A Formal Framework for UML Modelling with Timed Constraints: Application to Railway Control Systems

Rafael Marcano, Samuel Colin and Georges Mariano

National Institute for Transport and Safety Research
INRETS-ESTAS
20 rue Elisée Reclus - B.P. 317
F-59666 Villeneuve d'Ascq, France
{marcano,colin,mariano}@inrets.fr

Abstract. In the context of railway signalling systems, time related features play a relevant role at the validation process and specialists are more and more confronted with the necessity of applying formal methods as mean of preventing software faults. UML offers a standard notation for high quality systems modelling, however its current lack of formal semantics explains the existence of few tools supporting analysis and verification. In this paper, we propose a formal support of UML model-based verification using time-extended B specifications. The main goal is to enable consistency checking through UML diagrams using existing tools for B. A level crossing control system is developed in order to illustrate the approach.

Keywords: UML, B formal analysis, real-time, requirements engineering.

1 Introduction

The software engineering industry has devoted a relevant effort to the development of standardized design languages and methods such as UML [1], but so far it has dedicated much less attention to the integration of these design technologies with the verification and validation techniques as formal methods. More effort has been spent to develop new languages than to provide methodological guidance for using existing ones. The UML notation offers a standard language for high quality systems modelling. However, the current lack of formal semantics for UML explains the existence of few tools supporting analysis and verification of real-time and embedded systems, particularly in the context of railway applications. Although specialists in the train control systems domain are more and more confronted with the necessity of applying formal methods [2], they are often not familiar with formal specification techniques. Consequently, the use of formal methods is actually non-standard practice even in safety-critical applications.

In this paper we propose an approach aiming at the extension of UML-based analysis process with model based verification using the B formal method [3]. The integration of semi-formal notations with formal methods is motivated by their complementary strengths. Combining UML with formal methods has a dual benefit:

- The UML notation lacks of mathematical foundations, which can be provided by formal methods. Thus, analysis and consistency checking of UML models is allowed using mathematical techniques and automated proof tools.

– The notational complexity of the B method is often stronger than its advantages. Providing an UML support to the B notation could significantly facilitate the specification process contributing to decrease the “explication problem”.

We propose to manipulate in parallel a UML/OCL model and its associated B specification, which is automatically generated. The main goal is to enable the verification, validation and simulation of the UML/OCL model of a system in the same way that is done for a B specification. B is a complete formal method supporting specification, refinement and implementation steps of the life cycle through a unified notation. It allows a complete implementation of the software system in C, C++ or ADA code. It has already been used in significant industrial projects and commercial case tools are available. Here we consider an extended version of the B language allowing temporal properties specification.

We have carried previous work on combining UML and B for consistency checking in [4, 5]. Some related work about transformation of UML diagrams into B formal specifications has been presented in [6] and [7]. In [6] authors propose templates to derive B specifications from UML diagrams. In [7] translation rules mapping UML diagrams into B specifications based on a UML metamodel are presented. However, the objective of these works is to facilitate the construction of a formal specification, whereas our main purpose is to enable analysis and verification of UML. Moreover these works do not take into account OCL annotations [8] neither time related constraints.

The paper is structured as follows. We first describe the proposed process (section 2). In section 3, a railway level crossing control system is described using UML based modelling. Then we formalize the UML class and state diagrams, as well as OCL constraints, using B formal specifications (section 4). In section 5, we consider the time constraints in the B specification. The analysis and verification steps are discussed in section 6.

2 The Proposed Process

The aim of our project is not to design a completely new specification notation or method, but rather to integrate two existing ones: UML/OCL and B. This integration is the basis of our process, which is composed by three main steps as shown in Fig.1:

Requirements elicitation. At this step, the software system begins to be clarified. Both its global properties and its interactions with its environment have to be sketched. In order to guarantee the correction of the requirements, that is their consistency and their completeness, it is important to be able to put together all the information distributed among the different documents already written. To do so, we will translate them into UML diagrams: first, sequence diagrams will be defined in order to describe some typical behaviour, and second, state diagrams are used to describe the global interaction of a system with its environment. Both kinds of diagrams are annotated with OCL constraints.

Formal specification. The objective of this step is to translate into mathematics all the facts, hypothesis and needs described by the UML model. The classes and associations are translated into B abstract machines. The OCL constraints are used to

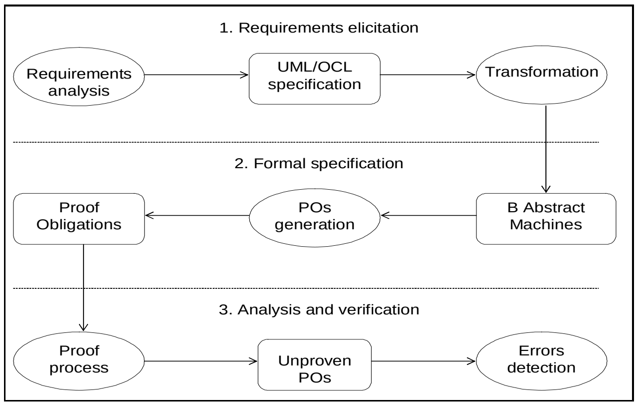

Fig. 1. The proposed process

define the invariants, as well as pre/post-conditions on the operations of abstract machines. The state diagrams allow these operations to be completed.

Analysis and verification. The main feature of the B specification is to allow the generation of Proof Obligations (POs). POs can be proven automatically using existing proof tools. Unproved POs may reveal inconsistencies on the system model, that is wrong invariants as well as erroneous pre/post-conditions. POs are a useful mean to perform model-based analysis at specification level. Each PO has to be interpreted in order to find possible mistakes, i.e. operations that do not preserve the invariant. Then, sequence diagrams are used to define test scenarios associated to these operations.

In the following, we will illustrate each of these steps by developing a railway level crossing system.

3 From Requirements to UML models

The ”explication problem”, i.e. the creation of a first valid description of the required system properties, is a decisive activity in order to ensure the correctness of the future system. The consistency of the system relies on the developer’s ability to capture its key safety properties. Therefore, the knowledge of the related domain plays an important role in the requirements elicitation activity. Our approach of requirement analysis is based on [9] where both static and dynamic properties are taken into account by different UML diagrams.

The comprehension of the static properties is obtained by describing involved entities and their invariant. Here, the Object Constraint Language is used to express all the properties that cannot be expressed using only diagrammatic notation, i.e. hypotheses and facts on subsystems, classes, attributes and associations. Safety conditions are also described at this step using OCL. The comprehension of the dynamic behaviour of the future system is obtained by the description of the manner the actors will interact with it. Two steps are necessary: first, UML sequence diagrams will be defined in order to describe both usual and failure scenarios. Second, the complete expected behaviour of the system will be described as a set of OCL pre and post-conditions on operations. Putting together all the operations, UML state diagrams will be defined to describe the global interaction of a system with each actor of its environment.

3.1 Specification of the RLC system

A complete description of the traffic control system considered here is given in [10]. This description includes also domain knowledge as a basis for formal specification. The problem is the specification of a radio-based Railway Level Crossing (RLC) application that has been developed for the German Railways [11]. It is distributed over three subsystems: a train-borne control system (on-board system), a level crossing control system and an operations centre. The level crossing is situated at a single-track railway line and a road crossing at the same level. The intersection area of the road and the railway line is called danger zone, since trains and road traffic must not enter it at the same time. Note that this is the main safety constraint that will be taken into account during the description of the system.

3.2 UML-based modelling

The whole system is composed of three subsystems which communicate each other. Our study begins by adopting a centric approach regarding the Level Crossing Control subsystem (called LCC). From this view point, the Trainborne Control system (TC) and the Operation Centre (OC) are actors cooperating and making use of the LCC.

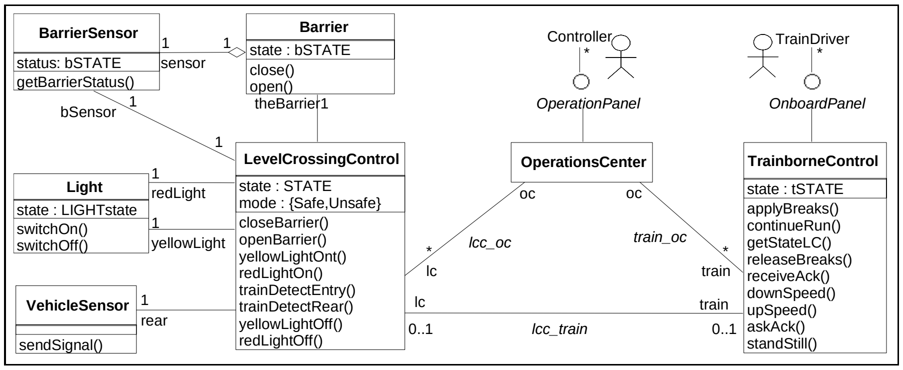

Fig. 2. Class diagram of the RLC system

At this stage, the main entities of interest to be modelled regarding possible failure conditions of the LCC system have to be identified. A main cause of failures is the malfunctioning of sensors or actuators. Defects may occur in the main physical structures, but also control systems themselves may fail. In the case study only a limited number of failures are regarded: failures of yellow or red traffic light (to be regarded separately), the barriers, the vehicle sensor and the delay or loss of telegrams on the radio network. Consequently, we consider the following objects interacting with the LCC system, as shown in Fig.2: the lights, the barriers, the vehicle sensors, the trainborne control system and the operations centre. We consider here only one-side railway line of the level crossing in order to make more readable the specification of the system.

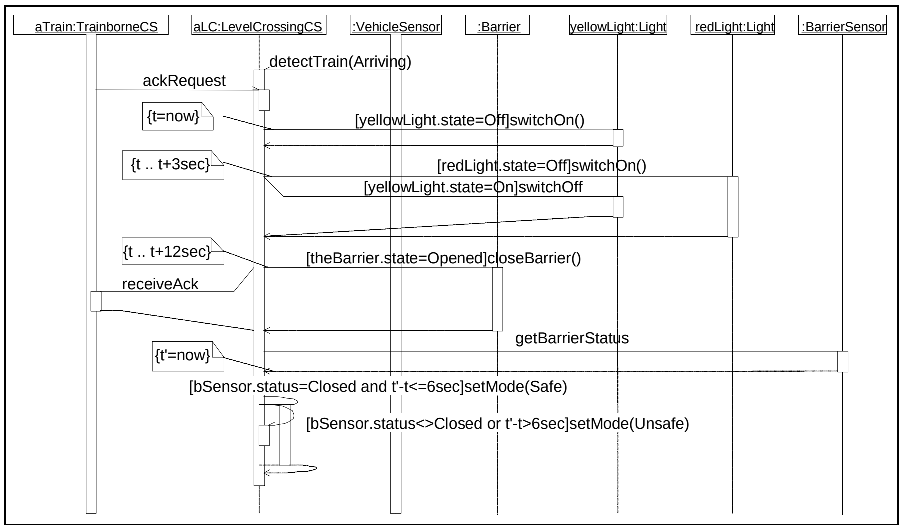
aTrain:TrainborneCS
aLC:LevelCrossingCS
:VehicleSensor
:Barrier
yellowLight:Light
redLight:Light
:BarrierSensor
detectTrain(Arriving)
ackRequest
{t=now}
[yellowLight.state=Off]switchOn()
{t .. t+3sec}
[redLight.state=Off]switchOn()
[yellowLight.state=On]switchOff
{t .. t+12sec}
[theBarrier.state=Opened]closeBarrier()
receiveAck
getBarrierStatus
{t'=now}
[bSensor.status=Closed and t'-t<=6sec]setMode(Safe)
[bSensor.status<>Closed or t'-t>6sec]setMode(Unsafe)

Fig. 3. Sequence diagram - scenario of train approaching

The traffic lights and barriers at the level crossing are controlled by the LCC system. The LCC system has to be activated when a train is approaching the level crossing. In the activated mode a sequence of actions are performed by the LCC at a specific timing in order to safely close the crossing and to ensure the danger zone to be free of road traffic. First, the traffic lights are switched on to show the yellow light, then after 3 seconds they are switched to red. After some further 9 seconds the barriers are started to be lowered. The LCC system signals the safe state of the level crossing if the barriers have completely been lowered within a maximum time of 6 seconds, allowing the train to pass the level crossing (Fig.3).

The level crossing may be opened again for road traffic when the train has completely passed the crossing area and the LLC system switches back to the deactivated mode. The detection of a train approaching at the level crossing is based on continuous self-localisation of the train and radio-based communication between the train and the LCC system. Triggering the vehicle sensor at the rear of the level crossing will allow the barriers to be opened again and the traffic lights to be switched off. In the activated mode the LCC system may be in one of the following substates (Fig.4): showing the yellow light; closing the barrier; retaining the barrier closed; or opening the barrier. Note that time expirations occurring after the activation of the LCC are denoted by the following events: timeOut_1 (3 seconds later), timeOut_2 (9 seconds after timeOut_1) and timeOut_3 (6 seconds after timeOut_2).

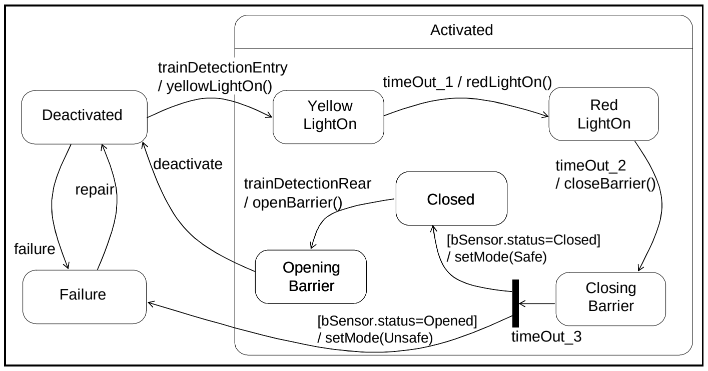

Fig. 4. State diagram of the LCC system

When a train is approaching the level crossing, it sets a braking curve for speed supervision making the train stop at the potential danger point in failure situation. The LCC system acknowledges receipt of the activation order to the train. After receipt of the acknowledgement the TC system waits an appropriate time for the level crossing to be closed and then sends a status request to the LCC system. If the level crossing is in its safe state it will be reported to the train. This will allow the train to cancel the braking curve and safely pass over the level crossing. This scenario is illustrated in the sequence diagram of Fig.5. The sequence diagrams include OCL constraints which are used to define pre-conditions on operations and to define some time properties.

3.3 Adding OCL constraints

Using a standard formal language for constraint specification is an important step towards formalising complex models, particularly in the context of safety critical systems. The purpose of OCL (Object Constraint Language) is to allow constraints on the objects of a system to be formally specified, preserving the comprehensibility and readability of the UML models. It facilitates to express the properties and invariants on the objects and the pre/post-conditions on the operations. OCL provides a navigation mechanism allowing attributes, operations and associations to be referenced in the context of a class or an object (a class variable). It includes query operators permitting to select and/or modify a set of elements. Each OCL expression has a specific type and belongs to a specific context. The context of an OCL expression determines its scope. Only the visible elements in the context of the expression can be referenced by means of navigation expressions.

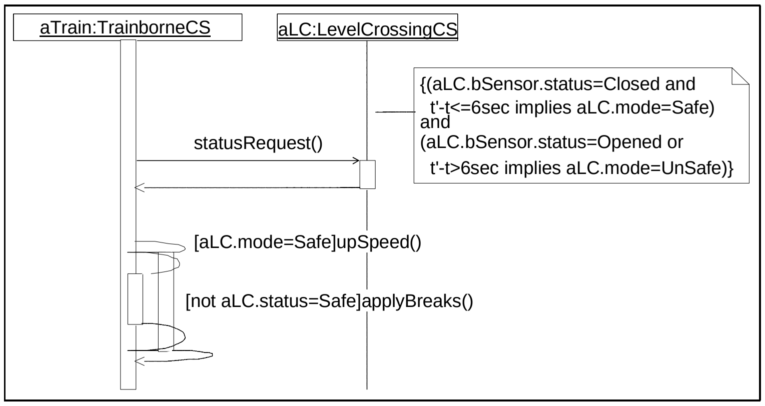
aTrain:TrainborneCS
aLC:LevelCrossingCS
{(aLC.bSensor.status=Closed and t'-t<=6sec implies aLC.mode=Safe) and (aLC.bSensor.status=Opened or t'-t>6sec implies aLC.mode=UnSafe)}
statusRequest()
[aLC.mode=Safe]upSpeed()
[not aLC.status=Safe]applyBreaks()

Fig. 5. Sequence diagram - scenario of train crossing

Safety properties are included in the invariant of the system, in order to ensure their preservation from the abstract specification through the implementation. As the main property of the LCC system is to preserve road traffic and trains to enter the danger zone at the same time, on a high level of abstraction it is sufficient to model the crossing area and its barrier, as well as the train which may cross the level crossing at any time. In the case study, the notion of "train passing the crossing area" is used in connection with the activation of the railway level crossing. Accordingly, the front of a train has to be detected somehow for accomplishing this task. It is the same for the rear end of a train. We assume that the train can be detected in a direct way by introducing abstract vehicle sensors. The detection of the barrier state is also performed by introducing a barrier sensor. Therefore the main safety property on the LCC system class of Fig.2 is expressed by the following OCL constraints :

1. The red light is switched on whenever the barrier is closed and the yellow light is switched on when the barrier is closing. If both the yellow and the red lights are switched off then the barrier is opened :

```
context LCC_System inv:
self.theBarrier.state=Closed implies
self.redLight.state=On and
self.theBarrier.state=Closing implies
self.yellowLight.state=On and
self.yellowLight.state=Off and self.redLight.state=Off
implies self.theBarrier.state=Opened
```

2. While there is still a train at the danger zone the level crossing is in the activated state. The activated state is composed by four substates (WaitingAck, Closing, Closed, Opening) :

```
context LCC_System inv:
not(self.train ->isEmpty()) implies
self.state=Activated and
Set(Activated)=Set(WaitingAck ->Union(Closing)
->Union(Closed) ->Union(Opening))
```

3. When the LCC system is in the activated state and the barrier is opened then the level crossing is in unsafe mode :

```
context LCC_System inv:
self.state=Activated and self.bSensor.state=Opened
implies self.mode=Unsafe
```

4. If the registered state of the barrier is closed whereas triggering the sensor indicates that it is opened, then the level crossing is in unsafe mode. This is the case when the barrier is in the closing state (the lcc remains unsafe until the barrier has been completely closed) :

```
context LCC_System inv:
self.bSensor.state=Opened and self.theBarrier.state=Closed)
implies self.mode=Unsafe
```

The operations of the LCC class are specified with OCL pre and post-conditions. OCL is additionally used in sequence diagrams to complete preconditions and invariants on operations (Fig.3). Although state diagrams are used to derive a first specification of each operation, i.e. describing a state transition, OCL constraints are needed to add supplementary information which can not be retrieved from state diagrams.

Let us consider the closing of the barrier raised by the event timeOut_1. The precondition of the operation closeBarrier has to verify that the yellow light is switched on before sending the closing order to the barrier. It also has to verify than the barrier is not yet closed. The postcondition ensures that the state of the yellow light is off, the state of the red light is on and the state of the barrier is closed. The operation is specified as follows:

```
context LCC_System::closeBarrier
pre:
self.yellowLight.state=On and self.theBarrier.state=Opened
post:
self.yellowLight.state=Off and self.redLight.state=On and
self.theBarrier.state=Closed
```

The precondition of the operation openBarrier which is activated by the trainDetectionRear event verifies that the barrier is closed and the LCC system is in its safe mode. The postcondition ensures the barrier is in the opened state:

```
context LCC_System::openBarrier
pre: self.theBarrier.state=Closed and self.mode=Safe
post: self.theBarrier.state=Opened
```

3.4 Time constraints in UML and OCL

According to [12], as for the latest OCL 2.0 proposal, "apart from OCL messages, there is no other concept in OCL to specify temporal constraints", even if these can be expressed in the following types of UML diagrams :

– State diagrams, where they appear labelling the transitions
– Sequence diagrams, where temporal constraints can be based on whether duration observations or temporal observations.
– Timing diagrams, introduced in [1], being a way of focusing on timing constraints instead of the sequence of actions of the model. Unfortunately this type of diagram

does not add any additional expressiveness to the temporal properties of a model with regard to sequence diagrams.

In all these diagrams the temporal constraints have the same meaning and expressiveness. Moreover, the lack of formal semantics of OCL introduces imprecision, hence the need for translating OCL specifications into another formalisms to help in the validation of the corresponding model. Here an approach using B is presented, but [12, table 1] mentions several other approaches aimed at giving OCL miscellaneous formal semantics.

In the following, we describe first the translation of UML class and state diagrams into B. Then, we propose transformation rules translating OCL constraints into B expressions, and we discuss several approaches to include time constraints from the model into the B specification.

4 From UML models to B specifications

The basis of our approach is the transformation of the UML model into a B formal specification. A major asset of combining UML and B is to give semantics to object models and to enable the use of automated verification and validation tools. We use B in order to specify precisely the structure and the behaviour of the entities composing a system and to prove rigorously that these satisfy the desired structural and behavioural properties. These promise increased reliability of software systems, and the potential of automating the software development process.

4.1 Formalisation of class and state diagrams

The initial B specification (called abstract specification) is obtained from the UML diagrams and used to check inconsistencies. To do so, an abstract machine is associated to each class. Subsequently, the B notation is used to detail each component with the behaviour of class operations and the global invariants. In our approach, the specification is composed by a two levels hierarchy of abstract machines regarding the B inclusion. At the first level, the root abstract machine represents the system itself. This machine specifies the whole structure of the system and it introduces all the associations between classes. Some global properties and constraints formalizing inheritance and aggregation are also added. At the second level of the specification, we introduce an abstract machine representing each class. Each machine is linked to the root machine by the INCLUDES link. Fig.6 shows the structure of the B specification of the railway level crossing system (Fig.2).

This very simple approach makes it easier to define and modify the whole specification. Since the resulting B specification contains less B machines and its hierarchical structure is less complex, there is no problem of knowing where to attach a new class in a hierarchy. Also proof obligations (OPs) concerning internal properties of a class are generated independently from its relationships.

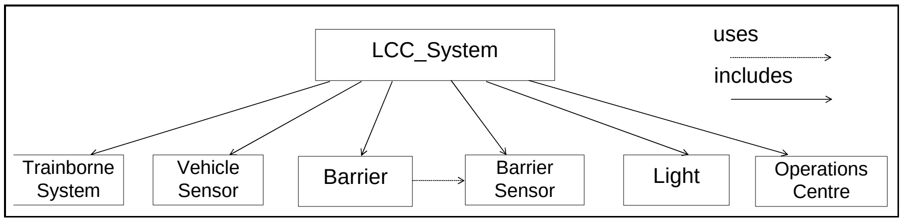

Fig. 6. Structure of the B specification

Classes. Let us consider the class *Barrier* and its first B specification, presented in Fig.7. Since a class includes both static and behavioural properties of a set of objects, it seems natural to model it by one abstract machine. The resulting abstract machine Barrier describes the deferred set BARRIER of all the possible instances of the class *Barrier*. The set of the existing instances is modelled by a variable barrier constrained to be a subset of BARRIER. Each attribute, i.e. *bState*, is represented by a variable, i.e. bState, defined in the INVARIANT clause as a total function between the set barrier and its associated type, i.e. bSTATE. Each operation of the machine has at least one parameter obj representing the object on which the operation is called. It may have a list of typed arguments args which will be completed in the further translation of state diagrams and OCL constraints.

```
MACHINE
  Barrier
SETS
  BARRIER; bSTATE={Opened,Closed}
VARIABLES
  barrier, bState
INVARIANT
  barrier ⊆ BARRIER ∧
  bState ∈ barrier →bState
OPERATIONS

openBarrier(obj) =
     PRE
         obj ∈ barrier ∧
          bState(obj) =Closed
     THEN
         bState(obj) :=Opened
     END;

closeBarrier(obj) =
    PRE
        obj ∈ barrier ∧
         bState(obj) =Opened
    THEN
        bState(obj) :=Closed
    END;
getState(obj , args) =
     PRE
         obj ∈ barrier ∧
          args ∈ Type_args ∧. . .
     THEN
         st :=bState(obj)
    || . . .
     END
```

Fig. 7. Formalisation of classes (machine Barrier)

Associations. Since associations between classes represent couples of instances, they are expressed in B as binary relations between the existing instances of classes. Associations can be expressed more precisely according to the values of the role multiplicities. This is done by constraining the binary relation (↔) as a function (→), partial function

($\nrightarrow$), injection ($\rightarrowtail$) or bijection ($\twoheadrightarrow$) with additional properties on its domain or range. Therefore, the association between the LCC class and the Barrier class *lcc_barrier* is formalized by a variable lcc_barrier. Since the class diagram of Fig.2 establishes that only one barrier is associated to a LCC system, a typing predicate defining it as a function between the set of level crossings (variable lcc) and the set of barriers (variable barrier) is added to the invariant of the LCC_System machine: lcc_barrier $\subseteq$ lcc $\rightarrow$ barrier. Fig. 8 shows the translation of associations.

```
MACHINE
   LCC_System
INCLUDES
   Barrier, BarrierSensor, Yellow.Light,
   Red.light, TrainborneCS
VARIABLES
   barrier, bState
INVARIANT
   lcc_barrier ∈ lcc →barrier ∧
   lcc_sensor ∈ lcc →bSensor ∧
   lcc_train ∈ lcc ⇸train ∧
   redLight ∈ lcc →Red.light ∧
   yellowLight ∈ lcc →Yellow.light ∧...
```

```
OPERATIONS

setLcc_Barrier(obj_i , obj_j) =
    PRE
        obj_i ∈ lcc ∧
         obj_j ∈ barrier
    THEN
        lcc_barrier :=lcc_barrier ∪{ obj_i ↦obj_j }
    END;
rmvLcc_Barrier(obj_i , obj_j) =
      PRE
          obj_i ∈ lcc ∧
           obj_j ∈ barrier ∧
           {(obj_i ↦obj_j)} ∈ lcc_barrier
      THEN
          lcc_barrier   :=lcc_barrier   −{   obj_i
          ↦obj_j }
      END;
```

Fig. 8. Formalisation of classes (machine LCC_system)

Formalisation of state diagrams State diagrams are used to introduce behavioural properties in the B specification. The set of all possible states of a class is formalised by an abstract set which is defined in the respective B machine. An abstract variable is used to reference the current state of the class objects. It is defined as a total function, whose domain is the set of instances and whose range is the set of possible states. Each transition between two states is formalised by a B operation whose name is the name of the incoming event. Whereas the precondition of the operation is deduced from the guard of the transition, the postcondition describes the transition to the new state. Let us consider the state diagram of the LCC_System class (Fig.4). The transition from the showingRlight state to the closingB state activated by the event timeOut_2 is formalized as shown in Fig.9. Note that we have included here some information obtained from the OCL definition of the operation closeBarrier, since this operation is activated by the event timeOut_2 (we describe the translation of OCL below).

When the same event may activate two different transitions depending on a guard condition then both transitions are formalized by the same operation of the B machine. The SELECT close is used to describe each transition, as illustrated on Fig.9 for the formalisation of the event timeOut_3.

```
MACHINE
  LCC_System ...
OPERATIONS

timeOut_1_redLightOn(obj) =
    PRE
        obj ∈ lcc ∧
         state(obj) =ShowingYlight ∧
         bStatus(lcc_sensor(obj)) =Opened ∧
         bState(lcc_barrier(obj)) =Opened ∧
         Red.lState(redLight(obj)) =Off ∧
         Yellow.lState(yellowLight(obj)) =On
    THEN
        state(obj) :=ShowingRlight
     || Yellow.switchOff(yellowLight(obj))
     || Red.switchOn(redLight(obj))
    END;
timeOut_2_closeBarrier(obj) =
       PRE
          obj ∈ lcc ∧
           state(obj) =ShowingRlight ∧
             Red.lState(redLight(obj))   =On
          ∧
            Yellow.lState(yellowLight(obj))
          =Off
       THEN
          state(obj) :=ClosingB
      ||  closeBarrier(lcc_barrier(obj))
       END;

timeOut_3_setMode(obj) =
    PRE
        obj ∈ lcc ∧
         state(obj) =ClosingB ∧
         bState(lcc_barrier(obj)) =Closed ∧
         Red.lState(redLight(obj)) =On ∧
         Yellow.lState(yellowLight(obj)) =Off
    THEN
        SELECT
            bStatus(lcc_sensor(obj)) =Closed
         THEN
            state(obj) :=ClosedB
        ||  mode(obj) :=Safe
         WHEN
            bStatus(lcc_sensor(obj)) =Opened
         THEN
            state(obj) :=Failure
         ELSE
            skip
         END
    END;
```

Fig. 9. Formalisation of state diagrams

Once class and state diagrams are translated and integrated into the initial specification, OCL constraints are used to complete the invariants and operations of the B machines.

4.2 Formalisation of OCL constraints

The semi-formal nature of the OCL definition restricts its appropriate utilization in safety critical applications, leading users to ambiguous interpretations of the UML models. This difficulty is increased by the lack of tools supporting the analysis and proof of the OCL expressions as well of the whole UML models. Some work related to tools for checking UML design is presented in [13] and [14]. The first one proposes an approach for validation of UML models based on simulation. The second one proposes an analyzer for object models using Alloy, which is based on Z. We have carried previous work on formalizing OCL with B based on translation rules between the abstract syntaxes of both languages [5].

We take into account two types of OCL constraints. The first type of constraint specifies an invariant of a class. The second type of constraint specifies a precondition and/or a postcondition of an operation. In the first case, the translation of the OCL constraint consists in a conjunction of a new predicate with the invariant of the related B machine, whereas in the second case, it consists in a completion of an operation of the machine. We illustrate the formalisation of the OCL invariant of the LCC system in Fig.10.

```
MACHINE
  LCC_System
PROPERTIES
  Activated ⊆ STATE ∧
    Activated ={ ShowingYlight, ShwoingRlight,
  ClosingB, OpeningB, ClosedB }
INVARIANT
  ...
  ∀obj (obj ∈ lcc ∧bState(lcc_barrier(obj))
  =Closed
     ⇒ Red.lState(redLight(obj)) =On) ∧

  ∀obj (obj ∈ lcc ∧bState(lcc_barrier(obj)) =Clos-
  ing
     ⇒ Red.lState(yellowLight(obj)) =On)

∀obj (obj ∈ lcc ∧
 Yellow.lState(yellowLight(obj)) =Off ∧
   Red.lState(redLight(obj)) =Off
   ⇒ bState(lcc_barrier(obj)) =Opened) ∧

∀obj (obj ∈ lcc ∧obj ∈ dom(lcc_train)
   ⇒ state(obj) ∈ Activated) ∧

∀obj (obj ∈ lcc ∧state(obj) ∈ Activated ∧
   bStatus(lcc_sensor(obj)) =Opened
   ⇒ mode(obj) =Unsafe) ∧

∀obj (obj ∈ lcc ∧bStatus(lcc_sensor(obj)) =
   Opened ∧bState(lcc_barrier(obj))=Closed
   ⇒ mode(obj) =Unsafe)
```

Fig. 10. Formalisation of OCL invariants

OCL pre and postconditions are used to complete the operations of B machines. In Fig.9, the precondition of the operation timeOut_1 constraints not only the LCC system to be in the yellowLight state (which is generated from the state diagram) but also the red light to be switched on and the barrier to be closed, this is the translation of the OCL predicate :

```
self.yellowLight.state=On and self.theBarrier.state=Opened
```

The postcondition of the operation initially includes only the substitution "state(obj):=ClosingB" setting the new state of the lcc instance (obj). It is completed by the translation in B of the OCL postcondition :

```
self.yellowLight.state=Off and self.redLight.state=On and
self.theBarrier.state=Closed
```

which generates the following parallel substitutions:

```
   closeBarrier (lcc_barrier(obj))
|| Yellow.switchOff (yellowLight(obj))
|| Red.switchOn (redLight(obj))
```

5 Time constraints in the B specification

We present in next paragraphs different, possible approaches to specify time constraints from the UML model and verify them once they are embedded in the obtained B machines.

5.1 The ground B approach

Given the low expressiveness of temporal constraints that can be specified in UML diagrams (see for instance 3), a first possible solution to translate temporal constraints to B and checking them is to use the approach described in [15], which can roughly be described as the use of the own B mechanisms to specify and validate temporal constraints. This is achieved by specifying a clock abstract machine and defining variables holding the times we are interested in. With the help of these variables, we can then specify simple temporal constraints as well as the properties of the different operations. Let us illustrate this by translating the temporal constraints and adding them to the B model corresponding to diagrams 4 and 3.

For instance, the left part of figure 11 represents the different means the clock variable can be used to specify that an operation takes some time, or that an operation can be triggered only in a certain interval of time. Let us detail those means:

– **closeBarrier** specifies that an amount of time of at most **ClosingDelay** time units is required for the barrier to close.
– **timeOut_2_closeBarrier** shows that the closing of the barrier takes place as at most 12 time units. Actually, the models does not precise whether this delay must be *exactly* or *at most* 12 time units. We take here a safe bet, as it is still work-in-progress.
– **timeOut_3_setMode** shows here a dynamical behaviour depending on the past events and the time they took: if the barrier successfully closed in less that 6 time units then the system is in a safe state. Remember that we specified for the closing of the barrier that it took at most **ClosingDelay** time units. Hence at the verification step, we will have to ensure that this delay is compatible with the 6 time units of the model.

```
closeBarrier(obj) =
   PRE
      obj ∈ barrier ∧
       bState(obj) =Opened
   THEN
      bState(obj) :=Closed
   ||  ANY
          newtime
       WHERE
          newtime ∈ ℕ∧
           newtime ≥time ∧
           newtime ≤time + ClosingDelay
       THEN
          setTime(newtime)
       END
   END;

timeOut_2_closeBarrier(obj) =
   PRE
      obj:lcc ∧
       state(obj) =ShowingRlight ∧
       Red.lState(redLight(obj)) =On ∧
       Yellow.lState(yellowLight(obj)) =Off ∧
       time −trainDetectionEntry_Time ≤12
   THEN
      state(obj) :=ClosingB
   ||  closeBarrier(lcc_barrier(obj))
   ||  closeBarrierTime :=time
   END;
```

```
timeOut_3_setMode(obj) =
   PRE
      obj:lcc ∧
       state(obj) =ClosingB ∧
       bState(lcc_barrier(obj)) =Closed ∧
       Red.lState(redLight(obj)) =On ∧
       Yellow.lState(yellowLight(obj)) =Off
   THEN
      SELECT
          bStatus(lcc_sensor(obj)) =Closed ∧
           time −closeBarrierTime ≤6
       THEN
          state(obj) :=ClosedB
      ||  mode(obj) :=Safe
       WHEN
          bStatus(lcc_sensor(obj)) =Opened ∧
           time −closeBarrierTime >6
       THEN
          state(obj) :=Failure
       ELSE
          skip
       END
   END;
```

Fig. 11. Operations including time constraints

As it can be seen, the addition of temporal constraints is straightforward: depending on whether they are dynamic or not, they will appear in select-like substitutions or in preconditions. For instance, the static requirement of figure 5 will appear in the precondition of the operation that need it, namely **statusRequest**.

There is one subtlety though, because this way of specifying timed constraints does not apply immediately to concurrent (by the way of the substitution) operations, because the time variable of the clock would be modified on both the sides of it, which is forbidden by the B method. The simple solution here is to use a modified substitution whose semantics is to retain the two values the modified variable can take, and presented in [16]. Hence, for instance, in operation **timeOut_1_redLightOn**, the time would be advanced by both the operations **Yellow.switchOff** and **Red.switchOn**, in a non-deterministic way. Thus, at the verification step, one has to ensure the two possible new values of the time verify the specified time constraints.

As we can see, the temporal constraints from the UML model can easily be expressed in terms of B constraints and substitutions. This method, because it is compatible with the classical B method, simplifies the addition of temporal requirements to the B semantics of OCL formulas. However, the advantage is the drawback here, as any strong update of the OCL specifications with better temporal expressiveness might lead to insufficiencies on the B side. But so far, the method is quickly adaptable to an UML model of a similar complexity as the level crossing model.

5.2 With B extended with a temporal logic

In [17] we describe an approach to extend B with the duration calculus, an interval temporal logic. Without going too much into detail, this approach allows to label operations with temporal formulas to be checked against them. The formula representing the temporal properties of the operation is generated with the help of the operation (as expected) and a formula known to be valid after the operation is evaluated, i.e. in general a postcondition. Then we check we can deduce the temporal constraint of the operation from its temporal properties.

For our UML model, the approach that seems the most natural is to focus on the different changes of the state of the system w.r.t. their durations, hence showing a better adequation of this approach with the timing diagrams introduced in [1]. Nevertheless, we can sketch an approach dealing with the static and dynamic constraints that can be embedded into the model :

– The dynamic constraints (as in **timeOut_3_setMode**) still have to use a clock to keep track of certain points of time.
– The static constraints then facilitate defining the operations w.r.t. durations of states: for instance, instead of holding the 12 time units constraints into the **timeOut_2_closeBarrier** operation, we would rather specify an operation containing all the calls made from the beginning of the arrival of the train, to the time the red light has been set. See [17] for more technical details.

Several remarks related both to this approach and the UML model have to be made, though:

– The modelling process has to be thought of again, in order to include the timing diagram, i.e. a perspective of the model focused on the durations of its different states
– Some of the ambiguities of the sequence diagram have to be relieved: for instance the “dead delays” between the return of an operation result and the following operation call: are they undetermined delays, or needn’t they be taken into account ? This question is for instance answered in section 5.1, where these “dead delays” do not appear in the final B machines. But other semantical choices are possible.

5.3 With event B and timed automata

Yet another approach allowing to deal with temporal properties of B models is described in [18]: the B model is associated to a timed automata describing the evolution of the states of the model. This approach would integrate well with the state diagram modelling used in UML tools, as these diagrams can be annotated with time constraints. Moreover, the approach of [18] also focuses on the possible refinement of the model in conjunction with its automata, hence giving the possibility to give a refinement semantics (that of event B) to UML and OCL. As this paper doesn’t deal with the semantic changes required to have UML models translated to event B machines, the interested reader is encouraged to read [18].

5.4 Discussion about these approaches

In the three approaches presented here, several remarks about the complexity of timed properties that can be expressed have to be addressed :

– Because it is the simplest one, the approach of section 5.1 does not easily allow the expression of very complex timed properties, like liveness and fairness. However its implementation is the easiest to achieve, because it does not intervene much on the translation from UML models to B machines, and is enough to state the real-time properties of the level-crossing model presented in this paper
– The approach of section 5.2 fits also well with the UML model. It would even be closer to it if a timing diagram (see discussion in sec. 3.4) was present. Apart from the details of the translation to this extended B, the underlying logic (Duration Calculus) allows to express very complex time properties (including of course liveness and fairness properties). Thus with this approach any refinement of the temporal properties and constraints of the UML model would be mirrored adequately in the B machines
– While it is not described into detail, the approach of sec. 5.3 is also sufficient to express the properties of the model, and also has the nice property of allowing easy refinement : indeed, the UML model maps almost one-on-one with the timed automata, so any refinement of the former is easily reflected in the latter. As for the temporal properties to be checked, they depend on the logic used to state them and verify them with respect to the timed automata. In general, logics like CTL are used, and are sufficient to express liveness and fairness properties.

6 Verification of the whole model

In order to automate the formalization process we have implemented a prototype tool performing the derivation of UML/OCL models into B specifications. It is part of the Brillant [1] project, which aims at providing automated tool support for the B method. The translation is composed of three main steps (Fig.12):

1. From UML to XML. At the first step, a UML model is encoded into a XML schema. At the design level, we have chosen the Poseidon [2] modelling tool in order to draw a UML model and generate its associated XMI model (model.xmi). A transformation schema (xmi2uml) is written using the XSLT language allowing the XMI file to be translated into a XML file (model.xml).
2. From XML to UML-parsed models. The resulting XML file contains the information about the UML model elements. These model elements have to be parsed into OCaml types according to the UML abstract syntax tree definition (uml.ml). This is done by the IOXML processor. Therefore, the resulting file (model.ml) can be used to generate the B specification.
3. From UML models to B specifications. The uml2b module implements translation rules from UML class and state diagrams, as well as OCL constraints, into B specifications. The translation rules are implemented in OCaml as mappings of the UML abstract syntax (uml.ml) into the B abstract syntax (b.ml).

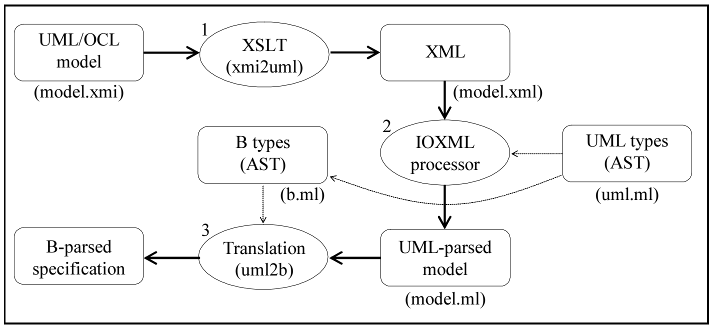

Fig. 12. Automated translation of UML models into B specifications

Once the whole B formal specification is generated from the UML/OCL model it has first to be type-checked and then verified through a proof process. The result of performing type check on the B specification is the detection of syntax and type errors on UML/OCL models. The theorem prover of Brillant is then used to automatically generate and proof the proof obligations (OPs). The OPs guarantee the conformance of the operations of a B machine to its invariant. Each operation raises proof obligations related to its precondition and substitution parts. The non proven OPs are used to detect

[1] https://gna.org/projects/brillant/
[2] http://www.gentleware.com/

inconsistencies between invariant and preconditions as well as incompleteness of a postcondition. Fig. 13 shows an extract of the proof obligation for the timeOut_3_setMode operation.

$$
\begin{array}{l}
(\mathrm{LCC} \in \mathbb{P}_1 \mathbb{INT} \\
\quad \wedge\ \mathrm{STATE} \in \mathbb{P}_1 \mathbb{INT} \\
\quad \wedge\ \mathrm{STATE} = \{\mathrm{Deactivated, ShowingYlight, ShowingRlight,} \\
\qquad \mathrm{ClosingB, OpeningB, ClosedB, Failure}\} \\
\quad \wedge\ \mathrm{MODE} \in \mathbb{P}_1 \mathbb{INT} \\
\quad \wedge\ \mathrm{MODE} = \{\mathrm{Safe, Unsafe}\} \\
\quad \wedge\ \mathrm{Activated} \subseteq \mathrm{STATE} \\
\quad \wedge\ \mathrm{Activated} = \{\mathrm{ShowingYlight,\ ShowingRlight,\ ClosingB,\ OpeningB,\ ClosedB}\} \\
\quad \wedge\ \mathrm{LCC\_System\_INVARIANT} \\
\quad \wedge\ \mathrm{BARRIER} \in \mathbb{P}_1 \mathbb{INT} \\
\quad \wedge\ \mathrm{barrier} \subseteq \mathrm{BARRIER} \\
\quad \wedge\ \mathrm{bState} \in \mathrm{barrier} \rightarrow \mathrm{bSTATE} \\
\quad \wedge\ \ldots \text{(additional Sets and invariants from included machines)} \\
\Rightarrow \\
\\
\mathrm{bStatus(lcc\_sensor(obj))} = \mathrm{Closed} \Rightarrow \\
\qquad [\ \mathrm{state(obj)} := \mathrm{ClosedB} \parallel \mathrm{mode(obj)} := \mathrm{Safe}\ ]\mathrm{LCC\_System\_INVARIANT} \\
\ \wedge\ \mathrm{bStatus(lcc\_sensor(obj))} = \mathrm{Opened} \Rightarrow \\
\qquad [\ \mathrm{state(obj)} := \mathrm{Failure}\ ]\mathrm{LCC\_System\_INVARIANT} \\
\ \wedge\ \neg(\mathrm{bStatus(lcc\_sensor(obj))} = \mathrm{Closed} \vee \mathrm{bStatus(lcc\_sensor(obj))} = \mathrm{Opened}\ ) \Rightarrow \\
\qquad [\ \mathrm{skip}]\mathrm{LCC\_System\_INVARIANT}
\end{array}
$$

Fig. 13. Proof obligation for the **timeout_3_setMode** operation

If a proof obligation cannot be proven using the theorem prover then the developer is required to review the related OCL invariant or operation and to make the necessary modifications. The approach presented here is a one-way approach as we can translate from UML/OCL to B but not the other way round. When the type checker or the prover finds an error in the specification, the user must be able to understand the B specification and then has to search in the UML/OCL model where is the error. Let us note that it is quite simple for the developer to find the UML element associated to a B expression. However, in order to facilitate this task, it could be possible to create and maintain concrete links between UML/OCL and B specifications elements during all the development process, because the names are roughly the same and each OCL expression is translated into a simple B expression.

7 Conclusion

This paper is a part of a major project aiming at combining formal and semi-formal methods, particularly in the context of railway control systems. The work presented here is an effort to put together the complementary strengths of the UML notation and a time extended version of the B formal method. The resulting approach promises increased reliability of software systems and the potential of automating the software development process.

The paper puts a particular emphasis on the formalisation of the UML class, state and sequence diagrams as well as OCL constraints. So far we have implemented a prototype tool automating the translation of class and state diagrams with OCL annotations into B specifications. The translation of sequence diagrams is being implemented.

Several approaches to deal with real-time constraints of an UML model have been presented here (section 5), although the transformation mechanics do not include them (yet). We saw that all these approaches allow to give more or less expressiveness to the B specification that is to be checked. Nonetheless, the lack of semantics for real-time constraints in UML and OCL leaves us with indecision, hence preventing us from choosing the semantics that would be adequate for a future version of UML/OCL including real-time semantics.

References

1. Object Management Group OMG. Unified modeling language superstructure, version 2.0. final adopted specification, omg document ptc/2003-08-02, August 2003.
2. EN 50128 CENELEC. Railway applications - software for railway control and protection systems, 1997.
3. J.-R. Abrial. *The B-Book : Assigning Programs to Meanings*. Cambridge University Press, 1996.
4. R. Marcano and N. Levy. Using B formal specifications for analysis and verification of UML/OCL models. In *Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language*, Dresden, Germany, September 2002.
5. R. Marcano and N. Levy. Transformation rules of ocl constraints into b formal expressions. In *CSDUML'2002, Workshop on critical systems development with UML. 5th International Conference on the Unified Modeling Language*, Dresden, Germany, September 2002.
6. Eric Meyer and Jeannine Souquières. Systematic approach to transform OMT diagrams to a B specification. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 – Formal Methods*, number 1709 in Lecture Notes in Computer Science (Springer-Verlag), pages 875–895. Springer Verlag, September 1999.
7. R. Laleau and A. Mammar. Overview of a method and its support tool for generating B specifications from UML notations. In *Proceedings of ASE'2000 : 15th International Conference on Automated Software Engineering*, Grenoble (France), September 2000.
8. Object Management Group OMG. Object constraint language, version 2.0. final adopted specification, omg document ptc/2003-10-14, October 2003.
9. N. Levy, R. Marcano, and J. Souquières. From requirements to formal specification using uml and b. In *2nd International Conference in Computer Systems and Technologies CompSysTech2002*, Sofia, Bulgaria, June 2002.
10. L. Jansen and E. Schnieder. Traffic control system case study: Problem description and a note on domain-based software specification. technical report, 2000.
11. Betriebliches Lastenheft für FunkFahrBetrieb. Stand 1.10.1996.
12. Stephen Flake. Temporal ocl extensions for specification of real-time constraints. In *Workshop Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS'03)*, San Francisco, CA, USA, October 2003. UML 2003.
13. M. Richters and M. Gogolla. Validating UML models and OCL constraints. In *Proceedings UML 2000*, 2000.
14. Jackson Daniel, Schechter Ian, and Ilya Shlyakhter. Alcoa: the Alloy costraint analyzer. In *International Conference on Software Engineering*, Limerick, Ireland, June 2000.

15. Helen Treharne and Steve Schneider. Capturing timing requirements formally in AMN. Technical Report CSD-TR-99-06, Royal Holloway, Department of computer science, Egham, Surrey TW20 0EX, England, June 1999.
16. Jean-Paul Bodeveix, Mamoun Filali, and César Munoz. A formalization of the B method in Coq and PVS. In *FM'99 – B Users Group Meeting – Applying B in an industrial context : Tools, Lessons and Techniques* [19], pages 32–48.
17. Samuel Colin, Georges Mariano, and Vincent Poirriez. Duration calculus: A real-time semantic for B. In *First International Colloquium on Theoretical Aspects of Computing*. UNU-IIST, september 2004. Guiyang, China.
18. A. Hammad, Jacques Julliand, H. Mountassir, and D. Okalas Ossami. Expression en B et raffinement des sytèmes réactifs temps réel. In *AFADL'2003* [20], pages 211–226.
19. *FM'99 – B Users Group Meeting – Applying B in an industrial context : Tools, Lessons and Techniques*. Springer-Verlag, 1999.
20. IRISA. *Approches Formelles dans l'Assistance au Développement de Logiciels*, IRISA Rennes – France, January 2003. IRISA.