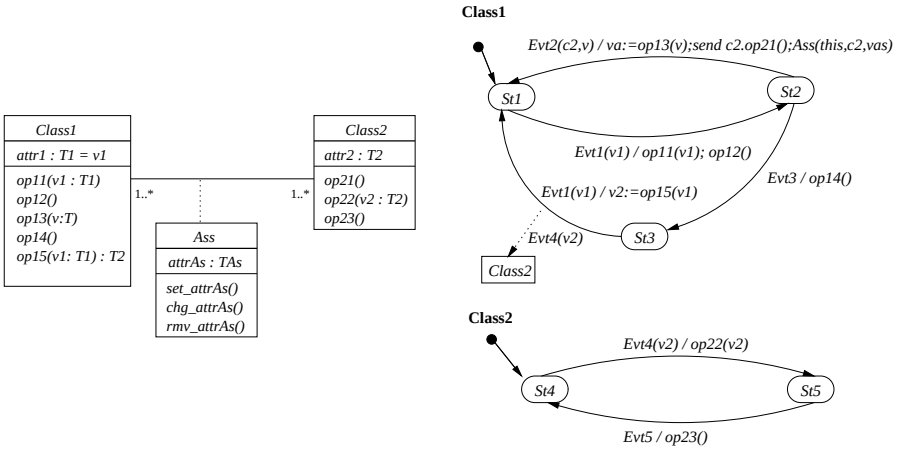

Fig. 1. An UML specification

2 Modeling in B state-chart diagrams: state-of-the-art

So far, the work of Meyer [12, 11] has been considered the latest one for modeling state-chart diagrams in B. This work grouped all the "best" ideas proposed previously by Lano [5] and Sekerinski [15] in modeling state-chart diagrams. This section recalls main points in Meyer work.

States. For each state-chart diagram associated to the class `Class`, we create a B enumerated set *STATE* which gathers all the states of the diagram. The state of an object is recorded by a B variable *state* defined as a function from the B variable *class*, which models the set of effective instances of `Class`, to *STATE*. Thus, the state of an object `oo`

is defined as *state(oo)*. Changing the state corresponds to the modification of *state*. The initial state is set up in the instance creation operations as it is done for class attributes. Figure 2 presents the formalization of the states assigned to the class `Class1` (cf. Figure 1).

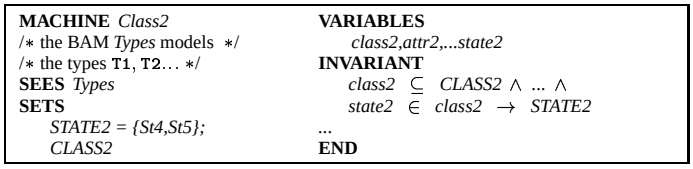

Fig. 2. Template for the state

Let us recall that the special BAM *Types* is used to declare all attribute types of classes. This BAM is seen (link "SEES") by BAMs derived from classes in which we model the attributes (cf. [11]).

Transitions. Each transition is formalized by a B operation which models the change of the state. Figure 3 shows the B operation *transitionSt4_St5(...)* modeling the transition from `St4` state to `St5`.

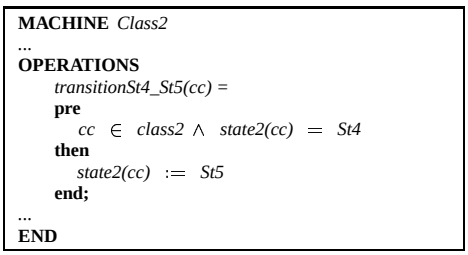

Fig. 3. Transition modeling

Actions. Each action in state-chart diagrams corresponds to a class operation. Thus, each action is naturally modeled as a B operation (cf. Figure 4).

Events. Each event is also formalized by a B operation. This operation is parameterized by the target objects and the eventual parameters of the event. Parameters are typed by a predicate in the precondition clause. The post condition clause is made up by invocations to B operations modeling the triggered transitions and their associated actions. Figure 5 shows the B operation *Evt4(...)* for the event `Evt4` (cf. Figure 1).

At the first glance, the approach for modeling events in B seems to be evident. However, two problems can be raised:

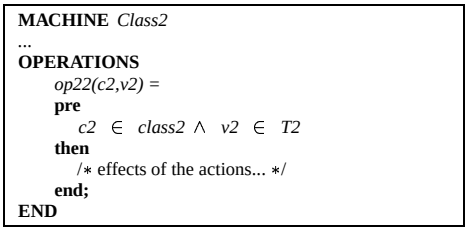

Fig. 4. Action modeling

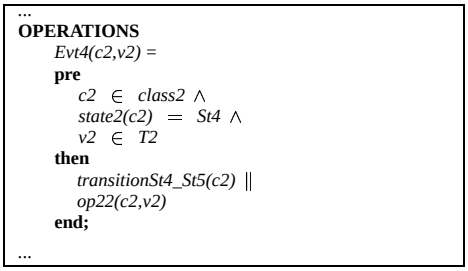

Fig. 5. Event modeling

(i) how to distribute B operations into BAMs? Considering the example in Figure 5. Because *Evt4(...)*, *transitionSt4_St5(...)* and *op22(...)* are in BAMs, the BAM for *Evt4(...)* must "INCLUDES" the BAM for *transitionSt4_St5(...)* and *op22(...)* (cf. Figures 3 and 4). However, in this case the fact of calling two B operations from the same included BAM is not allowed according to [1, 18]. Meyer proposed to declare *transitionSt4_St5(...)* in the clause DEFINITION. Nevertheless, this proposal is too restrictive. It only works if there is on more than one action for each action being operations in the same class. In case where a transition has more than two actions from the same class (cf. the transition from `St1` to `St2`) the proposal becomes unrealistic;

(ii) what about the sequential actions? Suppose that we have an appropriate solution for distributing B operations for transitions and for actions into BAMs. The situation in the transition from `St2` to `St1` does not allow us to model sequential call of actions in the B operation *Evt2(...)*. This point is again due to technical restrictions of the B language [1, 18].

The two above problems were justified by the fact that there has not been, so far, an appropriate solution to automatically map state-chart diagrams into B. In [4], Laleau and Mammar have presented a support tool for generating B specifications from UML diagrams of data intensive applications. Although they considered UML state-chart diagrams, nothing new is added with respect to Nguyen's work [13].

3 New approach for modeling events in B

3.1 The two-stages approach

As usual, we model each event as a B operation. However the B operation for each event must be implemented. In other words, there are two B specifications for each event: the first is a B abstract operation in a BAM; this abstract operation is implemented by the second. For each event, in the B abstract operation we specify directly the effect of the event on the related data. It is only in the B implementation operation we make explicitly invocations to B operations of the transitions and actions related to the event. Building the abstract content and implementation content for B operations of events constitutes two stages in modeling events into B[2].

3.2 Grouping event and data in the same BAM

In order to specify directly the effect of an event on its concerned data, we propose grouping a class operation and its related data in the same BAM. Thus, the problem of modeling events becomes the one of how B substitutions can be used to express the pre-/post specification related to the event. This is similar to model actions. Figure 6 shows a BAM *System*, in which the B operation *Evt4(...)* corresponds to the event `Evt4`. In the data declaration section (clauses SETS, VARIABLES...) of *System* we notice the presence of the data derived from the class `Class2`. In addition, *System* must "SEES" *Types* because *System* models the attributes of `Class2`.

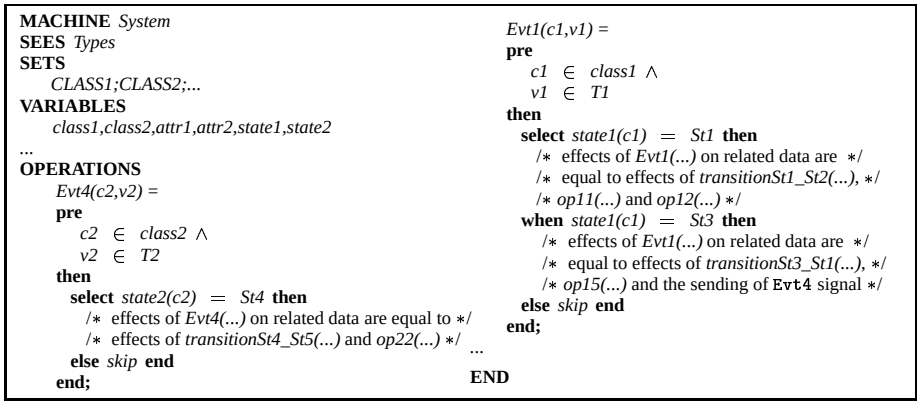

Fig. 6. Grouping event and related data in the same BAM

We propose creating a BAM (*System*) for all the events. Hence, data of the created BAM are derived from the whole class diagram (at least all classes associated with state-chart diagrams). However, *System* does not contain B operations for transitions and

[2] Similar ideas have been used in our previous work for modeling use cases and class operations [6, 9, 7, 8].

actions which must be modeled in the BAMs for classes and associations. In addition, the BAMs for transitions and actions participate to implement *System* as described in the following section.

3.3 Implementing B operations of events

By importing BAMs of transitions and actions in the implementation of *System* we can model the effects of events by calling B operations of transitions and actions. By using the B implementation construct and the B importation mechanism we are able to overcome two questions mentioned in Section 2. In a B implementation component, the sequence is allowed; in addition, we can make several calls to operations in the same imported BAM. This is also the justifi cation to choose the implementation and importation dual instead of refi nement and inclusion dual. In Figure 7, the B operation *Evt4(...)* of *System* is implemented by calling *transitionSt4_St5(...)* and *op22(...)* of *Class2*[3].

```
IMPLEMENTATION System_imp
SEES Types
REFINES System
/* we rename Class2 because its data are identical to one party of data in System, */
/* and this point must expressed in the clause invariant of System_imp */
IMPORTS im.Class2; ...
INVARIANT
    class2=im.class2 ∧
    attr2=im.attr2 ∧...
...
OPERATIONS
    Evt4(c2,v2) =
    var bb in
        /* to implement conditions in the substitution select */
        bb ← im.isState4(c2);
        if bb = TRUE then
          im.transitionSt4_St5(c2);
          im.op22(c2,v2)
        else skip end
    end;
...
END
```

Fig. 7. Implementing B operations of events

As we can see, both *System* and *Class2* contain some data with the same name and properties; this is because they are all derived from the same class (`Class2`). The clause INVARIANT in *System_imp* must assert this remark. In addition, we must rename *Class2* in the clause IMPORTS in order to clearly distinguish the data with the same name in *System* and *Class2* in the clause INVARIANT of *System_imp*.

Let look at the B operation *isState4(c2)*. This utility operation is necessary to implement the condition *state2(c2) = St4* in the substitution select of the operation *Evt4(...)* in the BAM *System* (cf. Figure 6). Note that, for each such state checking, there is an

[3] Indeed, as you can see in Section 5.3, *System_imp* imports a BAM *Basic* which includes *Class2*.

operation. We can also apply this technique for guard conditions that are associated to events and are modeled in the substitution select clause of B operations for events. Please notice that, the B operations modeling guard conditions and state checking can be assigned to BAMs derived from classes or associations.

4 Modeling the asynchronous communication amongst state-chart diagrams

4.1 Communication amongst state-chart diagrams

According to Rumbaugh et al. [14], an object can send messages to another objects by synchronous or asynchronous manners. A call event is a type of synchronous message. In Figure 1, `sendc2.op21()` is such a message. This means that the class operation `op21()` is called in the transition from `St2` to `St1`. Hence, an approach like the one in Section 3 is appropriate for modeling synchronous messages.

Signals are explicit means by which objects may communicate with each other asynchronously. In Figure 1, `Evt4` is a signal sent from the state-chart diagram for `Class1` to the state-chart diagram for `Class2`. In this case the state-chart diagrams of `Class1` and of `Class2` evolve independently. That means that `Evt4` is registered in a signal queue related to the state-chart diagram of `Class2`. Treating `Evt4` may be differed later and is not concerned by the state-chart diagram of `Class1`. In the following section, the signal sending amongst state-chart diagrams is discussed.

4.2 Sending a signal

At the receiver, the received signals must be stored before they are handled. We must store all informations related to the signal: the sender, the receiver and eventual parameters. For this purpose, we create for each type of signal `S` a queue `S_Queue` to store all the received signal of type `S` which are not treated. Each item in the queue `S_Queue` is a record comprising fields modeling the sender, the receiver and eventual parameters of signals. In our example, the record for `Evt4` contains three fields:

(i) the reference to an object of `Class1` acting the sender;
(ii) the reference to an object of `Class2` acting the receiver and
(iii) the third field for the argument v2 of the type `T2`.

Hence, `Evt4_Queue` can be modeled in B as a B variable *Evt4_Queue* defined as followed:

$$Evt4\_Queue \subseteq class1 \leftrightarrow class2 \leftrightarrow T2$$

Sending an `Evt4` signal is therefore modeled as adding a triplet $\{c1 : class1, c2 : class2, v2 : T2\}$ into *Evt4_Queue.* Figure 8 shows the B operation *SendEvt4(...)* modeling the registering of a signal. This operation is called when we implement the B operation *Evt1(...)* of the event `Evt1` in the state-chart diagram of `Class1`. In the abstract content of *Evt1(...)* (cf. Figure 6), it is sufficient to place the code inside *SendEvt4(...)*[4].

[4] Hence *SendEvt4(...)* is in a BAM that is imported in the implementation of *System* which contains *Evt1(...).*

MACHINE ... /∗ this name is discussed in Section 5.3 ∗/
SEES *Types*
SETS
CLASS1;CLASS2; ...
VARIABLES
class1,class2,attr2,state1,state2,Evt4_Queue
INVARIANT
... ∧ $Evt4\_Queue \subseteq class1 \leftrightarrow class2 \leftrightarrow T2$
INITIALISATION
... || $Evt4\_Queue := \phi$ || ...
OPERATIONS
SendEvt4(c1,c2,v2) =
pre
$c1 \in class1 \wedge c2 \in class2 \wedge v2 \in T2$
then
$Evt4\_Queue := Evt4\_Queue \cup (\{c1\} \leftrightarrow \{c2\} \leftrightarrow \{v2\})$
end;
...
END

Fig. 8. Sending `Evt4`

5 Integrating state-chart diagrams into B specifications

This section presents the way to develop B specifications from class and state-chart diagrams. We consider here only the event-based part of the state-chart diagrams. The reason is that the activity-based part concerns only class operations and therefore we can use the approach for modeling class operation to derive the B specification corresponding to the activity-based part.

5.1 Data in the B specification

By definition (cf. [11]) the data in the B specification models the data in class and state-chart diagrams. That means that we have B data for:

(i) classes, association, attributes in the class diagrams;
(ii) states in the state-chart diagrams.

According to Section 4.2, if there is any asynchronous communication amongst state-chart diagrams, we also have B data for signal queues.

5.2 Operations in the B specification

In general, the B operations are mainly used to model events, transitions, actions. However, according to Sections 3 and 4, there are also B operations modeling:

(i) state checking, guard condition checking (cf. Section 3.3);
(ii) signals amongst objects.

The B operations modeling state-checking and guard condition are called utility operations and are used in the implementation of the B operations for events. A B operation modeling a signal is also used to implement the B operation of the event whose occurrence provokes the signal (cf. Section 4.2).

5.3 Generating the architecture of the B specification

As stated in Section 3.2, we propose grouping all B operations modeling events in a BAM called *System* (cf. Figure 6). Data in *System* are mentioned in Section 5.1.

At first glance, the B operations for transitions, actions, sending signals, state checking and guard conditions are grouped in the another BAM *Basic* (cf. Figure 9). The data in *Basic* are also derived from the whole class and state-chart diagrams. We implement *System* by importing *Basic* so that we can implement events' B operations by making invocations to B operations of transitions, actions, sending signals and other utility operations in *Basic*. According to Section 3.3, *Basic* is renamed in the clause IMPORTS inside *System_imp*.

MACHINE *Basic*
SEES *Types*
SETS
CLASS1;CLASS2; ...
VARIABLES
class1,class2,attr2,state1,state2,Evt4_Queue
INVARIANT
... ∧ Evt4_Queue ⊆ class1 ↔ class2 ↔ T2
INITIALISATION
... || Evt4_Queue := ϕ ...
OPERATIONS
transitionSt4_St5(c2) = ...
op22(c2,v2) = ...
bb ← isState4(c2) = ...
SendEvt4(c1,c2,v2) = ...
...
END

Fig. 9. *Basic* models transitions, actions, signal sending, state and guard condition checking

However, because each transition is local for each state-chart diagram, which in turn is associated in a given class. That means that we can delegate transitions' B operations of *Basic* to BAMs for classes. The actions are also local to classes and associations, so their B operations in *Basic* can be also delegated to BAMs of classes and associations. The situation is also the same for B operations modeling the state checking and guard conditions. Hence, *Basic* is created by including BAMs for classes and associations. In *Basic*, we declare only data and operations relating to the signal sending amongst state-chart diagrams. The remainder data and operations are distributed into BAMs for classes and associations (cf. Figure 10).

5.4 Generating the content of B operations

At present we can only automatically derive the architecture of B specifications. The data, the skeleton of B operations in the B specification are also automatically derived. According to Meyer [11], the B operations for transitions can be automatically derived. According to Section 4.2 the B operations for sending asynchronous messages can also be automatically derived. For the purpose of a complete automation of the derivation,

MACHINE *Basic*
SEES *Types*
/∗ distributing B operations for transition, actions, state checking, ∗/
/∗ guard conditions into BAMs *Class1,Class2,Ass* ∗/
EXTENDS *Class1, Class2, Ass*
VARIABLES
Evt4_Queue
INVARIANT
$Evt4\_Queue \subseteq class1 \leftrightarrow class2 \leftrightarrow T2$
INITIALISATION
$Evt4\_Queue := \phi$
OPERATIONS
SendEvt4(c1,c2,v2) = ...
...
END

Fig. 10. Decomposing *Basic* in BAMs for classes and associations

we propose to attach OCL-based specifications to events, actions and guard conditions. Hence, the abstract content of B operations for events, actions and guard conditions can be derived by using OCL-B rules of Marcano [10]. The implementation content of B operations for events can be derived from state-chart diagrams. The precise tranlation rules will be proposed at a later stage.

6 Conclusion

In this paper, we present a new approach for modeling UML state-chart diagrams in B. The B operation modeling an event is implemented by B operations modeling the triggered transition and its associated actions of the event. Our approach is "better" than the one of Meyer and Sekerinski [12, 11, 15] in the sense that it allows more than one actions, which are sequential, to be associated to a transition. We also present an approach for modeling asynchronous communication, which has not been so far considered.

Our approach can be implemented in a piece of software. Together with previous works [6, 9, 7, 8] we are able to provide a complete framework for deriving B specifications from UML structure and behavior diagrams. Hence, the conformance between two aspects (the structure and the behavior) of an UML specification can be formally verified by analyzing the corresponding B specification.

For further work, a collaboration with Marcano and Lévy is envisaged to integrate their OCL-B translation rules [10] in our work (cf. Section 5.4); a study to translate UML state-chart diagrams into B implementation operations is also envisaged. In addition, the support tool for automatically translating class diagrams into B specifications developed by Meyer [11] will be extended to take into account UML behavioral diagrams.

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] R. Laleau and A. Mammar. An Overview of a Method and its support Tool for Generating B Specifications from UML Notations. In *The 15st IEEE Int. Conf. on Automated Software Engineering*, Grenoble (F), September 11-15, 2000.

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

[6] 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.

[7] 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 (H), http://www.st.informatik.tu-darmstadt.de/phdws/wstimetable.html, June 18-19, 2001.

[8] H. Ledang and J. Souquière. 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

[16] C. Snook and M. Butler. Verifying Dynamic Properties of UML Models by Translation to the B Language and Toolkit. Technical Report DSSE-TR-2000-12, Declarative Systems & Software Engineering Group, Department of Electronics and Computer Science University of Southampton, September 2000. Available at http://www.dsse.ecs.soton.ac.uk/techreports/2000-12.html.

[17] 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.

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