Test of object-based specifications using B notations

Ninh Thuan Truong, Jeanine Souquières

Test of object-based specifications using B notations

Ninh-Thuan Truong, Jeanine Souquières

LORIA, Université Nancy 2, Campus Scientifique BP 239
54506 Vandœuvre lès Nancy cedex France
Email: {truong,souquier}@loria.fr

Abstract. We propose an approach to test object-based specifications using B notations. We start from an UML specification in the form of a class diagram and sequence diagrams which express scenarios modelling the system's behavior. These diagrams are transformed into a B specification which is completed by the definition of the operations (messages in the sequence diagrams corresponding to the methods in the class diagram), and safety and dynamic properties on the system. The test of scenarios and the satisfaction of the properties is done by means of a theorem prover.

1 Introduction

Object-oriented approaches [1, 8, 22] offer a natural way for developing software systems. However, the majority of these approaches suffer from the absence of formal methods at all stages of the development. Therefore, the systems developed using these approaches generally are not reliable.
Formal methods are necessary for specifying a rigorous software. They allow mathematical manipulation and reasoning, and facilitate rigorous testing procedures. Research on formal methods in object-oriented frameworks is still in an early stage.
Object-based approaches based on B notations [23, 15, 20] usually integrate UML [25, 8] and B [3, 29]. Those are object-oriented systems in which inheritance and sub-typing are not considered. Object-based systems are composed of objects which run concurrently and communicate by means of message passing. In these approaches, a B abstract machine corresponds to a class, B relation clauses are used to model relations between UML classes. B proof obligations guarantee the correctness of the specification of each separated operation with the invariant.
In object-based approaches, a scenario is a textual description or a procedural description of the actions of objects through time to perform a particular task. It is usually defined in a UML sequence diagram describing the interactions of messages between objects. Sequence diagrams, in which the communication aspect is predominant, are a basis for the description of tests [27].
Dynamic constraints have been considered in B-Event [4, 6] and in object oriented approaches, establishing a link between time and object-orientation [11, 9]. Dynamic constraints express temporal properties on the messages exchanged between objects, like liveness properties that the system must satisfy.

In this paper, we propose an approach to test object-based specifications using B notations. We start from an UML specification in the form of a class diagram and sequence diagrams which express scenarios modelling the system's behavior. These diagrams are transformed into a B specification allowing us to test:

- the consistency of the sequential execution of operations of a scenario, expressed by sequence diagrams,
- the execution of a scenario with safety and dynamic properties of the system.

To reach this aim, we propose:

- to derive a B specification from the UML class diagram,
- to introduce a new B machine, called a *simulation machine*, in order to specify test scenarios as sequences of operations calls,
- to integrate safety and dynamic constraints on the system in this new machine.

Tests will be performed by using a B theorem prover. As we have introduced new notations in the B ones, we have to define the proof obligations which characterise these notations in order to use B existing provers.
The structure of this paper is as follows. Section 2 gives the background of the approach with a brief presentation of the B method and the derivation of UML classes into a B specification. Section 3 presents our approach with the structure of the simulation machine, the expression of safety and dynamic constraints and the definition of the proof obligations to test the execution of a scenario with system properties. Section 4 illustrates the approach on a case study. Section 5 presents related works. Finally, section 6 concludes and discusses further work.

2 Background

In this section, we give a brief overview of the B formal method and the background necessary to understand the transformation of UML classes into B.

2.1 The B method

B [3] is a formal software development method, originally developed by J.-R. Abrial. The B notations are based on set theory, the language of generalised substitutions and first order logic. A system development begins by defining an abstract view which can be refined step by step until an implementation. The different refinement steps can be verified using theorem provers [19, 30]. The method has been successfully used in the development of several complex real-life applications, like the METEOR project [26]. It is one of the few formal methods which has robust, commercially available support tools for the entire development life-cycle from the specification down-up to code generation [7]. Specifications are composed of abstract machines which are very close to notions well-known in programming under the name of modules, classes or abstract data types. Each abstract machine consists of a set of variables, invariant properties of those variables and operations. The state of the system, i.e. the set of variable

values, is modifiable by operations which must preserve its invariant. The proof obligations to the preservation of invariants are generated automatically by support tools like AtelierB [30], B-Toolkit [19] and B4free [10], an academic version of AtelierB. Checking proof obligations with B support tools, automatic or interactive proofs [2], is an efficient and practical way to detect errors encountered during the specification development.

2.2 Transformation of UML classes to B

The transformation of UML to B [13, 20, 23, 16] aims at using B as an object-based specification language and to verify UML specifications thanks to B support tools. Our proposal takes into account this work in which a UML class is derived into a B machine (figure 1), where:

- a constant *CLASS* corresponds to a set of possible objects of the class. *CLASS* is defined as a subset of the set of all possible object, *OBJECTS*, which is defined as a deferred set,
- a variable *class* models the set of objects generated by the class. *class* is defined as a subset of *CLASS*.

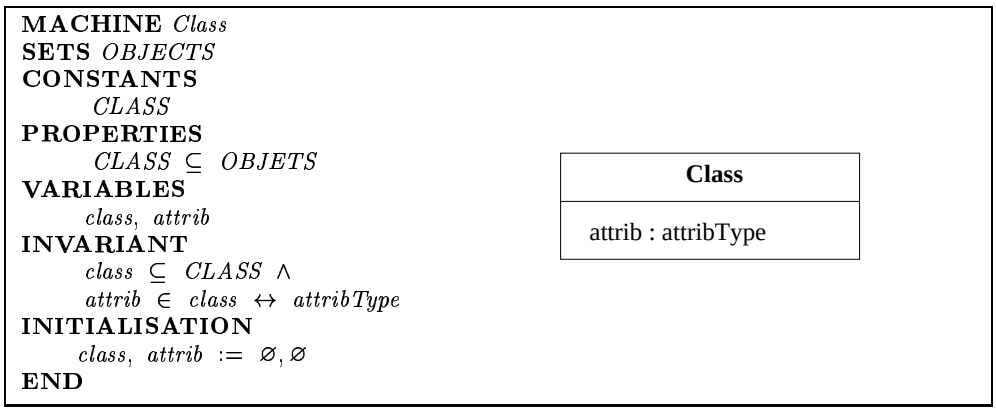

Fig. 1. Derivation of a UML class to B

The attribute attrib is derived into a variable, *attrib*, in the abstract machine *Class*. Its type is defined in the INVARIANT clause as a relation between the set of objects instantiated from the class and its type attribType. Operations of a class are derived as operations of the B abstract machine Class. The relation clauses between B abstract machines are used to connect the machines derived from classes in a class diagram.

3 Simulation machine

We have introduced a new machine in B, namely the simulation machine, in order to express execution for scenario corresponding to the transformation of

UML sequence diagrams, taking into account safety and dynamic properties of the system. We present the structure of the simulation machine and propose proof obligations to test the execution of scenarios using the B theorem prover.

3.1 Structure of the simulation machine

From a UML sequence diagram which describes the interaction between messages, we can build a sequential composition of B operations. Each message in object-based system is specified by an operation in B abstract machines. We for to express this sequential composition in a machine called simulation (figure 2). This machine is composed of three clauses:

- the clause INVARIANT expresses safety properties on the system,
- the clause MODALITIES expresses dynamic properties, which are presented in the section 3.3,
- the clause INITIALISATION allows to give initial values to the state of the system for the given scenario,
- the clause OPERATIONS contains the main operation. This one is composed of a sequence of operation calls, each operation been specified in a B abstract machine.

SIMULATION *Scenario*
INVARIANT
  $\Box J$
MODALITIES
  $\Box(P \Rightarrow \Diamond Q)$
  $\Box(P \Rightarrow Q \ \mathcal{W} \ R)$
INITIALISATION
  Init
OPERATIONS
  main =
  begin
    OP
  end
END

Fig. 2. Structure of the simulation machine

3.2 Proof obligations associated to the operations clause

As presented above, the simulated execution is specified by a sequence of n operation calls, each one is indexed by i:

$$OP = OP_1; OP_2; ...; OP_n$$

Each operation is composed of a precondition and a substitution body:

$$OP_i = P_i \mid S_i$$

so

$$OP = [P_1 \mid S_1]; [P_2 \mid S_2]; ...; [P_n \mid S_n]$$

The invariant of each abstract machine and the one of the simulation machine must hold in the simulated execution. In order to perform the simulation, we replace parameters in the called operations by their effective values. The definition of the operation OP_i is expressed by:

$$r_i \longleftarrow OP_i(para_1, para_2, ..., para_m)$$

and its call is of the form:

$$v_i \longleftarrow OP_i(value_1, value_2, ..., value_m)$$

For each operation OP_i called, according to the definition of the semantics of the substitution, we have to prove that the effective values of its parameters satisfy its precondition P_i:

$$P_{iv} = [para_1, para_2, para_m := value_1, value_2, .., value_m]P_i$$

After replacing each parameter in the body of the operation by their value:

$$S_{iv} = [r_i, para_1, para_2, ..., para_m := v_i, value_1, value_2, .., value_m]S_i$$

Let $[S_v^i]$ be the execution of substitutions in the body of the first i operations of the scenario after replacing each parameter of each operation by its value, taking into account the initialisation of the simulation machine:

$$[S_v^i] = [Init][S_{1v}][S_{2v}]...[S_{iv}]$$

The main operation of the simulation machine calls n operations defined in abstract machines. Let A be the conjunction of constraints of sets, $Prop$ the conjunction of properties and I the conjunction of invariants of all the k abstract machines of the system ($j \in [1..k]$):

$$A = \bigwedge A_j,\ Prop = \bigwedge Prop_j,\ I = \bigwedge I_j$$

The execution expressed by a sequence of operation calls in the clause OPERATIONS of the simulation machine is validated if the following proof obligations are verified:

- For $i = 1\ ..\ n{-}1$. The proof obligations guarantee that the execution of the system establishes the invariant I of all the abstract machines defining the system. For each called operation OP_i, we have to verify that the precondition of OP_{i+1} of the $(i{+}1)^{th}$ operation call is satisfied by the postcondition obtained by the execution of the first i operation calls of the scenario. The execution of the $(i{+}1)^{th}$ operation call is ensured by the result of the execution of the first i operation calls of the scenario:

$$A \wedge Prop \wedge I \wedge P_{iv} \Rightarrow [S_v^i](P_{(i+1)v} \wedge I)$$

- For $i = n$. The proof obligations guarantee that the execution of the scenario preserves the invariant I of all the abstract machines:

$$A \wedge Prop \wedge I \wedge P_{nv} \Rightarrow [S_v^n]I$$

3.3 Expression of system dynamic properties and proof obligations

We present the system properties that must be satisfied by the execution of a scenario and their proof obligations. These properties are expressed in the simulation machine by:

- safety properties in the clause INVARIANT J and
- dynamic properties in the clause MODALITIES. We introduce liveness properties [21], which have been already introduced in object-oriented notations [11, 12, 31].

A. Safety property. A *safety property* refers to a formula P and requires that P is an invariant over all the computations of the specification in which P is defined. In the temporal logic notation, such a property is expressed by $\Box P$.

The following proof obligation guarantees that the execution of the scenario establishes the invariant J of the simulation machine.

$$\forall\, i.(i \in [1..n] \;\Rightarrow A \wedge Prop \wedge I \wedge P_{iv} \Rightarrow [S_v^i]J)$$

B. Liveness properties. We consider two kinds of liveness property, the response property and the precedence property [11].

B1. A *response property* refers to two formulae P and Q and requires that every P-state, i.e. a state satisfying P, arising in an execution is eventually followed by a Q-state. In the temporal logic notation, this is expressed by $\Box(P \Rightarrow \Diamond Q)$. To verify this property, the following two proof obligations have to be established:

$$\exists\, i.(i \in [1..n{-}1] \;\wedge\; A \wedge Prop \wedge I \wedge P_{iv} \Rightarrow [S_v^i]P) \;(1)$$

This first proof obligation (1) verifies if P can be established by the execution of the scenario. If this proof obligation is satisfied, which means that there exists an operation OP_i $(i \in [1...n{-}1])$ in the scenario which leads to the state s_i where P holds, we have to verify the second proof obligation (2):

$$\exists\, j.(j \in [i{+}1..n] \;\wedge\; A \wedge Prop \wedge I \wedge P_{jv} \Rightarrow [S_v^j]Q) \;(2)$$

This proof obligation verifies if the predicate Q is satisfied on the state s_j which follows the state s_i in the execution of the scenario $(j > i)$.

B2. A *precedence property* refers to three formulae P, Q and R. It requires that every P-state is followed by a sequence in which Q is satisfied and that sequence is either terminated by a R-state or by a Q-state. In the temporal logic, this property is expressed by $\Box(P \Rightarrow Q\ \mathcal{W}\ R)$.
First, we have to verify that the predicate P is established by the execution of the scenario:

$$\exists\, i.(i \in [1..n{-}1] \;\wedge\; A \wedge Prop \wedge I \wedge P_{iv} \Rightarrow [S_v^i]P)$$

If the predicate P is satisfied on the state s_i, $(i < n)$, we verify that there exists an operation call OP_j $(j \in [i{+}1..n]\ )$ in the scenario which leads to a state s_j where R holds:

$$\exists j.(j \in [i{+}1..n] \ \wedge \ A \wedge Prop \wedge I \wedge P_{jv} \ \Rightarrow [S_v^j]R)$$

– If the predicate R is not established, we have to prove that each operation call OP_j (where $j \in [i{+}1..n]$) in the scenario establishes the predicate Q:

$$\forall j.(j \in [i{+}1..n] \ \Rightarrow A \wedge Prop \wedge I \wedge P_{jv} \ \Rightarrow [S_v^j]Q)$$

– If the predicate R is established on a state s_j, we have to prove that the predicate Q is not established for this state:

$$A \wedge Prop \wedge I \wedge P_{jv} \ \Rightarrow [S_v^j](\neg \ Q))$$

3.4 The use of the proof to test B object-based specifications

The basic idea we have developed is to check the specification of B operations specified in abstract machines relatively to the safety and dynamic properties by simulating scenarios of the system's behavior. With the use of B notations and the definition of proof obligations for the simulation machine introduced as a new notation in B, we are able to use the B theorem prover to test the execution of the system. This is done in two steps:

– first, we prove a scenario of the behavior of the system defined as a sequence diagram,
– once the execution of this given scenario has been proven, we have to prove the scenario satisfies safety and dynamic properties of the system.

4 Case study

We illustrate our approach on a simplified case study. The system will be able to control the access of certain persons to a given building.

4.1 Presentation of the case study

The control takes place on the basis of the authorisation that each concerned person is supposed to possess. Each person involved receives a magnetic card with a unique identifying code, which is engraved on the card itself. A card reader is installed at the entrance (and at the exit) of the building. A person wishing to enter the building follows a systematic procedure composed of the sequence of events which follows:

– he puts his card into the card reader and inputs his code. One is then faced with the following alternative:
  • if he is authorised, his entrance is accepted:
    * the door is open,
    * the card is ejected by the card reader,
    * the person takes his card,
    * the person enters the building and
    * the door is closed;
  • if he is not authorised, his entrance is refused:
    * the door remains closed,
    * the card is ejected by the card reader and
    * the person takes his card.

4.2 UML Specification

We first introduce a UML class diagram to structure this system. Then we model the proposed system's behavior by means of sequence diagrams. We precise the properties of the system.

1. **UML class diagram.**
   As cards are the only informations known by the controller, we have decided to mix up a person and a card in our model, introducing a class Card. As we have simplified the problem taking into account only one building, we do not introduce the notion of building in the model.
   The operation insertCard in the class Reader includes the insertion of the card into the card reader and the input of the code by the person.
   The authorisations are represented in the class Controller by a set named authorised_cards. The dynamic situation of persons in the building is represented by the set inside_cards.

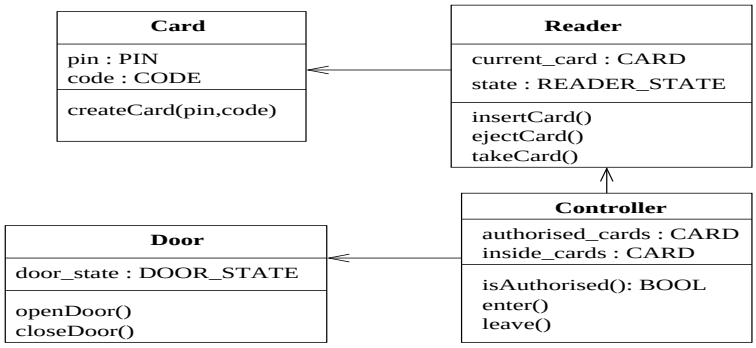

Fig. 3. Class diagram of our simplified access control system

2. **Sequence diagram.**
   We express Figure 4 the scenario of the entry to the building informally presented in section 4.1 by a UML2.0 sequence diagram [24].
   When using UML1.5, we have to introduce two scenarios to express this behavior, one corresponding to the authorised case, the second to the not authorised case. Figure 5 presents the first case.

3. **Constraints on the system.**
   An implicit property concerns the impossibility for a same person to be in the given building and to wish to enter in this building. This safety property can be expressed as follow:
   - *at any one moment, a person authorised to enter the building is either inside the building or outside.*

   The system must satisfy the next dynamic constraints:
   - *if a person input a card, this card will be ejected,*
   - *the door is maintained closed until a person is authorised to enter.*

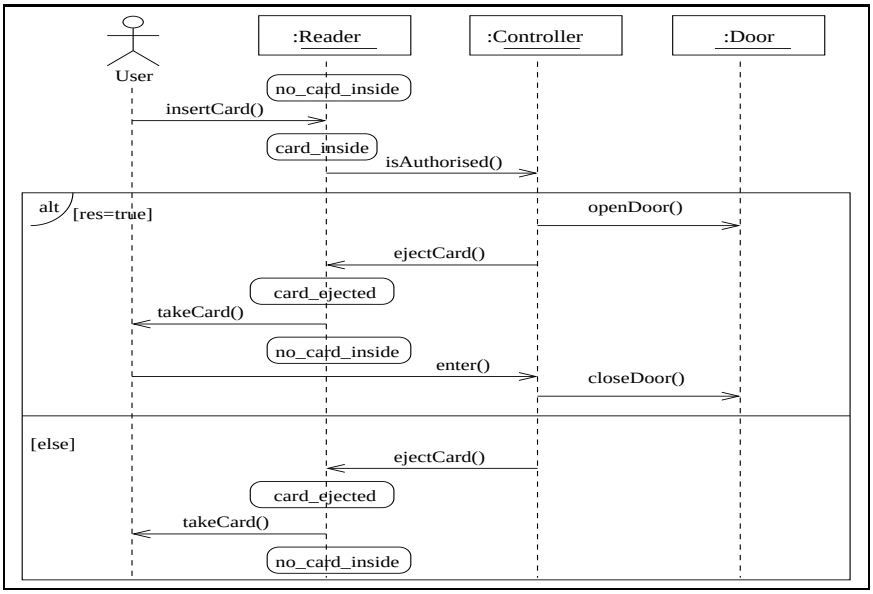

Fig. 4. A UML2.0 global scenario for the entry to a building

4.3 B specification

1. **Abstract machines.**
   The class diagram of the Figure 3 is derived into a B specification using automatic transformation rules [23]. Each abstract machine is completed by the definition of the needed operations. For example, the operation enter in the Controller abstract machine, as shown Figure 7, provokes the entrance of a person in the building. This event should only be able to happen (necessarily condition) if the person is authorised to be in the building and if he is not already inside. This event can happen, that does not necessarily mean that it is in fact going to happen. It is triggerable and therefore could be observed. The invariant of this machine says that the persons present in the building at a given moment do have the right to be there by stipulating that the set inside_cards is included (or equal) in the set authorised_cards.
   The different B abstract machine are presented in the Annex, with Figures 7, 8, 9 and 10.

2. **Simulation machine.**
   In this paper, we have not defined proof obligations corresponding to the **if** statement in the simulation machine. But we are able to simulate scenarios like the ones described using UML1.5 sequence diagrams.
   Figure 6 gives the simulation machine of the sequence diagram of Figure 5, integrating safety and dynamic properties introduced in the section 4.2. The initialisation gives a starting point for testing this scenario.

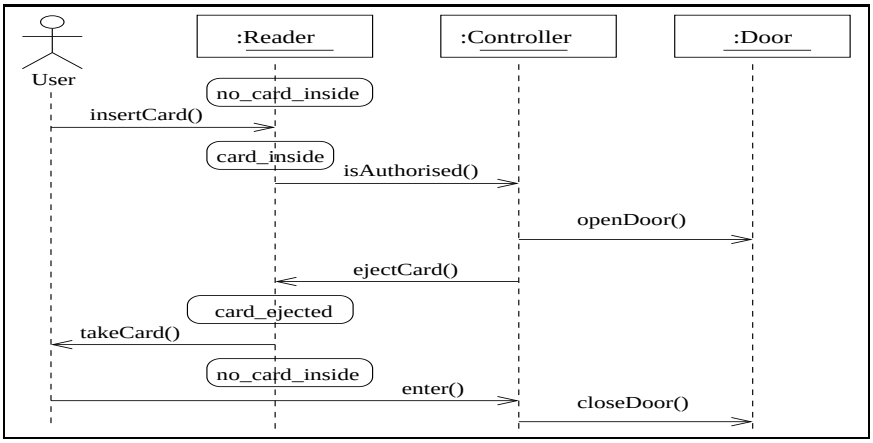

Fig. 5. A UML1.5 scenario for the authorised case of the entry to a building

4.4 Test of the scenario

This scenario does not modify the set of existing cards nor the set of authorised cards. Table 1 presents the evolution of the different variables when executing each step of the scenario. At the end of the execution, we can see that:

- the scenario is proved and
- the safety as well as the dynamic properties of the system are satisfied.

The principal property of the system which shows that the control is being done correctly, that is to say that each person is, at each moment, authorised to be in the building in which he finds himself is satisfied.

Operation i	state	door_state	inside_cards	current_card
0 (*Init*)	*no_card_inside*	*close*	$\{3 \mapsto c\}$	$\varnothing$
1 (*insertCard*)	*card_inside*	*close*	$\{3 \mapsto c\}$	$\{1 \mapsto a\}$
2 (*isAuthorised*)	*card_inside*	*close*	$\{3 \mapsto c\}$	$\{1 \mapsto a\}$
3 (*openDoor*)	*card_inside*	*open*	$\{3 \mapsto c\}$	$\{1 \mapsto a\}$
4 (*ejectCard*)	*card_ejected*	*open*	$\{3 \mapsto c\}$	$\varnothing$
5 (*takeCard*)	*no_card_inside*	*open*	$\{3 \mapsto c\}$	$\varnothing$
6 (*enter*)	*no_card_inside*	*open*	$\{3 \mapsto c, 1 \mapsto a\}$	$\varnothing$
7 (*closeDoor*)	*no_card_inside*	*close*	$\{3 \mapsto c, 1 \mapsto a\}$	$\varnothing$

Table 1. Evolution of the variables when executing the scenario

The result of this table can be tested with system properties by the automatic proof of the proposed proof obligations.

5 Related work

The concept of specification based testing has been initiated by the work of Hall [14]. His proposition, based on a Z specification, is to partition its input space by examining predicates in the operations.
BZ-TT [17, 5] is an environment for boundary-value test generation from Z and B specifications. The underlying method is based on a set-oriented constraint logic programming technology. Z and B specifications are translated into constraints and the constraint solver is used to calculate boundary value test cases. All the possible behaviors of the specification are tested at every boundary state using their input boundary values: the goal is to invoke each modification operation specified in the system, with extremum values of the sub-domains of its input parameters. The environment concentrates on testing the precondition and the execution of substitutions of each operation.
ProTest [28] is an automatic test environment for B specifications. It is based on ProB, a model checker and an animation tool for B [18]. It generates test cases from B specifications by partition analysis of the state invariant and the operation preconditions of a specification. It simultaneously animates the specification and runs the implementation with respect to the test cases and assigns verdicts whether the implementation has passed the tests. This test environment imposes some restrictions on arguments and results. ProTest animates and model checks only one single B machine. As BZ-TT, ProTest only checks the correctness of a single operation at a time and does not take into account dynamic properties.
With our proposition, we test the execution of a sequence of operations, including dynamic constraints on the system. These points are not been taken into the BZ-TT and ProTest approaches.
The use of scenario expressed by UML sequence diagrams to test object oriented specification is considered in [27]. This approach uses the *TeLa* language[1] where tests can be described using TeLa one-tier scenario, associating conditions on messages in the UML sequence diagram, or TeLa two-tier scenario diagrams, combining UML activity diagrams with sequence diagrams. Our proposition is inspired from this approach to test the execution of operations specified in B abstract machines.

6 Conclusion

We have presented an approach to test object-based specifications using B notations. We start from an UML specification in the form of a class diagram and a set of sequence diagrams expressing scenarios of the behavior of the system. The class diagram is then derived automatically into a B specification. This specification is completed by the definition of the operations (messages in the sequence diagrams corresponding to the methods in the class diagram) and by a new machine, called the simulation machine. This machine contains the derivation of the sequence diagram augmented by safety and dynamic properties of the system. At the end of the construction process, we have an object-based B specification at our disposal.

[1] Test Description Language

The test of scenarios and the satisfaction of the properties is done by means of a B theorem prover. In order to use this prover, we have defined the proof obligations for the simulation machine introduced as a new notation in B. The test is done in two steps:

- first, we test a scenario of the behavior of the system by a sequential execution of the operation calls expressed in this scenario;
- once the execution of this given scenario has been proved, we test safety and dynamic properties of the system.

Future work. We are working on the introduction of the **if** statement in the simulation machine in order to express UML2.0 sequence diagrams in order to take into account more complete scenarios.
The use of theorem provers presents some limits in the proof of predicates. As a perspective, we will study their replacement by a model checker, allowing to check all states of objects instantiated for the execution.
We are working on the implementation in Java of the generation of the new prof obligations corresponding to the new notations introduced in the B notations.

Acknowledgement. Our work benefits from interactions with many colleagues at LORIA and in particular with Samir Chouali.

References

1. M. Abadi and L.Cardelli. *A Theory of Objects*. Springer, 1996.
2. J-R. Abrial and D. Cansell. Click'n'Prove: Interactive Proofs Within Set Theory. In David Basin et Burkhart Wolff, editor, *16th International Conference on Theorem Proving in Higher Order Logics - TPHOLs'2003, Rome, Italy*, volume 2758 of *Lecture notes in Computer Science*, pages 1–24. Springer Verlag, 2003.
3. J.R. Abrial. *The B-Book, Assigning Programs to Meanings*. Cambridge University Press, 1996.
4. J.R. Abrial and L. Mussat. Introducing dynamic constraints in b. In *B'98: Recent Advances in the Development and Use of the B Method*, volume 1393 of *LNCS*, pages 83–128. Springer Verlag, 1998.
5. F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, N. Vacelet, and M. Utting. BZ-TT: A Tool-Set for Test Generation from Z and B using Constraint Logic Programming. In *Fomal Approaches to Testing of Software Workshop*, 2002.
6. H. R. Barradas and D. Bert. Specification and Proof of liveness properties under Fairness Assumptions in B Event Systems. In *Integrated Formal Method, IFM'02*, volume 2335 of *LNCS*, pages 360–379. Springer Verlag, 2002.
7. D. Bert, S. Boulmé, M-L. Potet, A. Requet, and L. Voisin. Adaptable Translator of B Specifications to Embedded C Programs. In *Integrated Formal Method, IFM'03*, volume 2805 of *LNCS*, pages 94–113. Springer Verlag, 2003.
8. G. Booch, J. Rumbaugh, and I. Jacopson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
9. E. Canver and F. W. Henke. Formal development of object-based systems in a temporal logic setting. In *Formal Methods for Open Object-Based Distributed Systems*, pages 419–436. IFIP, Kluwer Academic Publishers, 1999.
10. Clearsy. *B4free*. Available at *http://www.b4free.com*, 2004.
11. F. Dietrich. *Modelling and testing object-oriented communication services with temporal logic*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 2000.

12. D. Distefano, J-P. Katoen, and A. Rensink. On a temporal logic for object-based systems. In Scott F. Smith and Carolyn L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV - Proc. FMOODS'2000*. Kluwer Academic Publishers, 2000.
13. P. Facon, R. Laleau, and H.P. Nguyen. Mapping Object Diagram into B. In *Methods Integration Workshop*, Leeds, March 25-26 1996.
14. P.A.V. Hall. Relationship between Specifications and Testing. In *Information and Software technology*, 1991.
15. K. Lano, D. Clark, and K. Andoutsopoulos. UML to B: Formal Verification of Object-Oriented Models. In *Integrated Formal Methods (IFM)*, volume 2999 of *LNCS*, pages 187–256. Springer Verlag, 2004.
16. H. Ledang and J. Souquières. Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B. In *9th Asia Pacific Software Engineering Conference, APSEC'02*, Lecture Notes in Computer Science. Springer Verlag, 2002.
17. B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In *Formal Method Europe, FME'02*. Springer Verlag, 2002.
18. M. Leuschel and M. Butler. ProB: A model checker for B. In *Integrated Formal Method, IFM'03*. Springer Verlag, 2003.
19. B-Core(UK) Ltd. *B-Toolkit User's Manual*. Oxford (UK), 1996. Release 3.2.
20. A. Malioukov. An object-based approach to the B formal method. In *B'98: Recent Advances in the Development and Use of the B Method*, volume 1393 of *LNCS*, pages 162–181. Springer Verlag, 1998.
21. Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. Technical report, Stanford University, June 1991.
22. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.
23. E. Meyer and T. Santen. Behavioral Conformance Verification in an Integrated Approach Using UML and B. In *Integrated Formal Methods, IFM00*, volume 1945 of *LNCS*, page 358. Springer Verlag, 2000.
24. OMG. UML 2.0 Superstructure Specification. Available at http://www.omg.org, 2004.
25. OMG. *Unified Modeling Language*. OMG *http://www.omg.org/docs/formal/03-03-01.pdf*, Version 1.5 March 2003.
26. P.Behm, P. Benoit, and J.M. Meynadier. METEOR: A Successful Application of B in a Large Project. In *Integrated Formal Methods, IFM99*, volume 1708 of *LNCS*, pages 369–387. Springer Verlag, 1999.
27. S. Pickin and J.M Jézéquel. Using UML Sequence Diagrams as the Basis for a Formal Test Description Language. In *Integrated Formal Method, IFM'04*. Springer Verlag, 2004.
28. M. Satpathy, M. Leuschel, and M. Butler. ProTest: An automatic test environment for B specifications. In *International workshop on Model Based Testing*, 2004.
29. S. Schneider. *The B Method: An Introduction*. PALGRAVE, ISBN 0-333-79284-X, 2001.
30. Steria. *Obligations de preuve: Manuel de référence*. Steria - Technologies de l'information, version 3.0. Available at *http://www.atelierb.societe.com*.
31. D. A. Sykes and J. D. McGregor. *Practical guide to testing object-oriented software*. Addison Wesley, 2001.

7 Annex: B abstract machines

SIMULATION *Entry_Building*

INVARIANT
/** At any one moment, a person authorised to enter the building is either inside the building or outside */
$\forall xx.(xx \in cards)\ \Box(xx \in inside\_cards \vee xx \in cards - inside\_cards)$

MODALITIES
/** If a person input a card, this card will be ejected */
$\Box(state = card\_inside \Rightarrow \Diamond state = card\_ejected)$

/** The door is maintained closed until a person is authorised to enter*/
$\exists xx.(xx \in cards)\ \Box(state = no\_card\_inside \Rightarrow$
$door\_state = close\ \mathcal{W}\ xx \in authorised\_cards \wedge xx \notin inside\_cards)$

INITIALISATION
$cards := \{1 \mapsto a,\ 2 \mapsto b,\ 3 \mapsto c, 4 \mapsto d\ \} \parallel$
$authorised\_cards := \{1 \mapsto a,\ 2 \mapsto b,\ 3 \mapsto c\} \parallel$
$inside\_cards := \{3 \mapsto c\} \parallel$
$door\_state := close \parallel state := no\_card\_inside \parallel current\_card := \varnothing$

OPERATIONS
main =
begin
var $p1,\ p2$ **where**
$p1 \longleftarrow insertCard(1, a)$;
$p2 \longleftarrow isAuthorised$;
openDoor;
ejectCard;
takeCard;
enter(p1);
closeDoor
end
end
END

Fig. 6. *Simulation* machine for the authorised entry of a person to a building

MACHINE *Controller*

INCLUDES *Reader*, *Door*

VARIABLES
authorised_cards,
inside_cards,

INVARIANT
$authorised\_cards \subseteq cards \wedge$
$inside\_cards \subseteq authorised\_cards \wedge$

INITIALISATION
$authorised\_cards := \varnothing \parallel$
$inside\_cards := \varnothing \parallel$

OPERATIONS
$bb \longleftarrow isAuthorised(ca) =$
pre
$ca \in cards$
then
$bb := (ca \in authorised\_cards \wedge ca \notin inside\_cards)$
end;

$enter(ca) =$
pre
$ca \in cards$
then
if $isAuthorised(ca)$ **then**
$inside\_cards := inside\_cards \cup \{ca\}$
end
end

END

Fig. 7. *Control* abstract machine

MACHINE *Reader*

INCLUDES *Card*

SETS $READER\_STATE = \{no\_card\_inside, card\_inside, card\_ejected\}$

VARIABLES
$current\_card$,
$state$

INVARIANT
$current\_card \in cards \wedge$
$state \in READER\_STATE$

INITIALISATION
$current\_card := \varnothing \parallel$
$state := no\_card\_inside$

OPERATIONS

$ca \longleftarrow insertCard(pin, code) =$
pre
$pin \in PIN \wedge code \in CODE \wedge state = no\_card\_inside$
then
$current\_card := \{pin \mapsto code\} \parallel$
$state := card\_inside \parallel$
$ca := \{pin \mapsto code\}$
end;

$ejectCard =$
pre
$state = card\_inside$
then
$current\_card := \varnothing \parallel$
$state := card\_ejected$
end

$takeCard =$
pre
$state := card\_ejected$
then
$state := no\_card\_inside$
end

END

Fig. 8. *Reader* abstract machine

MACHINE *Card*
SETS $PIN = \{1, 2, 3, 4, 5, 6\}$;
$CODE = \{a, b, c, d, e\}$;
VARIABLES
cards
INVARIANT
$cards \in PIN \longrightarrow CODE$
INITIALISATION
$cards := \varnothing$
OPERATIONS

$createCard(pin, code) =$
pre
$pin \in PIN \wedge code \in CODE$
then
$cards := cards \cup \{pin \mapsto code\}$
end
END

Fig. 9. *Card* abstract machine

MACHINE *Door*
SETS
$DOOR\_STATE = \{open, close\}$
VARIABLES
door_state
INVARIANT
$door\_state \in DOOR\_STATE$
INITIALISATION
$door\_state := close$
OPERATIONS
$openDoor =$
pre
$door\_state = close$
then
$door\_state := open$
end;
$closeDoor =$
pre
$door\_state = open$
then
$door\_state := close$
end
END

Fig. 10. *Door* abstract machine