

Validation of UML scenarios using the B prover

Ninh-Thuan Truong and 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 validate object-based specifications by checking that sequence diagrams can be executed in the B world and do not conflict with safety and dynamic properties. The process begins by a UML specification in the form of a class diagram and sequence diagrams which express scenarios modelling the system's behaviour. 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), safety and dynamic properties on the system. The validation of scenarios and the satisfaction of the properties is done by means of a theorem prover. The approach is illustrated on a simplified case study: the access control of persons to a building.

Keywords: B, object-based specification, validation, verification, proof obligations, prover

1 Introduction

Experience has shown that the most critical and least supported phases of the software life cycle are requirement analysis and specification. Errors and misconceptions in the requirements will be passed on the system specifications and from them down the process to show up ultimately in the programs. Formal specifications could greatly help in reducing the amount of errors because of the absence of ambiguity in formal texts and the availability of powerful analysis techniques and prototyping tools [16]. However, formal specifications are hard to write and, more importantly, hard to read; this raises the problem of the validation of the specification. The validation consists in determining if the specification is an expression of the users' requirements. It requires users of the system to be able to "read" the specification, hence the importance of graphical notations. Walk-through or animation of specification are usual techniques to check that all the functional requirements have been taken into account and that the system overall behaviour has been adequately modelled. Graphical notations are well suited for these activities; support tools are available. Validating specifications by means of scenarios is worthwhile.

Object-oriented approaches [1, 9, 24] 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.

Object-based approaches based on B notations [25, 17, 22] usually integrate UML [27, 9] and B [3, 30]. 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 [28]. Dynamic constraints have been considered in B-Event [4, 6] and in object oriented approaches, establishing a link between time and object-orientation [12, 10]. 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 validate object-based specifications. It consists in checking that sequence diagrams can be executed in the B world and that they do not conflict with safety and dynamic properties on the whole system. The process begins by a UML specification in the form of a class diagram and sequence diagrams which express scenarios modelling the system's behaviour. These diagrams are transformed into a B specification allowing us to validate the consistency of the execution of each scenario expressed by a sequence diagrams and the simulation of scenarios 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 scenarios as sequences of operations calls,
- to integrate safety and dynamic constraints on the system in this new machine.

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 validate 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, generalised substitutions and

first order logic. The B method enables an incremental development process, known as a refinement process. A system development begins by the definition of an abstract view which can be refined step by step until an implementation is reached. The refinement over models is a key feature for developing incrementally models from a textually-defined system, while preserving correctness. It implements the proof-based development paradigm [21,31]. The method has been successfully used in the development of several complex real-life applications, like the METEOR project [7]. It is one of the few formal methods which has robust and commercially available support tools for the entire development life-cycle from specification down to code generation [8]. Specifications are composed of abstract machines which are very closed 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. Proofs for invariance and refinement are parts of each development. The proof obligations are generated automatically by support tools like AtelierB [31], B-Toolkit [21] and B4free [11], an academic version of AtelierB. The check of proof obligations with B support tools either through automatic or interactive proofs [2], is an efficient and practical way to detect errors introduced during the specification development.

2.2 Transformation of UML classes into B

The transformation of UML into B [14, 22, 25, 18] 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, see 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*.

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 introduce a new machine in B, namely the simulation machine, in order to express scenarios 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.

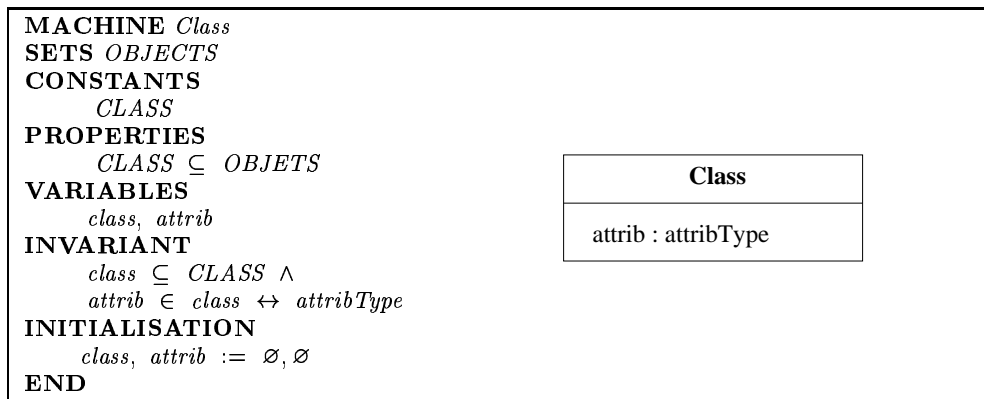


Fig. 1. Derivation of a UML class into B

3.1 Structure of the simulation machine

The basic idea is to simulate UML2.0 sequence diagrams which describe the interaction between messages. These diagrams may include guards: when modelling object interactions, there will be times when a condition must be met for a message to be sent to the object. Guards are not sufficient to handle the logic required for a sequence being modelled. We take into account the notation element called Combined Fragments in UML2.0, used to group sets of messages together to show conditional flow in a sequence diagram, namely alternative and option choices. Alternatives are used to designate a mutually exclusive choice between two or more message sequences, modelling the “*if then else*” statement. The option combination fragment is used to model a sequence that, given a certain condition, will occur; otherwise, the sequence does not occur. It is used to model a simple “*if then*” statement.

Scenarios described by UML2.0 sequence diagrams are transformed into a simulation machine, as presented Figure 2. Each message in an object-based system is specified by a B operation in an abstract machine. The simulation machine is composed of four 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 **SCENARIOS** contains the definition of different scenarios. It corresponds to the transformation of UML2.0 sequence diagrams, each one composed of a sequence of operation calls, including guard, alternative and option statements.

```

SIMULATION System
INVARIANT
  □ J
MODALITIES
  □ (P ⇒ ◇ Q)
  □ (P ⇒ Q W R)
INITIALISATION
  Init
SCENARIOS
  scenario-name(param) =
  begin
    OP
  end
  ...
END

```

Fig. 2. Structure of the simulation machine

3.2 Proof obligations associated to the scenarios clause

As presented above, a scenario is specified by a combination of sequence operation calls, including guards, alternatives and options to control flow and take into account choices.

```

MACHINE Class
SETS T
CONSTANTS C
PROPERTIES Prop
VARIABLES V
INVARIANT I
INITIALISATION U
OPERATIONS
  oo ← Op(ii) = pre P then S end;
  ...
END

```

Fig. 3. B abstract machine derived from a UML class

Let us recall the definition of proof obligations [3] for a B abstract machine with the clauses presented Figure 3. A number of proof obligations have to be checked to ensure that the machine is internally consistent and useful. In particular, it is necessary to ensure that:

- the initialisation U is guaranteed to establish the invariant I , under the assumption that $Prop$ holds, that the context of the machine is satisfactory:

$$T \wedge Prop \Rightarrow [U]I$$

- each operation, composed of a precondition P and a substitution body S , preserves the invariant I

$$T \wedge Prop \wedge I \wedge P \Rightarrow [S]I$$

(i). Proof obligations associated to a scenario defined by a sequence of operation calls

Let us consider the case where a scenario is specified by a sequence of n operation calls. Each one is indexed by i and composed of a precondition P and a substitution body S :

$$OP = OP_1; OP_2; \dots; OP_n$$

$$OP_i = P_i \mid S_i$$

so

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

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:

$$r_i \leftarrow OP_i(para_1, para_2, \dots, para_m)$$

and its call is of the form:

$$v_i \leftarrow OP_i(value_1, value_2, \dots, value_m)$$

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 :

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

After replacing each parameter in the body of the operation by its value, we obtain:

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

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:

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

A scenario 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 of this scenario is validated if the following proof obligations are verified:

– **For** $i = 0 .. n-1$.

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. In object-based approaches, 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:

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

– **For** $i = n$.

The proof obligations guarantee that the execution of the scenario preserves the invariant I of all the abstract machines:

$$A \wedge Prop \wedge I \wedge P_{nv} \Rightarrow [S_v^n]I \quad (2)$$

(ii). Proof obligations associated to a scenario which includes if statements

Let us now consider the case where a scenario is specified by a sequence of operation calls including *if* statements. A general conditional statement which allows branching depending on a particular value is defined as follows:

IF P THEN Q ELSE R END

This construction makes use of the interaction between choice and guards. In providing a weakest precondition rule for this construct, there are two cases to consider. In order to ensure that T will be true after its execution, if P is true then Q must establish T , and if P is false, then R must establish T . This results in the following rule:

$$[\mathbf{IF } P \mathbf{ THEN } Q \mathbf{ ELSE } R \mathbf{ END}]T = (P \Rightarrow [Q]T) [] (\neg P \Rightarrow [R]T)$$

Let us consider the case where both Q and R denote a list of operations, expressed by:

$$\begin{aligned} Q &= OP_{q1}; \dots; OP_{qq} \\ R &= OP_{r1}; \dots; OP_{rr} \end{aligned}$$

The scenario including a *if* statement is expressed by:

$$\begin{aligned} &OP_1; OP_2; \dots; OP_i; \\ &\mathbf{IF } P \mathbf{ THEN } OP_{q1}; \dots; OP_{qq} \mathbf{ ELSE } OP_{r1}; \dots; OP_{rr} \mathbf{ END}; \\ &OP_{i+1}; \dots; OP_n \end{aligned}$$

Two cases have to be considered:

- when $[S_1][S_2]\dots[S_i](P) = true$, the path of the execution of the scenario is:

$$OP_1; OP_2; \dots OP_i; OP_{q1}; \dots; OP_{qq}; OP_{i+1}; \dots; OP_n$$

- when $[S_1][S_2]\dots[S_i](P) = false$, the path of the execution of the scenario is:

$$OP_1; OP_2; \dots OP_i; OP_{r1}; \dots; OP_{rr}; OP_{i+1}; \dots; OP_n$$

For each case of the decomposition of the scenario, we obtain a sequence of operations, going back to the proof obligations of a scenario defined by a sequence of operation calls, see (1).

Remark. Guarded and option statements allowed in UML2.0 sequence diagrams are included in the *if* statement. They both correspond to a simple “*if P then Q*” statement:

- in the case of a guarded statement, Q corresponds to a single message,
- in the case of an option statement, Q corresponds to a sequence of messages.

3.3 Expression of system dynamic properties and proof obligations

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

- safety properties in the clause INVARIANT and
- dynamic properties in the clause MODALITIES. We introduce liveness properties [23], which have been already introduced in object-oriented notations [12, 13, 32].

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.

$$\forall i.(i \in [0..n] \Rightarrow A \wedge Prop \wedge I \wedge P_{iv} \Rightarrow [S_v^i]J) \quad (3)$$

B. Liveness properties. We consider two kinds of liveness property, the response property and the precedence property [12].

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:

$$\exists i.(i \in [0..n-1] \wedge A \wedge Prop \wedge I \wedge P_{iv} \Rightarrow [S_v^i]P) \quad (4)$$

This first proof obligation for this property (4) 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 (5):

$$\exists j.(j \in [i+1..n] \wedge A \wedge Prop \wedge I \wedge P_{jv} \Rightarrow [S_v^j]Q) \quad (5)$$

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:

$$\exists i.(i \in [0..n-1] \wedge A \wedge Prop \wedge I \wedge P_{iv} \Rightarrow [S_v^i]P) \quad (6)$$

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:

$$\exists j.(j \in [i+1..n] \wedge A \wedge Prop \wedge I \wedge P_{jv} \Rightarrow [S_v^j]R) \quad (7)$$

- 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 :

$$\forall j.(j \in [i+1..n] \Rightarrow A \wedge Prop \wedge I \wedge P_{jv} \Rightarrow [S_v^j]Q) \quad (8)$$

- 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:

$$A \wedge Prop \wedge I \wedge P_{jv} \Rightarrow [S_v^j](\neg Q) \quad (9)$$

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

The 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 behaviour. 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 validate the execution of scenarios of the system. This is done in two steps:

- first, we prove a scenario of the behaviour of the system defined as a UML2.0 sequence diagram,
- once the execution of this given scenario has been proven, we prove that 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 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.

The person 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.

The system will control if persons wishing to enter the building are not already in the building.

4.2 UML Specification

We first introduce a UML class diagram to structure this system. Then we model the proposed system's behaviour by means of a sequence diagram. We clarify 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`: at any moment, we know which is inside the building.

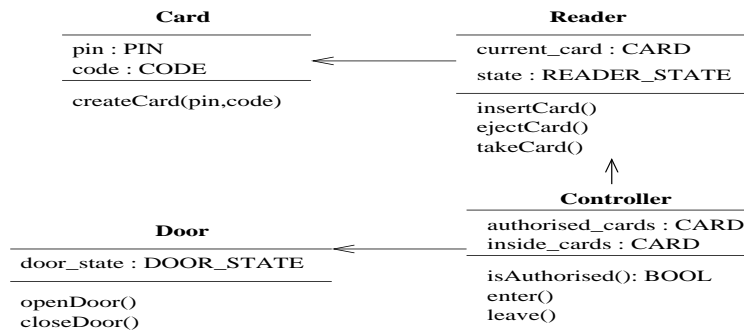


Fig. 4. Class diagram of the simplified access control system

2. Sequence diagram.

As an example of scenario, we propose to describe the entry of a person into the building. It is presented in Figure 5 by a UML2.0 sequence diagram [26]. This scenario is parameterised by a card, known by means of a pin and a code. In the sequence diagram, the “alt” combined fragment corresponds to the two cases:

- the person (i.e. the card) is authorised to enter,
- the person is not authorised to enter.

Remark. A person is authorised to enter if he is known of the system and he is not inside.

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 inputs a card, this card will be ejected,*
- *the door is maintained closed until a person is authorised to enter.*

4.3 B specification

1. Abstract machines.

The class diagram of the Figure 4 is derived into a B specification using automatic transformation rules [25]. 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 6, 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

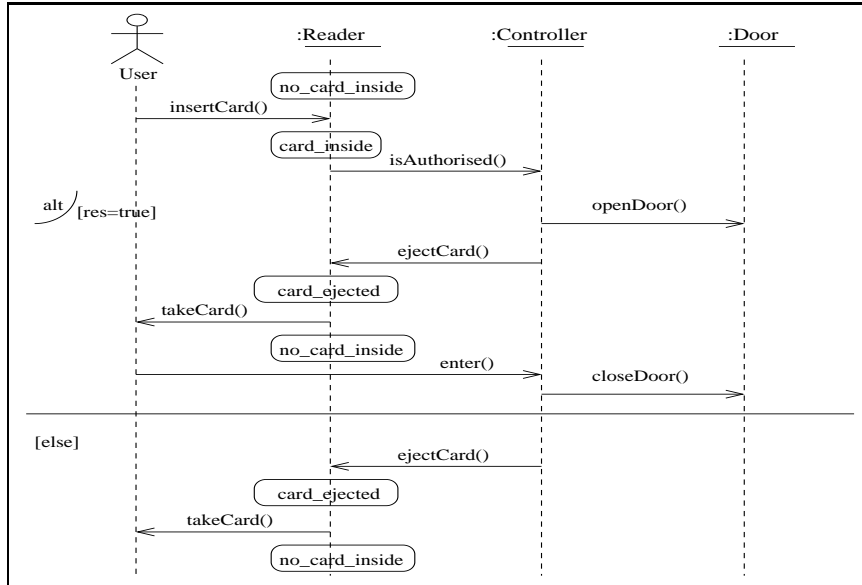


Fig. 5. Scenario for the entry to a building

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`.

2. Simulation machine.

Figure 7 gives the simulation machine of this system with one scenario corresponding to the entry to the building described in the sequence diagram presented Figure 5. Constraints on the system are expressed by the invariant and the modalities clauses, including safety and dynamic properties presented in the section 4.2. The initialisation gives a starting point for validating this scenario.

4.4 Validation of the scenario *Entry_Building*

The scenario proposed Figure 7 does not modify the set of existing cards nor the set of authorised cards. Giving a card, i.e. a pin and a code, it first verifies if the card is authorised to enter or not the building. To simulate each situation, we have to introduce three test cases.

(i.) Normal case where the person is authorised to enter the building.

Table 1 presents the evolution of the different variables of the system when executing each step of the scenario *Entry_Building* for a given card, *pin = 1*, *code = a*. The column *Precondition of the operation* recalls, for each operation of the

```

MACHINE Controller
INCLUDES Reader, Door
VARIABLES authorised_cards, varsinside_cards
INVARIANT  $authorised\_cards \subseteq cards \wedge inside\_cards \subseteq authorised\_cards$ 
INITIALISATION  $authorised\_cards := \emptyset \parallel inside\_cards := \emptyset$ 
OPERATIONS
bb  $\leftarrow isAuthorised(ca) =$ 
  pre  $ca \in cards$ 
  then  $bb := (ca \in authorised\_cards - inside\_cards)$ 
  end;
enter(ca) =
  pre  $ca \in (authorised\_cards - inside\_cards)$ 
  then  $inside\_cards := inside\_cards \cup \{ca\}$ 
  end
END

```

Fig. 6. *Control abstract machine*

scenario, its precondition defined in the B abstract machines. We can see that they are satisfied for each operation call.

The execution of this scenario is validated if the proof obligations presented section 3.2 are proved. Let us see the proof obligation (1). As an example, we can see that after the execution of substitutions in the body of the first 4 operations of the scenario, $[S_v^4]$, the value of the variable *state* which shows the state of the card reader is $state = card_ejected$. This state satisfies the precondition of the next operation of the scenario, namely *takeCard* ($i=5$).

At the end of the execution of the scenario the invariant of the system *Controller* is satisfied, i.e. $inside_cards \subseteq authorised_cards$. This corresponds to the proof obligation (2).

Proof of the invariant: a safety property.

```

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

```

The proof obligation (3) is satisfied by the execution of the scenario.

Proof of the modalities: liveness properties.

The response property is expressed by:

```

/** If a person inputs a card, this card will be ejected */
 $\square(state = card\_inside \Rightarrow \diamond state = card\_ejected)$ 

```

In the introduction of response properties given Figure 2,

P corresponds to $(state = card_inside)$ and

Operation i	Precondition of the operation	$state$ variable	$door_state$ variable	$inside_cards$ variable	$current_card$ variable
0 (<i>Init</i>)	–	<i>no_card_inside</i>	<i>close</i>	$\{3 \mapsto c\}$	\emptyset
1 (<i>insertCard</i>)	$state = no_card_inside$	<i>card_inside</i>	<i>close</i>	$\{3 \mapsto c\}$	$\{1 \mapsto a\}$
2 (<i>isAuthorised</i>)	$card \in cards$	<i>card_inside</i>	<i>close</i>	$\{3 \mapsto c\}$	$\{1 \mapsto a\}$
authorised = true					
3 (<i>openDoor</i>)	$door_state = close$	<i>card_inside</i>	<i>open</i>	$\{3 \mapsto c\}$	$\{1 \mapsto a\}$
4 (<i>ejectCard</i>)	$state = card_inside$	<i>card_ejected</i>	<i>open</i>	$\{3 \mapsto c\}$	\emptyset
5 (<i>takeCard</i>)	$state = card_ejected$	<i>no_card_inside</i>	<i>open</i>	$\{3 \mapsto c\}$	\emptyset
6 (<i>enter</i>)	$authorised = true$	<i>no_card_inside</i>	<i>open</i>	$\{3 \mapsto c, 1 \mapsto a\}$	\emptyset
7 (<i>closeDoor</i>)	$door_state = open$	<i>no_card_inside</i>	<i>close</i>	$\{3 \mapsto c, 1 \mapsto a\}$	\emptyset

Table 1. Validation of the scenario *Entry_Building* with an authorised card

Q to ($state = card_ejected$)

The proof obligation (4) is established for $i = 1$:

$$[S_v^1]P = true$$

The proof obligation (5) is established for $j = 4$:

$$[S_v^4]Q = true$$

The precedence property is expressed by:

```
/** The door is maintained closed until a person is authorised to enter */
∃ xx.(xx ∈ cards) □ (state = no_card_inside ∧ door_state = close ⇒
  door_state = close ∧
  (xx ∈ authorised_cards − inside_cards ∧ door_state = open))
```

In the introduction of precedence properties given Figure 2,

P corresponds to ($state = no_card_inside \wedge door_state = close$)
 Q to ($door_state = close$)
 R to ($xx \in authorised_cards - inside_cards \wedge door_state = open$)

The proof obligation (6) says that P is established for $i = 0$, at the initialisation. The proof obligation (7) says that R is established for $j = 3$, with the operation *OpenDoor*. The proof obligation (9) is proved for $j = 3$: $[S_v^3] \neg Q$. The proof obligation (8) is proved for $j = 1..2$:

$$\forall j.(j \in [1..2] \Rightarrow [S_v^j]Q)$$

(ii.) Case where the person is not authorised to enter the building.

Table 2 presents the evolution of the different variables of the system when executing each step of the scenario *Entry_Building* for a non authorised card: *pin*

Operation i	Precondition of the operation	$state$ variable	$door_state$ variable	$inside_cards$ variable	$current_card$ variable
0 (<i>Init</i>)	-	no_card_inside	$close$	$\{3 \mapsto c\}$	\emptyset
1 (<i>insertCard</i>)	$state = no_card_inside$	$card_inside$	$close$	$\{3 \mapsto c\}$	$\{1 \mapsto f\}$
2 (<i>isAuthorised</i>)	$card \in cards$	$card_inside$	$close$	$\{3 \mapsto c\}$	$\{1 \mapsto f\}$
authorised = false					
3 (<i>ejectCard</i>)	$state = card_inside$	$card_ejected$	$close$	$\{3 \mapsto c\}$	\emptyset
4 (<i>takeCard</i>)	$state = card_ejected$	no_card_inside	$close$	$\{3 \mapsto c\}$	\emptyset

Table 2. Validation of the scenario *Entry_Building* with a non authorised card

= 1, code = f.

(iii.) Case where the person is already inside the building.

Table 3 presents the evolution of the different variables of the system when executing each step of the scenario *Entry_Building* for a card already inside the building: $pin = 3$, code = c.

Operation i	Precondition of the operation	$state$ variable	$door_state$ variable	$inside_cards$ variable	$current_card$ variable
0 (<i>Init</i>)	-	no_card_inside	$close$	$\{3 \mapsto c\}$	\emptyset
1 (<i>insertCard</i>)	$state = no_card_inside$	$card_inside$	$close$	$\{3 \mapsto c\}$	$\{1 \mapsto f\}$
2 (<i>isAuthorised</i>)	$card \in cards$	$card_inside$	$close$	$\{3 \mapsto c\}$	$\{1 \mapsto f\}$
authorised = false					
3 (<i>ejectCard</i>)	$state = card_inside$	$card_ejected$	$close$	$\{3 \mapsto c\}$	\emptyset
4 (<i>takeCard</i>)	$state = card_ejected$	no_card_inside	$close$	$\{3 \mapsto c\}$	\emptyset

Table 3. Validation of the scenario *Entry_Building* with a card already inside

At the end of the execution of the scenario with the three test cases, 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.

The result of the three tables can be validated with system properties by the automatic proof of the proposed proof obligations.

5 Related work

The use of scenario expressed by UML sequence diagrams to validate object oriented specification is considered in [28]. This approach uses the *TeLa* language¹ 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.

The concept of specification based testing has been initiated by the work of Hall [15]. His proposition, based on a Z specification, is to partition its input space by examining predicates in the operations.

BZ-TT [19, 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 behaviours 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 [29] is an automatic test environment for B specifications. It is based on ProB, a model checker and an animation tool for B [20]. 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 validate the execution of a sequence of operations, and check that they do not conflict with safety/dynamic constraints of the system. These points are not been taken into account the BZ-TT and ProTest approaches.

6 Conclusion

We have presented an approach to validate object-based specifications checking that sequence diagrams can be executed in the B world and do not conflict with safety and dynamic properties. We start from an UML specification in the form of a class diagram and a set of sequence diagrams expressing scenarios of the behaviour 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

¹ Test Description Language

machine contains the derivation of sequence diagrams 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.

The validation 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 validation is done in two steps:

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

Future work.

In order to take into account more complex scenarios, we have to introduce loop statement in the scenarios. Another question to solve concerns the possibility of introducing an initialisation specific to each scenario.

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 proof obligations corresponding to the simulation machine.

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 D. Basin et B. Wolff, editor, *16th International Conference on Theorem Proving in Higher Order Logics - TPHOLs'2003*, volume 2758 of *LNCS*, 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 *Formal 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. 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.
8. 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.
9. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

10. 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.
11. Clearisy. *B4free*. Available at <http://www.b4free.com>, 2004.
12. F. Dietrich. *Modelling and testing object-oriented communication services with temporal logic*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 2000.
13. 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.
14. P. Facon, R. Laleau, and H.P. Nguyen. Mapping Object Diagram into B. In *Methods Integration Workshop*, Leeds, March 25-26 1996.
15. P.A.V. Hall. Relationship between Specifications and Testing. In *Information and Software technology*, 1991.
16. Michael G. Hinchey and Jonathan P. Bowen. *Applications of Formal Methods*. Prentice Hall, 1995.
17. 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.
18. 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.
19. B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In *Formal Method Europe, FME'02*. Springer Verlag, 2002.
20. M. Leuschel and M. Butler. ProB: A model checker for B. In *Integrated Formal Method, IFM'03*. Springer Verlag, 2003.
21. B-Core(UK) Ltd. *B-Toolkit User's Manual*. Oxford (UK), 1996. Release 3.2.
22. 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.
23. Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. Technical report, Stanford University, June 1991.
24. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.
25. 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.
26. OMG. UML 2.0 Superstructure Specification. Available at <http://www.omg.org>, 2004.
27. OMG. *Unified Modeling Language*. OMG [http : //www.omg.org/docs/formal/03-03-01.pdf](http://www.omg.org/docs/formal/03-03-01.pdf), Version 1.5 March 2003.
28. 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.
29. M. Satpathy, M. Leuschel, and M. Butler. ProTest: An automatic test environment for B specifications. In *International workshop on Model Based Testing*, 2004.
30. S. Schneider. *The B Method: An Introduction*. PALGRAVE, ISBN 0-333-79284-X, 2001.
31. Steria. *Obligations de preuve: Manuel de référence*. Steria - Technologies de l'information, version 3.0. Available at <http://www.atelierb.societe.com>.
32. D. A. Sykes and J. D. McGregor. *Practical guide to testing object-oriented software*. Addison Wesley, 2001.

```

SIMULATION Access_Control

INVARIANT
/** At any one moment, a person authorised to enter the building is either inside the
building or outside */
∀ xx.(xx ∈ authorised_cards)
    □(xx ∈ inside_cards ∨ xx ∈ authorised_cards - inside_cards)

MODALITIES
/** If a person inputs a card, this card will be ejected */
□(state = card_inside ⇒ ◇state = card_ejected)

/** The door is maintained closed until a person is authorised to enter*/
∃ xx.(xx ∈ cards) □(state = no_card_inside ∧ door_state = close ⇒
    door_state = close ∨
    (xx ∈ authorised_cards - inside_cards ∧ door_state = open))

INITIALISATION
cards := {1 ↦ a, 2 ↦ b, 3 ↦ c, 4 ↦ d } ||
authorised_cards := {1 ↦ a, 2 ↦ b, 3 ↦ c } ||
inside_cards := {3 ↦ c } ||
door_state := close ||
state := no_card_inside ||
current_card := ∅

SCENARIOS
Entry_Building(pin, code) =
    begin
        var card, authorised in
            card ← insertCard(pin, code);
            authorised ← isAuthorised(card);
            if authorised then
                openDoor;
                ejectCard;
                takeCard;
                enter(card);
                closeDoor
            else
                ejectCard;
                takeCard
            end
        end
    end
END

```

Fig. 7. Simulation machine with one scenario