

Contributions for Modelling UML State-Charts in B

Hung LEDANG and Jeanine SOUQUIÈRES

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

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

In this paper, we address the problem of modelling UML state-charts in B, which has not been, so far, completely treated. We distinguish between event-related and activity-related parts of UML state-charts. We propose deriving the B specification of the event-related part independently with the activity-related part. For this purpose, a new approach for modelling events is proposed; the communication among state-charts is also considered.

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

1 Introduction

The Unified Modelling Language (UML)[22] has become a de-facto standard notation for describing analysis and design models of object-oriented software systems. The graphical description of models is easily accessible. Developers and their customers intuitively grasp the general structure of a model and thus have a good basis for discussing system requirements and their possible implementation. However, since the UML concepts have English-based informal semantics, it is difficult even impossible to design tools for verifying or analysing formally UML specifications. This point is considered as a serious drawback of UML-based techniques.

To remedy such a drawback, one approach is to develop UML as a precise modelling language. The pUML¹ (precise UML) group has been created to achieve this goal. However the main challenge [5] of the pUML is to define a new formal notation that has been up to now an open issue. Furthermore, the support tool for such a new formalism is perhaps another challenge.

In waiting for a precise version of UML and its support tool, the necessity to detect semantic defects inside UML specifications should be solved in a pragmatic approach (cf. [26, 4]): formalising UML specifications by existing formal languages and then

¹ <http://www.cs.york.ac.uk/puml/>

analysing UML specifications via the derived formal specifications. In this perspective, using the B language [1] to model UML specifications has been considered as a promising approach [13, 25, 18, 20]. By formalising UML specifications in B, one can use B powerful support tools like AtelierB [27], B-Toolkit [3] to analyse and detect semantic defects inside UML specifications (cf. [14]). On the other hand, we can also use UML specifications as a tool to develop B specifications which can be refined automatically to the executable code [2, 8].

This paper addresses the modelling in B of UML state-charts. Such a modelling has been previously done by Meyer, Nguyen and Lano [18, 19, 20, 10, 23]. However, their rules for mapping UML state-chart-related concepts into B have got followed shortcomings:

1. the modelling of the activity-related part (cf. [10]) in state-charts did not consider the interference of event-related part;
2. the modelling of events could not work if events trigger transitions with several sequential actions. This is the case in which a state entry/exit action must follow/precede actions of incoming/outgoing transitions of the state;
3. There has been no mechanised derivation procedure for UML state-charts due to ambiguities of modelling rules;
4. deferred events and the communication of UML state-charts have not been considered.

We propose to distinguish between activity-related and event-related parts of state-charts. The event-related part in state-charts relates to events. These elements comprise events, state, transition and actions. Remainders in state-charts constitute the activity-related part. Since activities do not affect states, it is natural to model the event-related part independently with respect to the activity-related part. However, the modelling of the activity-related part should take into account the interference of the event-related part.

This paper addresses only the modelling in B of the event-related part in UML state-charts. The interference of the event-related part to the activity-related part need a further investigation and is beyond the scope of the current paper. We propose a new approach for modelling events in which the different semantics of non deferred events and deferred events should be taken into account. Each non deferred events is modelled in two stages:

1. creating B abstract operations for non deferred events in which the event effect on the related data is directly specified;
2. implementing or refining the B operation in the first step by calling B operations for the triggered transition and actions.

In our opinion, this two-stages approach allows us to overcome the second shortcoming above. In addition, modelling non deferred events in two stages gives also possibilities to verify the conformance of the envisaged effects of an event with respect to its triggered actions. Deferred events are modelled in a similar manner, however, the queueing and dequeuing of those events should be taken into account. The approach for modelling deferred events can be extended to model the asynchronous communication

among state-charts. Dealing with the synchronous communication among state-charts is another contribution in this paper. The idea is to avoid calling the B operation for *message receipt event* in the B operations of the event that send the message because both events are modelled in the same abstract machine. For this purpose, the content of the B operation for the message receipt event is inserted in the B operation of the event that send the message. Hence, our proposals together with previous proposals by Meyer, Nguyen and Lano for modelling states, transitions and actions give rise to a complete framework for deriving B specifications from the event-related part of UML state-charts.

In Section 2, basic concepts of UML state-charts and a UML specification example are presented. The example will be used through the whole presentation. In Section 3, previous work for modelling UML state-charts in B is presented. Section 4 presents a new approach to model non deferred events. The modelling of deferred events is presented in Section 5. In Section 6 we go on to present the modelling of the communication among UML state-charts. Section 7 presents an automatic integration procedure to derive B specifications from UML class diagrams and UML state-charts. Finally, in Section 8, concluding remarks complete our presentation.

2 UML state-charts

2.1 Basic concepts

State-chart diagrams are UML diagrams for modelling dynamic aspects of systems. State-charts were invented by Harel [7], the semantics and the notation of UML state-charts are substantially those of Harel state-charts with adaptations to the object-oriented context.

UML state-charts focus on the event-ordered behaviour of an object, a feature which is specially useful in modelling reactive systems. A state-chart shows the event triggered flow of control due to transitions which lead from state to state, i.e it describes the possible sequences of states and actions through which a model element can go during its lifetime as a result of reacting to discrete events. A state reflects a situation in the life of an object during which this object satisfies some condition, performs some action, or waits for some event.

Transitions are viewed in UML as relationships between two states indicating that an object in the first state will enter the second state and performs specific actions when a specified event occurs provided that certain conditions are satisfied.

The semantics of event processing in UML state-chart is based on the *run to completion* (rtc) assumption: events are processed one at a time and when the machine is in a stable configuration, i.e. a new event is processed only when all the consequences of the previous event have been exhausted. Therefore, an event is never processed when the state-chart is in some intermediate, unstable situation.

Events may be specified by a state as being possibly deferred. They are actually deferred if, when occurring, they do not trigger any transition. This will last until a state is reached where events are no more deferred or where events trigger a transition.

2.2 Example: a lift control system

This section presents the UML specification of a system that controls a set of lifts and buttons. The class diagram is shown in Figure 1. For a lift, we consider information about the current movement direction ($dir \in \text{DIRECTION} \triangleq \{up, down\}$), the next floor ($curDestFloor \in \text{FLOOR} \triangleq \text{ground} \dots \text{top}$) where the lift will arrive and the lift door status ($doorStatus \in \text{DOORSTATUS} \triangleq \{open, closed\}$). Each button is attached to a floor. Given one lift and one floor, there are two buttons: the *indication button* is in the lift and the *call button* is at the given floor.

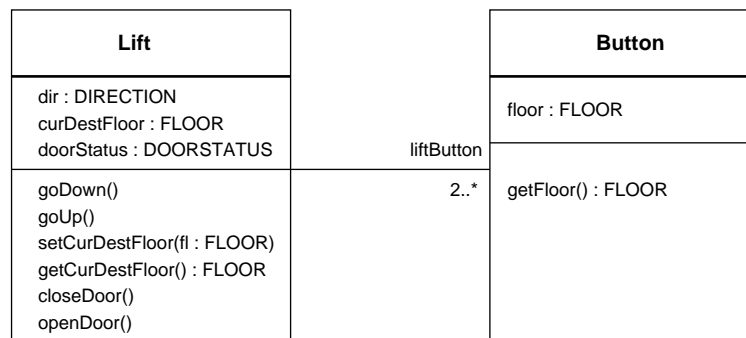


Fig. 1. Class diagram for the lift control system

Figure 2 shows the state-charts of classes Button and Lift. Each button has two states *on* and *off* corresponding to the state of the button light. When the button is pressed, it is in the state *on*; when the floor of the button is visited, it returns to the state *off*. When a button in the state *off* is pressed, an event call is sent asynchronously to the lift associated to the button. The parameter of call is the floor of the button. Each lift has three states *ready*, *visit* and *movement*. It is only in the state *ready* that the lift serves an eventual call by changing into one of the states *visit* or *movement* depending on the calling floor. Because a lift should process all eventual events call, the event call is deferrable in states *visit* and *movement*. The door of a lift in the state *visit* should be open, meaning that `openDoor()` is the entry action of *visit*. The lift in the state *visit* should change to *ready* before process another event call. At the exit of the state *visit*, the lift door should be closed, hence `closeDoor()` is the exit action of *visit*. In the state *movement*, if the lift arrives at the destination floor (event *arrive*), the event release is sent synchronously for lighting off two buttons attached to the visited floor. Note also that a lift will not change its destination floor during the movement.

3 Modelling UML state-charts in B : state-of-the-art

This section recalls essential points in the work of Meyer, Nguyen and Lano for modelling UML state-charts in B. Those works are based on the UML state-chart concepts

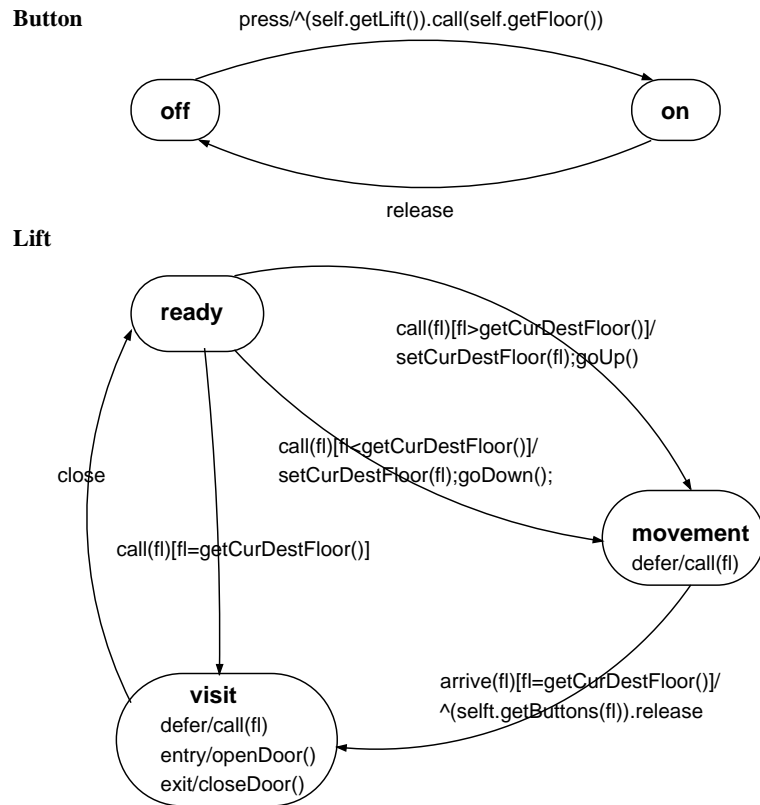


Fig. 2. State-charts for the lift control system

inherited from OMT state-chart [21]: state, sequential sub state, concurrent sub state, transition, action and non deferred event. Each of such elements is modelled by a derivation scheme. The B derivation of UML state-charts is integrated in the B derivation of class diagrams.

3.1 Derivation schemes to B for UML state-charts

Derivation 1 (States) There has been derivation schemes for states, subs states, however, we recall here only, as an example, the derivation scheme for states.

For each state-chart attached to a class *Class*, we create a B enumerated set *STATE* which gathers all the states of the diagram. The state of an object is recorded by a B variable *state* defined as a function from the B variable *class*, which models the set of effective instances of *Class*, to *STATE*. Thus, the state of an object *oo* is defined as *state(oo)*. Transitions between states correspond to the modification of *state*. The initial state is set up in the instance creation operations as it is done for class at-

tributes. Figure 3 presents the formalisation of the states assigned to the class Lift (cf. Section 2.2).

```

MACHINE Lift
/* the abstract machine Types declares OBJECTS and models attributes types */
SEES Types
CONSTANTS
  LIFT
PROPERTIES
  LIFT  $\subseteq$  OBJECTS
SETS
  LIFT_STATE = {ready, visit, movement}
VARIABLES
  lift, dir, curDestFloor, doorStatus, liftState
INVARIANT
  lift  $\subseteq$  LIFT  $\wedge$ 
  dir  $\in$  lift  $\rightarrow$  DIRECTION  $\wedge$ 
  curDestFloor  $\in$  lift  $\rightarrow$  FLOOR
  doorStatus  $\in$  lift  $\rightarrow$  DOOR_STATUS
  liftState  $\in$  lift  $\rightarrow$  LIFT_STATE
...
END

```

Fig. 3. B formalisation for states

Let us recall that the special abstract machine *Types* is used to model attribute types of classes. *Types* is seen (link **SEES**) by abstract machines derived from classes in which we model the attributes. An interesting point of Meyer work is to model a class instance space by a constant. For example, the constant *LIFT* models the instance space of Lift. *LIFT* is considered as a subset of the set *OBJECTS*, which models instance space of all classes. *OBJECTS* is also declared in *Types*. One can therefore model the multiple inheritance between class that could not be treated in the work of Nguyen and Lano in which the class instance space is modelled as a B deferred set.

Derivation 2 (Transitions) Each transition is formalised by a B operation which models the change of the state. Figure 4 shows the B operation *transVisitReady(...)* modelling the transition from the state *visit* to *ready* in the state-chart Lift.

Derivation 3 (Actions) Each action corresponds to a class operation. Thus, each action is naturally modelled as a B operation (cf. *closeDoor(...)* in Figure 4).

Derivation 4 (Events) Each event is also formalised by a B operation. This operation is parameterised by target objects and eventual parameters of the event. Parameters are typed by a predicate in the precondition clause. The body of the B operation for an

```

MACHINE Lift
...
OPERATIONS
  transVisitReady(ll) =
    pre
      ll ∈ lift ∧ liftState(ll) = visit
    then
      liftState(ll) := ready
    end;
  closeDoor(ll) =
    pre
      ll ∈ lift
    then
      doorStatus(ll) := closed
    end;
...
END

```

Fig. 4. B formalisation for transitions and actions

event constitutes invocations to B operations modelling the triggered transitions and their associated actions. Figure 5 shows the B operation *close(...)* for the event close.

```

...
OPERATIONS
  close(ll) =
    pre
      ll ∈ lift
    then
      select liftState(ll) = visit then
        transVisitReady(ll) ||
        closeDoor(ll)
      else skip end
    end;
...

```

Fig. 5. Current B formalisation for events

3.2 Observations

In our opinion, derivation schemes for states, transitions and actions have been well defined. Following observations are about the modelling of events. First of all, one cannot apply the event derivation scheme above for deferred events since the semantics

of deferred events has not been considered. For non deferred events, at the first glance, the derivation scheme seems to be evident. However, at a close inspection, two problems can be raised:

1. **how to distribute B operations into abstract machines?** Considering the example in Figure 5. Because *close(...)*, *transVisitReady(...)* and *closeDoor(...)* are in abstract machines, the abstract machine for *close(...)* should have a link **INCLUDES** to the abstract machine *Lift* for *transVisitReady(...)* and *closeDoor(...)* (cf. Figure 4). However, calling two B operations from the same included abstract machine is not allowed according to [1, 27]. The solution by Meyer to declare *transVisitReady(...)* in the clause **DEFINITION** cannot deal with transitions from *ready* to *movement* in which several actions are attached to an individual transition;
2. **what about the action sequence?** The action sequences in transitions from *ready* to *movement* cannot be modelled in the abstract machine context since the sequential substitution “;” is not allowed.

The two above problems were justified by the fact that there has not been, so far, an appropriate solution to automatically map UML state-charts into B (cf. Self-evaluation of Meyer in section 6.2.3 of the chapter 6 in [18]). The prototype **Argo/UML+B** by Meyer works actually only with UML class diagrams. In [9], Laleau and Mammar have presented a support tool for generating B specifications from UML diagrams of data intensive applications. Although they considered UML state-charts, nothing new is added regarding the work of Nguyen. In the tool **U2B** by C. Snook and M. Butler [24], which implements a subset of Meyer derivation schemes for UML class diagrams, the modelling in B of UML state-chart could not deal with the action sequence. Furthermore, no mechanised way to organise B operations into abstract machines has been proposed (cf. [6]).

4 New approach for modelling non deferred events in B

We distinguish between non deferred events and deferred events due to their different semantics. A non deferred event must be handled immediately just after its occurrence or it should be lost. A deferred event can occur at a state and can be handled afterward at another state. This section presents a way to model non deferred events. The modelling of deferred events will be discussed in the next section.

4.1 The two-stages approach

As said earlier, we model each non deferred event by a B operation. However the B operation for each non deferred event is implemented or refined afterward. In other words, there are two B specifications for each non deferred event: the first is a B abstract operation; this abstract operation is then implemented or refined by the second. The expected effect of a non deferred event on related data is specified directly in the corresponding B abstract operation. It is only in the B implementation or refinement operation that we make explicitly invocations to B operations of the triggered transitions and actions.

Building the abstract content and implementation or refinement content for B operations of non deferred events constitutes two stages in modelling non deferred events in B².

4.2 Creating B abstract operations for non deferred events

```

MACHINE System
SEES Types
...
VARIABLES
  lift, liftState, doorStatus, ..., button, ...
...
OPERATIONS
  close(l) =
  pre
    l ∈ lift
  then
    select liftState(l) = visit then
      doorStatus(l) := closed ||
      liftState(l) := ready
    else skip end
  end;
...
END

```

Fig. 6. New B formalisation for non deferred events

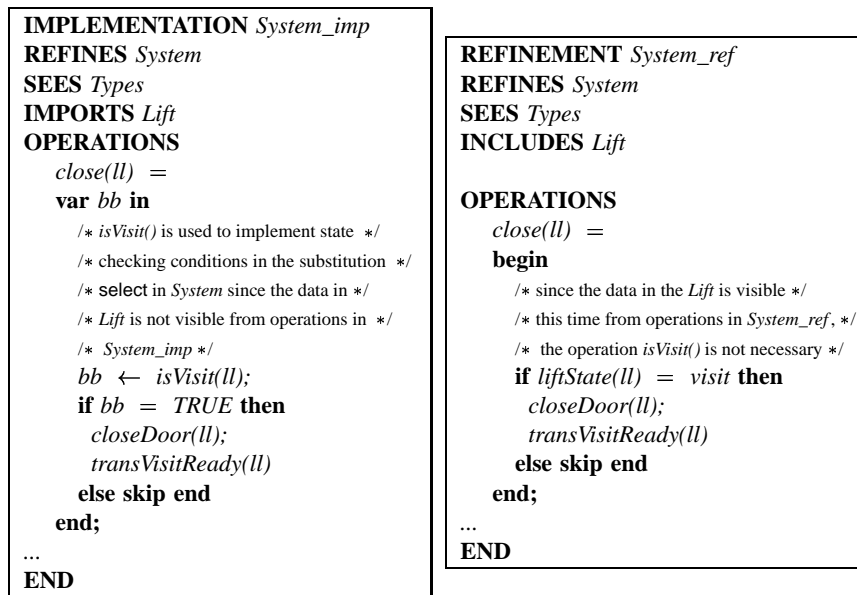
In order to specify directly the effect of an event on its concerned data, we propose to group an event and related data in the same abstract machine. Thus, the problem of modelling events becomes one of how B substitutions can be used to express the pre-/post of the event. This is similar to model actions or basic class operations that are local to one class. Figure 6 shows an abstract machine *System*, in which the B operation *close(...)* models the event *close*. In the data declaration section (clauses **SETS, VARIABLES...**) of *System* we notice the presence of the data derived from the class *Lift*. In addition, *System* has link **SEES** to *Types* because *System* needs references to B types defined in *Types* for modelling attribute types of *Lift*.

We propose to create an abstract machine (*System*) for all events; the reason is that an event can affect data from different classes (cf. the event *arrive*). Data of *System* are derived from the whole class diagram. However, *System* does not contain B operations for transitions and actions since those operations are used to implement or refine abstract operations of events as described in the following sections.

² Similar ideas have been used in our previous work for modelling use cases and class operations [12, 16, 15].

4.3 Solution 1 : implementing B operations of non deferred events

The first solution to deal with the relationship between non deferred events and their triggered transitions and actions is to use the B implementation construct and the B importation primitive. The abstract machine (*System*) for events is implemented by importing abstract machines for transitions and actions, so that B operations for events can be implemented by calling B operations of transitions and actions. In Figure 7(a), the B operation *close(...)* of *System* is implemented by calling operations *closeDoor(...)* and *transVisitReady(...)* of *Lift*.



(a) Solution 1 : using the B implementation construct

(b) Solution 2 : using the B refinement construct

Fig. 7. B formalisation for the relationship between events, transitions and actions

As you can notice, both *System* and *Lift* contain some data with the same name and properties since those data are all derived from the same class (*Lift*). This identity acts as the implicit gluing invariant between *System* and *Lift* in *System_imp*. In addition, the state checking expression *liftState*(*ll*) = *visit* has been implemented by an auxiliary operation *isVisit(...)*. The reason is that data in *Lift* is not visible in the operations of *System_imp* that imports *Lift*. The situation is similar for expressions that model eventual guard conditions of transitions. Those eventual auxiliary operations for the state checking and for guard conditions can be defined in abstract machines for classes or associations.

Remark 1

1. Since the B implementation construct allows sequence substitutions, the modelling of action sequence is enabled. Furthermore, since there is no restriction regarding operations in imported abstract machines to be called in the implemented operations, we are therefore free to arrange B operations of transitions and actions in different abstract machines or in one unique abstract machine.
2. The parallel substitution “||” is not allowed inside the B implementation construct so we can not model explicitly the parallelism of actions. However, since those actions affect different data (cf. page 434 in [22]), we can therefore simulate the parallelism of actions by the sequence without losing the total effect.

4.4 Solution 2 : refining B operations of non deferred events

The combination of the B refinement construct and the B inclusion primitive provides another way to deal with the relationship between non deferred events and their triggered transitions and actions (cf. Figure 7(b)). Both solutions for modelling the relationship between events and triggered transitions and actions are equivalent, however the use of the B refinement construct and the B inclusion primitive allows one to avoid auxiliary operations such as *isVisit(...)* since the data in included abstract machines are visible in the operations of including refinement components. For this reason, the combination of the B refinement construct and the B inclusion primitive is preferred and will be referenced afterward.

5 Modelling deferred events in B

In some modelling situations, we may want to recognise some events but postpone a response to them until later. UML allows one to specify such situations by using deferred events. We therefore distinguish between the adding of deferred events in an internal buffer in some states and the handling them in another states. Deferred events are taken off the buffers as soon as the object enters a state that does not defer those events and the events become active.

5.1 Modelling deferred event internal buffers

The internal buffer is composed of items. Each item is a record of fields for receiver objects and arguments of the event. In the context of the state-chart Lift, the event call is deferred in the states *movement* and *visit*. The buffer *call.Buffer* for call contains records of two fields: (i) the reference to an object *ll* of the class Lift; and (ii) the argument *fl* of the type FLOOR. *call.Buffer* can be modelled as a B variable *call_Buffer* defined as followed:

$$\boxed{call_Buffer \subseteq lift \times FLOOR}$$

Remark 2

If such a pair $\{ll, fl\}$ can appear several times in $call_Buffer$, $call_Buffer$ is defined as followed:

$$call_Buffer \in (lift \times FLOOR) \rightarrow NAT$$

where $call_Buffer(ll, fl)$ is the occurrence number of $\{ll, fl\}$ in $call_Buffer$.

5.2 B abstract operations for deferred events

As the case of non deferred events, we propose to model a deferred event by a B abstract operation and its refinement. In the B abstract operations we should model at the same time the buffering of the deferred events at states where they are deferrable and the handling them in states where they are not deferrable. Return to the event *call*; in the states *movement* and *visit*, *call* is deferred, it should be inserted in $call_Buffer$; in the state *ready*, *call* is no more deferred, it should be removed from $call_Buffer$ (in case it was buffered previously) and triggers transitions (cf. Figure 8) .

Remark 3

1. For the case where $call_Buffer$ is a multi-set (cf. Remark 2), the buffering of an event *call* on the object *ll* is modelled by increasing $call_Buffer(ll, fl)$ by 1. the removing of a call from $call_Buffer$ is modelled by decreasing $call_Buffer(ll, fl)$ by 1.
2. We have not yet considered the fact that *call* is asynchronously sent from the state-chart *Button* to the state-chart *Lift*, which will be discussed in Section 6.2.

5.3 B refinement operations for deferred events

In order to refine B abstract operations for deferred events, it is necessary to introduce B operations for inserting and removing deferred events. Such operations should be defined in a B abstract machine where the corresponding B variable for the internal buffer is defined. The refinement of B operations for deferred events is similar to the refinement of B operations for non deferred events (cf. Section 4.4).

6 Modelling the communication among UML state-charts in B

The solution in the works of Lano, Meyer and Nguyen for modelling the communication between state-charts can be applied in case of synchronous communication but not for the asynchronous communication. In addition, this solution could not deal with the case where a message is diffused to multiple objects. For this reason, the modelling in B of the communication between state-charts is discussed in this section.

```

MACHINE System
SEES Types
...
VARIABLES
  lift, button, ..., call_Buffer
INVARIANT
  ...  $\wedge$  call_Buffer  $\subseteq$  lift  $\times$  FLOOR
INITIALISATION
  ...  $\parallel$  call_Buffer :=  $\phi$ 
...
OPERATIONS
  call(lf,fl) =
  pre
     $lf \in \textit{lift} \wedge fl \in \textit{FLOOR}$ 
  then
    select liftState(lf) = visit then
      /* inserting call in call_Buffer */
      call_Buffer := call_Buffer  $\cup$  {lf  $\mapsto$  fl}
    when liftState(lf) = movement then
      /* inserting call in call_Buffer */
      call_Buffer := call_Buffer  $\cup$  {lf  $\mapsto$  fl}
    when liftState(lf) = ready  $\wedge$  fl = curDestFloor(lf) then
      /* specifying effects of call on related data. */
      liftState(lf) := visit  $\parallel$ 
      doorStatus(lf) := open  $\parallel$ 
      /* removing call from call_Buffer. */
      call_Buffer := call_Buffer - {lf  $\mapsto$  fl}
    when liftState(lf) = ready  $\wedge$  fl > curDestFloor(lf) then
      /* specifying effects of call on related data. */
      liftState(lf) := movement  $\parallel$ 
      curDestFloor(lf) := fl  $\parallel$ 
      dir(lf) := up  $\parallel$ 
      /* removing call from call_Buffer. */
      call_Buffer := call_Buffer - {lf  $\mapsto$  fl}
    when liftState(lf) = ready  $\wedge$  fl < curDestFloor(lf) then
      /* specifying effects of call on related data. */
      liftState(lf) := movement  $\parallel$ 
      curDestFloor(lf) := fl  $\parallel$ 
      dir(lf) := down  $\parallel$ 
      /* removing call from call_Buffer. */
      call_Buffer := call_Buffer - {lf  $\mapsto$  fl}
    else skip end
  end;
...
END

```

Fig. 8. B formalisation for deferred events

6.1 Modelling synchronous messages

The intuitive idea is to avoid calling the B operation of the *message receipt event* in the B operations of the event that sends the message, because both events are modelled in the same abstract machine (*System*). For this purpose, the content of B operation for the message receipt event is inserted in the B operation of the event that sends the message. Consider the example with the event *arrive* that send messages *release* to two objects *Button* (cf. Section 2). As you can notice, the effects of two events *release* are specified in the body of B operations for *arrive* (cf. Figure 9). Note also that the B operations for *release* are no longer needed.

6.2 Modelling asynchronous messages

Since B does not allow to model explicitly the asynchronous communication, we have to simulate it. The idea is similar to the management of deferred events, namely, we create for each signal type a buffer to store signals that have been sent but have not been treated by receiver state-chart. The buffer is written by sender state-chart and read by receiver state-chart. Generally, each item in the buffer should contain information about the receiver(s) objects, eventual parameters of the signals.

In the context of two state-charts for *Lift* and *Button*, the event *call* is no longer deferred in the states *visit* and *movement* but it is send asynchronously (message `self.getLift().call(self.getFloor())`) from the state-chart *Button* to the state-chart *Lift*. The signal buffer for *call* can be defined as followed:

$$\boxed{call_Signal_Buffer \subseteq lift \times FLOOR}$$

Remark 4

1. Remark 2 is still valid.
2. By introducing *call_Signal_Buffer*, we can model the sending of *call* in the B operations of the event *press* (cf. Figure 10).
3. Because *call* is no longer considered as deferred in the context of two state-charts *Lift* and *Button*, the B variable *call_Buffer* is no longer needed. As opposite to *release*, the B operations for *call* are needed and in those operations we model the effects of *call* as well as the removing of *call* from its signal buffer. For reason of space, the modified B operations for *call* is omitted.

7 Integrating UML state-charts into B specifications

This section presents the way to develop B specifications from UML class diagrams and the event-related part of UML state-charts.

```

MACHINE System
SEES Types
...
OPERATIONS
  arrive(ll,fl) =
    pre
       $ll \in \text{lift} \wedge fl \in \text{FLOOR}$ 
    then
      select  $\text{liftState}(ll) = \text{movement} \wedge fl = \text{curDestFloor}(ll)$  then
        /* effects of arrive are effects of itself on the object Lift */
         $\text{liftState}(ll) := \text{visit} \parallel$ 
         $\text{doorStatus}(ll) := \text{open} \parallel$ 
        /* and effects of two events release on two objects Button. */
        any b1,b2 where
           $b1 \in \text{button} \wedge b2 \in \text{button} \wedge$ 
           $\{b1, b2\} = \text{liftButton}^{-1}[\{ll\}] \cap \text{floor}^{-1}[\{fl\}]$ 
        then
           $\text{buttonState} := \text{buttonState} \leftarrow \{b1 \mapsto \text{off}, b2 \mapsto \text{off}\}$ 
        end
      else skip end
    end;
...
END

REFINEMENT System_ref
REFINES System
SEES Types
...
OPERATIONS
  arrive(ll,fl) =
    if  $\text{liftState}(ll) = \text{movement} \wedge fl = \text{curDestFloor}(ll)$  then
      transMovementVisit(ll);
      openDoor(ll);
      var b1,b2 in
         $b1, b2 \leftarrow \text{getButtons}(ll, fl);$ 
        if  $\text{buttonState}(b1) = \text{on}$  then
          transOnOff(b1)
        end;
        if  $\text{buttonState}(b2) = \text{on}$  then
          transOnOff(b2)
        end
      end
    else skip end
  end;
...
END

```

Fig. 9. B formalisation for synchronous messages

```

MACHINE System
SEES Types
VARIABLES
    lift,button,...,call_Signal_Buffer
INVARIANT
    ...  $\wedge$ 
    call_Signal_Buffer  $\subseteq$  lift  $\times$  FLOOR
INITIALISATION
    ... || call_Signal_Buffer :=  $\phi$ 
OPERATIONS
    press(bt) =
    pre
        bt  $\in$  button
    then
        select buttonState(bt) = off then
            /* effects of press */
            buttonState(bt) := on ||
            /* sending call */
            call_Signal_Buffer :=
                call_Signal_Buffer  $\cup$  {liftButton(bt)  $\mapsto$  floor(bt)}
        else skip end
    end;
    ...
END

```

Fig. 10. B formalisation for asynchronous messages

7.1 Data in the B specification

By definition (cf. [18]) the data in the B specification models the data in class and state-charts. Thus, the B data are derived from: (i) classes, association, attributes in the class diagram; (ii) states, sub-states in the state-charts. According to Section 5.1 and Section 6.2, we can also have B data for modelling the buffers of eventual deferred events and signals. In our example, the B data come from: (i) two classes Lift and Button, their attributes and the association between Lift and Button; (ii) the states in the state-charts for Lift and Button and (iii) the buffer call_Signal_Buffer.

7.2 Operations in the B specification

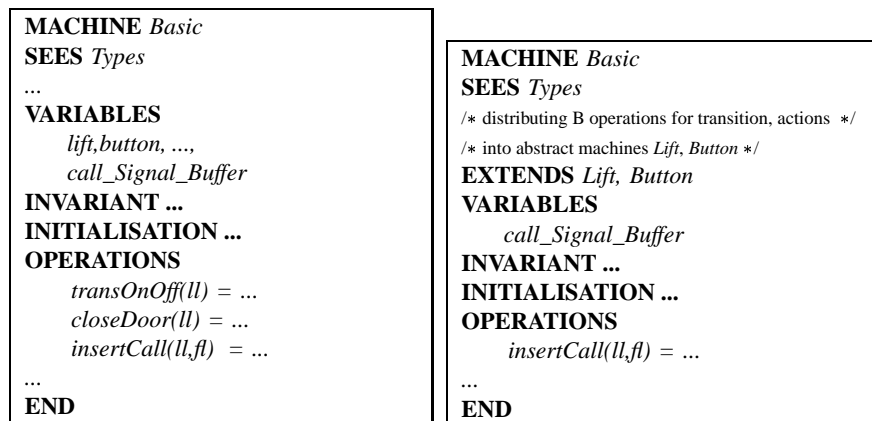
In general, the B operations are mainly used to model events ³, transitions, actions. However, according to Section 5, there are also B operations to model the inserting and removing of eventual deferred events and signals. In case using the B implementation construct, there should be B operations for the state checking and guarded conditions (cf. Section 4.3).

³ except the ones issued from the synchronous communication between state-charts

In our example, the B operations come from: (i) the transitions in state-charts for the classes *Lift* and *Button*; (ii) the class operations (they are basic); (iii) the events *close*, *arrive*, *press* and *call* but not *release* and the inserting and removing of call regarding the buffer *call_Signal_Buffer*.

7.3 Structuring the derived B specification

The operations modelling events are grouped in an abstract machine called *System* (cf. Figure 6 and Figure 8). Data in *System* are described in Section 7.1.



(a) *Basic* models transitions, actions, deferred event and signal inserting/removing

(b) Decomposing *Basic* in abstract machines for classes and associations

Fig. 11. The abstract machine *Basic*

We propose to create another abstract machine (*Basic*) to group B operations for transitions, actions and for inserting/removing eventual deferred events and signals (cf. Figure 11(a)). The data in *Basic* are identical to the *System* data. We refine *System* by including *Basic* so that we can refine B abstract operations for events in *System* by calling B operations of transitions, actions in *Basic*. The identity of data of *Basic* and *System* acts as the implicit gluing invariant in *System_ref*.

We can further structure *Basic* by delegating B operations and related data of transitions and actions to abstract machines for classes and eventual abstract machines for associations. Consequently, there are links **EXTENDS** from *Basic* to abstract machines for classes and eventual abstract machines for class associations; in *Basic*, we declare only data and operations for inserting and removing eventual deferred events and signals (cf. Figure 11(b)). Figure 12 shows the architecture of the B specification corresponding

to the UML specification in Section 2.2. The abstract machine *Basic*, in this case, includes (clause **EXTENDS**) two abstract machines *Lift* and *Button* for classes Lift and Button. The association 1..(2..*) is expressed by the link **USES** from *Button* to *Lift*; no abstract machine for this association is created. The B specification was developed with *AtelierB* and the code can be found in [11].

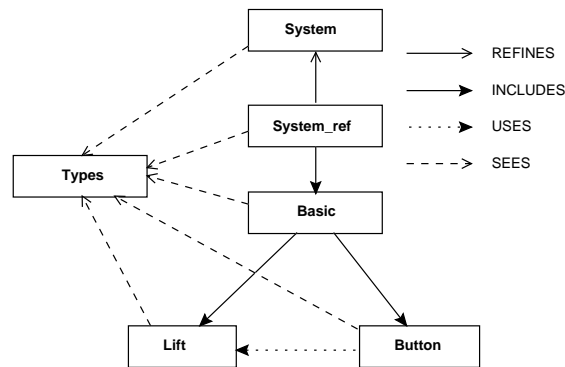


Fig. 12. The derived B specification architecture of the UML specification in Section 2.2

Remark 5

1. From experiences to develop the example in this paper, we have found that *LIFT* must be defined once in *Types*. Otherwise, *LIFT* should be defined twice, in *System* and in *Lift* but this situation is not allowed under *AtelierB*.
2. Furthermore, it is even possible to avoid the constants *LIFT* and *BUTTON*. In that case, *lift* that models the set of effective instances of Lift can be constrained directly to be a subset of *OBJECTS*.
3. Always under the *AtelierB* environment, *Basic* is necessary and we cannot dispense it since in that case, *System_ref* must “**INCLUDES**” directly *Lift* and *Button*, however this is impossible due to the link **USES** from *Button* to *Lift*.

7.4 Generating B operations’ body

We can presently automatically derive the architecture of B specifications. The data, the skeleton of B operations in the B specification are also automatically derived. According to Meyer [18], the B operations for transitions can be automatically derived. According to Section 5 the B operations for inserting and removing eventual deferred events and signals can also be automatically derived. For the purpose of a complete automation of the derivation, we propose to attach OCL-based specifications to events, actions and guard conditions. Hence, the of B abstract operations for events, actions can be derived by using OCL-B rules [17]. The B refinement operations for events can be derived from state-charts. The precise translation rules will be proposed at a later stage.

8 Conclusion

The contributions of our paper consist of a new approach for modelling non deferred events, the modelling of deferred events and the modelling of the communication between UML state-charts, which were not previously treated. Our proposals together with previous derivation schemes for states, actions, transitions give rise to a derivation procedure from the event-related part of UML state-chart in B (cf. Section 7).

Together with previous works [12, 16, 15] we are able to provide a complete framework for deriving B specifications from UML structure and behaviour diagrams. Hence, the conformance between two aspects (the structure and the behaviour) of UML specifications can be formally verified by analysing the corresponding B specification (cf. [14]). For further study, the adaptation of OCL-B translation rules [17] in our work has been envisaged ; a study to translate UML state-charts into B implementation operations is also envisaged. As said earlier, the interference of event-related part to activity-related part which has not been treated so far, is also a subject to investigate. In addition, the prototype Argo/UML+B by Meyer is currently extended to take into account UML behavioural diagrams.

References

- [1] J.R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-521-49619-5.
- [2] A. Amelot and D. Dollé. Le raffinement automatique. Available at http://www3.inrets.fr/B@INRETS/Events/2001-ESTAS/actes/MTI-2001-ESTAS.*, 2001. Slides.
- [3] B-Core(UK) Ltd, Oxford (UK). *B-Toolkit User's Manual*, 1996. Release 3.2.
- [4] J.M. Bruel. Integrating Formal and Informal Specification Techniques. Why? How? In *the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques*, pages 50–57, Boca Raton, Florida (USA), 1998. Available at <http://www.univ-pau.fr/~bruel/publications.html>.
- [5] J.M. Bruel, J. Lilius, A. Moreira, and R.B. France. Defining Precise Semantics for UML. In *Object-Oriented Technology*, LNCS 1964, pages 113–122, Sophia Antipolis and Cannes (F), June 12-16, 2000. ECOOP 2000 Workshop Reader.
- [6] C. Snook and M. Butler. Tool-Supported Use of UML for Constructing B Specifications. draft version.
- [7] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.
- [8] R. Laleau and A. Mammar. A Generic Process to Refine a B Specification into a Relational Database Implementation. In *ZB 2000: Formal Specification and Development in Z and B*, LNCS 1878, York (UK), August/September 2000. Springer.
- [9] R. Laleau and A. Mammar. An Overview of a Method and its support Tool for Generating B Specifications from UML Notations. In *The 15th IEEE Int. Conf. on Automated Software Engineering*, Grenoble (F), September 11-15, 2000.
- [10] K. Lano. *The B Language and Method : A Guide to Practical Formal Development*. FACIT. Springer-Verlag, 1996. ISBN 3-540-76033-4.
- [11] H. Ledang. Case Study : *Lift Control System B Specification*. Available at <http://www.loria.fr/~ledang/case-studies/Lift.zip>, 2001.

- [12] H. Ledang. Des cas d'utilisation à une spécification B. In *Journées AFADL'2001 : Approches Formelles dans l'Assistance au Développement de Logiciels*, Nancy (F), 11-13 juin, 2001. <http://www.loria.fr/~ledang/publications/afadl01.ps.gz>.
- [13] H. Ledang. Formal Techniques in the Object-Oriented Development: an Approach based on the B method. *PhDOOS2001: the 11th ECOOP Workshop for PhD Student in Object-Oriented Systems*, Budapest (Hu), <http://www.st.informatik.tu-darmstadt.de/phdws/wstimetable.html>, June 18-19, 2001. <http://www.loria.fr/~ledang/publications/PhDOOS01.ps.gz>.
- [14] H. Ledang and J. Souquières. Formalizing UML Behavioral Diagrams with B. In *the Tenth OOPSLA Workshop on Behavioral Semantics: Back to Basics*, Tampa Bay, Florida (USA), October 15, 2001. <http://www.loria.fr/~ledang/publications/oopsla01.ps.gz>.
- [15] H. Ledang and J. Souquières. Integrating UML and B Specification Techniques. In *the Informatik2001 Workshop on Integrating Diagrammatic and Formal Specification Techniques*, Vienna (Autria), September 26, 2001. <http://www.loria.fr/~ledang/publications/informatik01.ps.gz>.
- [16] H. Ledang and J. Souquières. Modeling Class Operations in B: Application to UML Behavioral Diagrams. In *ASE2001: the 16th IEEE International Conference on Automated Software Engineering, full paper*, Loews Coronado Bay, San Diego (USA), November 26-29, 2001. <http://www.loria.fr/~ledang/publications/ase01.ps.gz>.
- [17] R. Marcano and N. Lévy. Transformation d'annotations OCL en expressions B. In *Journées AFADL'2001 : Approches Formelles dans l'Assistance au Développement de Logiciels*, Nancy (F), 11-13 juin, 2001.
- [18] E. Meyer. *Développements formels par objets: utilisation conjointe de B et d'UML*. PhD thesis, LORIA - Université Nancy 2, Nancy (F), mars 2001.
- [19] E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. In *FM'99 : World Congress on Formal Methods in the Development of Computing Systems*, LNCS 1708, Toulouse (F), September 1999. Springer-Verlag.
- [20] H.P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles*. PhD thesis, Conservatoire National des Arts et Métiers - CEDRIC, Paris (F), décembre 1998.
- [21] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenzen. *Object-Oriented Modeling and Design*. Prentice Hall Inc. Englewood Cliffs, 1991.
- [22] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998. ISBN 0-201-30998-X.
- [23] E. Sekerinski. Graphical Design of Reactive Systems. In D. Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method - 2nd International B Conference*, LNCS 1393, Montpellier (F), April 1998. Springer-Verlag.
- [24] C. Snook and M. Butler. U2B: a tool for combining UML and B. Available at <http://www.ecs.soton.ac.uk/~cfs98r/U2Bdownloads.htm>.
- [25] C. Snook and M. Butler. Verifying Dynamic Properties of UML Models by Translation to the B Language and Toolkit. Technical Report DSSE-TR-2000-12, Declarative Systems & Software Engineering Group, Department of Electronics and Computer Science University of Southampton, September 2000. Available at <http://www.dsse.ecs.soton.ac.uk/techreports/2000-12.html>.
- [26] C. Snook and R. Harrison. Practitioners Views on the Use of Formal Methods: An Industrial Survey by Structured Interview. *Information and Software Technology*, 43:275-283, March 2001.
- [27] STERIA - Technologies de l'Information, Aix-en-Provence (F). *Atelier B, Manuel Utilisateur*, 1998. Version 3.5.