A Formal Development Method of Control Systems using Event B Approach

Olfa Mosbahi, Jacques Jaray, Leila Jemni Ben Ayed

Formal development method of control systems using the event-based B approach
Case study : A parcel sorting device

Olfa MOSBAHI
Faculty of Sciences of Tunis
University Tunis El Manar II
Tunis, Tunisie
olfa.mosbahi@loria.fr

Jacques JARAY
INRIA Lorraine
LORIA-INPL
Nancy, France
jacques.jaray@loria.fr

Leila JEMNI BEN AYED
Faculty of Sciences of Tunis
University Tunis El Manar II
Tunis, Tunisie
leila.jemni@fsegt.rnu.tn

Abstract

This paper presents a formal method for the development of control systems. We aim at developing a program controlling the operative part of a control system. We first build an abstract model of the operative part and complete this model to get a model of the control system. The elements introduced to change the abstract model of the operative part to the automated system forms the controller of the automated system. The next steps consists in refining the abstract model to get a model of the operative part capturing every important feature. The method is developped through a case study : a parcel sorting system.

1. Introduction

This paper is part of a work concerning the formal development of control systems. We aim at developing a program controlling a plant in which some components with a dynamic behavior must meet some requirements. In the sequel, we will call operative part, the object to be controlled.

We are concerned with the correctness of the controller. The usual way to deal with the correctness of a program is to prove that the interaction with its environment satisfies some required properties. Such requirements are rather difficult to state.

We take a different point of view and notice that the user is satisfied by the behaviour of the controller as soon as the automated system behaves in the expected way. Our development method consists in building an abstract discrete model of the operative part. Future refinements dealing with continuity are out the scope of the paper.

The model of the operative part is completed to obtain a controlled or automated system. Correctness concerns the controlled system, provided the model of the operative part is accurate.

We use the B abstract events technology to build our models. The expected behaviour of the controlled system is expressed by means of invariants. The model is then refined such that its operative part gets closer to the real operative one.

Our main contribution in this paper consists in dealing with control theory results, namely controllability, in formal development. The method is illustrated by a case study and the Atelier B environment [6] is used to prove the correctness of the development.

The paper is organized as follows : section 1 presents an overview of the B event based approach, section 2 a description of the method and section 3 deals with the case study.

2. Overview of the B event-based approach

B refers to a state-based method developed by Abrial [1] for specifying, designing and coding software systems. It is based on mathematical concepts and on Zermelo-Fraenkel set theory, the concept of generalized substitution and on structuring mechanisms (machine, refinement, implementation). Sets are used for data modeling, "Generalised Substitutions" [7, 8, 9] are used to describe state modification, the refinement calculus is used to relate models at varying abstraction levels.

2.1. Reactive systems modelling

To deal with reactive systems, Abrial has proposed a variant of the B method : the B event-based method [2, 10, 3] similar to the action systems by Back [5].

An event consists in a guard and an action. The guard is a predicate built on state variables and the action is a generalized substitution which defines a state transition. An event may be activated once its guard evaluates to true and a single event may be evaluated at once.

We adopt a closed system modeling approach which means that every possible change over state variables is defined by transitions corresponding to events defined in the model.

SYSTEM < *name* >
SETS < *sets* >
VARIABLES < *variables* >
INVARIANT < *invariants* >
INITIALISATION < *initialization of variables* >
EVENTS < *events* >
END

An event can take one of the forms shown in the table below.

Event	*Before-after Predicate* BA(x, x')
$evt = Begin\ x\ :\ P(x,x')\ END$	$P(x,x')$
$evt = SELECT\ G(x)$ $THEN\ x : Q(x,x')$	$G(x) \wedge Q(x,x')$
$evt = ANY\ t\ WHERE\ G(t,x)$ $THEN\ x\ :\ R(x,x',t)\ END$	$\exists t.(G(t,x) \wedge R(x,x',t))$

Proof obligations are produced from events in order to state that the invariant condition $I(x)$ is preserved. We next give the general rule to be proved. It follows immediately from the very definition of the before-after predicate, $BA(x,x')$ of each event :

$$I(x) \wedge BA(x,x') \Rightarrow I(x')$$

2.2. Refinement

Refinement [5, 4] is a technique to deal with the development of complex systems. It consists in building, starting from an abstract model, a sequence of models of increasing complexity (containing more and more details [16]). A model in the sequence follows the one it refines.

The invariant of the refined model is not weaker than the model it refines and it may contain new variables (some variables of the previous model may be suppressed). New events are introduced and they refined skip. It is also used to transform an abstract model into a more concrete version by modifying the state description. The abstract state variables, x, and the concrete ones, y, are linked together by means of a gluing invariant $J(x,y)$. A number of proof obligations ensures that (1) each abstract event is correctly refined by its corresponding concrete version, (2) each new event refines skip, (3) no new event take control forever, and (4) relative deadlock fairness is preserved.

Suppose that an abstract model AM with variables x and invariant $I(x)$ is refined by a concrete model CM with variables y and gluing invariant $J(x,y)$. If $BAA(x,x')$ and $BAC(y,y')$ are respectively the abstract and concrete before-after predicates of the same event, we have to prove the following statement :

$$I(x) \wedge J(x,y) \wedge BAC(y,y') \Rightarrow \exists x'.(BAA(x,x') \wedge J(x',y'))$$

This says that under the abstract invariant $I(x)$ and the concrete one $J(x,y)$, a concrete step $BAC(y,y')$ can be simulated ($\exists x'$) by an abstract one $BAA(x,x')$ in such a way that the gluing invariant $J(x',y')$ is preserved. A new event with before-after predicate $BA(y,y')$ must refine skip ($x' = x$). This leads to the following statement to prove :

$$I(x) \wedge J(x,y) \wedge BA(y,y') \Rightarrow J(x,y'))$$

Moreover, we must prove that a variant $V(y)$ is decreased by each new event (this is to guarantee that an abstract step may occur). We have thus to prove the following for each new event with before-after predicate $BA(y,y')$:

$$I(x) \wedge J(x,y) \wedge BA(y,y') \Rightarrow V(y') < V(y)$$

At last, we must prove that a concrete model does not introduce more deadlocks than the abstract one. This is formalized by means of the following proof obligations :

$$I(x) \wedge J(x,y) \wedge grds(AM) \Rightarrow grds(CM)$$

Where $grds(AM)$ stands for the disjunction of the guards of the events of the abstract model, and grds (CM) stands for the disjunction of the guards of the events of the concrete one.

The essence of the refinement relationship is that it preserves already proved system properties including safety properties. The invariant of an abstract model plays a central role for deriving safety properties and our method focuses on the incremental discovery of the invariant; the goal is to obtain a formal statement of properties through the final invariant of the last refined abstract model. During the development, proof obligations are generated by a computer aided software engineering Atelier B [6]. They are discharged by automatic and interactive proof procedures supported by a proof engine.

3. The method

In this section we present a development method of automated system. The problem can be stated as follows: being given an operative part (physical device and environment) which behavior is observed through a number of variables and on which it is possible to act by means of effectors. Some of the variables are controllable, they correspond to the results of some actuators, other are not controllable and their changes correspond to the effect of some physical process we do not control. We have to develop a control program which observes the operative part and change controllable variables such that the automated system meets some requirements. The behaviour of the automated system can be represented by a closed loop expressing relation between operative and control parts (Fig1). As the B event systems are closed systems and the controller composed with the operative part form a closed system which is a controlled one, the B event method can be used to describe these two components. The modeling method that we propose uses the event based method and consists on the following steps:

1. First of all, we build an abstract model of the operative part in defining its events with guards as weak as possible, just considering safety requirements. A minimal controller is considered such that the resulting system is a “permissive”

controlled system, it is likely not the case that all behaviours meet the automated system requirements but some should.

2. We have to prove the last part of this assertion, if not we can deduce that it is not possible to achieve the expected system. This point is similar to the notion of controllability in control theory [15, 14]. An assertion is added to the typing invariant of the system. Then, we constrain the permissive system such that it obeys some safety requirements, if any,
3. If the proof succeeds, we add the expected behaviour to the invariant, strengthen some guards and proof the improved system. It is worth noticing that the prover in failing to prove some properties gives hints to modify the guards,
4. Successive refinements can be used to produce the concrete model of the physical part and its environment and the concrete control system through the addition of expected behaviour to the invariant, events and guards for operative and control part, and also for their composition. Related proof obligations are discharged. At each refinement step, new model is shown to satisfy new requirements. The refinement process stops when the automated system model meets the desired properties. The quality of the last refinement supposed to model the controlled system depends on the first operative part model.
5. The last step consists in separating what concerns the operative component from what concerns the controller in order to build a program of the controller,

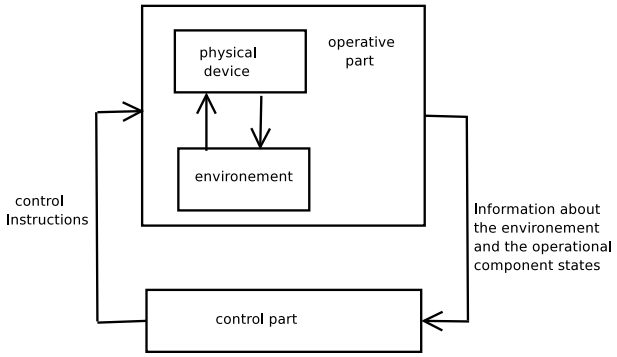

Figure 1. Automated system in closd loop.

In classical formal development methods, only a model of the controller is developed and once implemented it has to be composed with physical device and the required properties are verified. If the verification do not succeed, developer has to produce an other model for the controller. In the proposed method we verify that the composition of the controller and the physical device satisfies requirements at the abstract model and at each refinement step by adding successively invariants, events and guards. Controller is then produced by successive refinement and at each step we show that controlled system maintains environment in states satisfying some properties defined on the variables. The concrete controller is produced such that the controlled system meets all systems requirements. It is more easy to correct an error in one step of the development then correct it in the end. In the following, we illustrate the proposed method throught an exqmple of control system of a parcel sorting device.

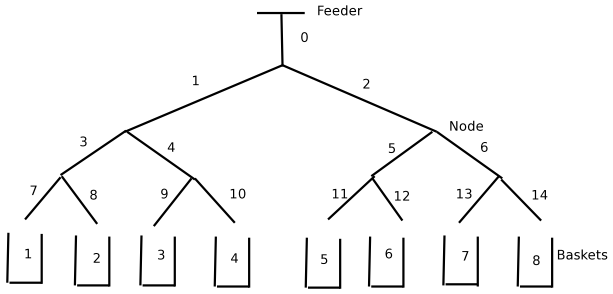

Figure 2. sorting plant.

4. Case study : a parcel sorting device

Our method is illustrated by the development of a case study : a sorting parcel system [11]. We start with an informal description of the problem and then apply our method.

4.1. Informal description of the problem

The problem is to sort parcels into sorting baskets according to an address written on the parcel. In order to achieve such a sorting function we are provided with a device made of a feeder connected to the root of a binary tree made of switches and pipes as shown Fig2. The switches are the nodes of the tree, pipes are the edges and leaves are the baskets. A parcel, thanks to gravity, can slide down through switches and pipes to reach a basket.

A switch is connected to an entry pipe and two exit pipes, a parcel crossing the switch is directed to an exit pipe depending on the switch position. The feeder releases a parcel at once in the router, the feeder contains a device to read the address of the parcel to be released. When released, a parcel enters a first switch (the root of the binary tree) and slides down the router to reach a basket.

The controller can activate the feeder and change the switches position. For safety reasons, it is required that switch change should not occur when a parcel is crossing it. In order to check this condition, sensors are placed at the entry and the exits of each switch. This problem statement is a simplified version of the one given in [11]. We will deal with the complete version in a future work and allow, for instance, the simultaneous sorting of many parcels at the same time.

4.2. Abstract model of the system

The sorting device

The sorting device consists of a feeder and a sorting layout. The feeder has two functions: selection of the next parcel to introduce into the sorting layout and opening the gate (releasing a

parcel in the sorting layout). We introduce the events *select* and *release* to capture the two functions. In order to produce the abstract model of the sorting layout, we have to notice that a given state of the switches forms a *channel* linking the entrance to a unique sorting basket. A basket is an element of a set named *Basket*. In future refinements, the tree structure of the sorting device will be introduced and the sorting baskets will be classed as nodes of the tree (leaves). Therefore, we introduce the set *Nodes* having *Basket* as a proper subset. Channels and sorting baskets are in a one to one correspondence. Therefore, the abstract model of the sorting device can be reduced to a single variable *channel* taking the value of the sorting basket it leads to, namely a value in the set *Basket*. The *channel* value is changed by the event *set_channel*, defined in the control part. The full description of the events will be given in the controller section. It is worth noticing that the abstraction forces a "sequential functionning" of the sorting device, i.e. the value of the channel remains unchanged as long as the parcel released in the sorting device has not reached a sorting basket. Up to a point, it is indeed possible to sort more than one parcel simultaneously. The chosen abstraction does not allow such a concurrency. We intend to deal with simultaneous sorting in a future work.

Parcels

Parcels, as part of the environment, are components of the operative part and are represented as elements of a set we name *PARCELS*. We use a total function (*adr*) from *PARCELS* to the interval *Baskets* to refer to the parcels address. The event concerning a selected parcel to sort is the crossing of the sorting device up to the basket the channel leads to. We give the status "arrived" to the parcel which has reached a sorting basket. The variable (*arrived*) is a total function from *PARCELS* to *Baskets*. The goal of the sorting system is to decrease the set of the parcels to sort. The variable *sorted* represents the set of sorted parcels. The remaining parcels are defined by the expression *PARCELS - sorted* named *UNSORTED*. As *pe* is undefined when the sorting device is empty, we have introduced a set *PPARCELS* of which *PARCELS* is a proper subset; *pe* is an element of *PPARCELS* and assignment of any value in *PPARCELS - PARCELS* stands for "undefined". The expression *PPARCELS - PARCELS* will be referred as *NOPARCELS*. The selection of a parcel is an event which may be activated once the device is free and the variable *pe* is undefined, which means it does not exist a parcel being sorted.

SYSTEM Parcel_Sorting
SETS *PPARCELS* ;
SortingState = {*free* , *busy* }
CONSTANTS *PARCELS, adr, Baskets*
PROPERTIES $PARCELS \subset PPARCELS \wedge$
$PARCELS \neq \emptyset \wedge Baskets \neq \emptyset \wedge$
$adr \in PARCELS \rightarrow Baskets$

select_parcel = ANY
p **Where** $p \in UNSORTED \wedge$
$pe \in NOPARCELS \wedge$
sorting = free
THEN
pe := p
END;

Moving parcels

In our abstraction a parcel takes no time to travel from the feeder to a basket. A parcel arrives in the basket to which the channel leads up. When the event occurs, the current parcel sorting is finished and then, of course, the current parcel becomes undefined.

cross_parcel = SELECT
sorting = busy
THEN
arrived(pe) := channel ||
sorted := sorted $\cup$ { *pe*} ||
pe :: NOPARCELS ||
sorting := free
END;

The safe operating physical layout

From the existing (physical) layout, it is possible to build different controlled systems having different behaviours. It is natural to dismiss systems having unsafe behaviours. The controlled system which respects only these safety constraints, is called the *permissive* controlled system. Here follows the definition of the *permissive* controller system events :

set_channel = SELECT
$sorting = free \wedge$
$pe \notin NOPARCELS \wedge$
$ready\_to\_sort = FALSE$
THEN
channel :: Baskets || *ready_to_sort := TRUE*
END;

release = SELECT
$sorting = free \wedge$
$pe \in PARCELS \wedge$
$ready\_to\_sort = TRUE$
THEN
sorting := busy || *ready_to_sort := FALSE*
END;

Here, the value assigned to the variable *channel* is randomly determined. This justifies the "permissive" qualification. The guard fulfills a safety constraint in preventing a change of the channel when a parcel crosses the sorting device, preventing the parcel to be damaged. The first part of the guard ensures that a single parcel, at once, is being sorted, the second that a parcel may be selected before being released.

4.3. The permissive controlled system

The EVENTS section of the B-event system modelling the permissive controlled system is obtained by merging the controller events.

The event *select_parcel* assigns the (next) current parcel to the variable *pe*. Once selected this parcel remains the current one as long as it has not reached a sorting basket. This captures a non

explicit property of the feeder in which the parcels are stored in a first in first out order.

Controllability

The permissive controller behaviour will likely not satisfy user's needs, in the sense that we can not certify that every parcel arrives in the right basket. Conversely, if we can prove that it is never the case that a parcel arrives in the expected basket, we assess that the system is not controllable with respect of the following requirement : $\forall p.(p \in PARCELS \wedge p \in dom(arrived(p)) \Rightarrow arrived(p) = adr(p))$, and the development process stops here.

We have added the following property to the invariants $\forall p.(p \in PARCELS \wedge p \in dom(arrived(p)) \Rightarrow arrived(p) \neq adr(p))$ and the prover failed in proving btrue = bfalse, which is a contradiction.

The first abstract controlled system

As far as we did not take into account the destination address of the parcel we can not expect a parcel to reach the right basket. The expected behaviour of the automated system can be stated as follows : $\forall p.(p \in PARCELS \wedge p \in dom(arrived(p)) \Rightarrow arrived(p) = adr(p))$. We add it to the invariant part of the B event model.

When attempting to prove the model, we expect a failure and use the report to find out what should be changed to satisfy the requirements.

[fh] [ds] [eh] [he] *sorting = busy*
[fh] [ds] *pe$0: PPARCELS*
[fh] [ds] *not(pe$0: PARCELS)*
[fh] [ds] *p: PARCELS*
[fh] [ds] [rm] *p: dom(arrived <+pe \|− > channel)*
[fh] [ds] [ph] [zm] *!p .([..] => arrived(p) = adr(p))*
[SL] [cl] ————————————
[ae] [aq] [ct] [ss] [ov] *(arrived <+pe\|− >channel)(p) = adr(p)*

When inspecting the interactive prover report, we deduce that it is necessary to change $channel :\in Baskets$ into $channel := adr(pe)$ in the event $set\_channel$ in order to transform the permissive controlled system into an abstract model of the expected controlled system. The technique may seem less convincing as the case study at the level of abstraction is rather simple, but we claim its interest when we are faced to combinatorial situations.

The parcel routing is a particular routing, obtained while making the control event more deterministic. At this abstract level, we have to prove the property asserting that any parcel which is presented in entry will arrive at the destination bin expressed by the following property:

$\forall p.(p \in PARCELS \wedge p \in dom(arrived(p)) \Rightarrow arrived(p) = adr(p))$

For verification purpose,we have added the following invariant properties :

(*sorting* $\neq$ free $\Rightarrow$ *pe* $\in$ *PARCELS*) $\wedge$
(*sorting* = busy $\Rightarrow$ *channel = adr(pe)*) $\wedge$
(*ready_to_sort* = TRUE $\Rightarrow$ *channel= adr(pe)*)$\wedge$
(*ready_to_sort* = TRUE $\Rightarrow$ *pe* $\notin$ *NOPARCELS*) $\wedge$
(*sorting* = busy $\Rightarrow$ *ready_to_sort* = FALSE)

4.4. First refinement

In the abstract model, the operative part of the system is reduced to a single variable *channel*. A channel value corresponds to a setting of node switches. In order to represent a more concrete sorting device, we introduce the nodes *Node* as elements in an interval of integers, the level of the nodes in the sorting device tree structure (the root node corresponding to the level 0) *Level*, a function *adr_level* which gives at a given level the node to be visited to reach a given basket, a variable function *path_level* which associates to each level the node visited. The left successor of the node i is labelled $2 * i + 1$ and the right one $2 * i + 2$. The root node is labelled 0. A pipe has the label of the node it "feeds". The leaves (terminal nodes) of the tree correspond to the baskets introduced in the abstract model, in order to bind the leaves to the basket we introduce a constant function *basket_node*. The following piece of B contains the formal definitions of the items introduced so far.

REFINEMENT **Parcel_routing1**
REFINES **Parcel_routing**

CONSTANTS
adr_level, n, basket_node, Node, Level, INTERNAL_N
PROPERTIES
n $\in$ NATURAL1 $\wedge$
Node = $(0..(2^{(n+1)} - 2)) \wedge$
Level = (0..n) $\wedge$
adr_level $\in$ *(Baskets * Level)* $\rightarrow$ Node $\wedge$
basket_node $\in$ *Baskets* $\rightarrow$ Node $\wedge$
INTERNAL_N = $0..(2^{(n)} - 2) \wedge$
$\forall p.(p \in PARCELS \Rightarrow basket\_node(adr\_level(adr(p), n - 1)) = adr(p)$
VARIABLES
arrived, channel, sorting, pe, sorted, ready_to_sort, path_level, i, l
INVARIANT
path_level $\in$ *Level* $\rightarrow$ *Node* $\wedge$
i $\in$ NATURAL $\wedge$
i $\in$ *Node* $\wedge$
l $\in$ NATURAL $\wedge$
l $\in$ *Level* $\wedge$
$\forall p.(p \in PARCELS \wedge p \in dom(arrived(p)) \Rightarrow arrived(p) = adr(p))$
DEFINITIONS
UNSORTED == PARCELS - sorted ;
NOPARCELS == PPARCELS - PARCELS

The event *cross_parcel* was extended by strengthening its guard with the condition $(l = n - 1)$, meaning that the variable l has

reached the baskets level and the action has been modified *arrived(pe)* is assigned the value *basket_node(adr_level(adr(pe), n-1))*, the (basket) node where the current parcel ends its run.

cross_parcel = SELECT
sorting = busy $\wedge$ *l = n-1*
THEN
arrived(pe) :=
basket_node (adr_level(adr(pe), n-1)) ||
sorted := sorted $\cup$ { *pe*} ||
pe :: NOPARCELS || *sorting := free*
END;

In the abstract model, the crossing of the current parcel concerned a single event *cross_parcel*, here we are concerned by the passing of the parcel through different nodes. The events *cross_right* and *cross_left* are introduced to determine the next node to visit which mean updating the variable i representing the current node. The variable l representing the current level is updated as well. Updating the current node variable depends on the value of *path_level(l+1)*. If the value is odd the next node to reach is the left son otherwise it is the right son. The coding of the nodes makes the computation of the left or right son easy.

cross_right = SELECT
i $\in$ *INTERNAL_N* $\wedge$
l $<$ *n* $\wedge$
*path_level(l+1) = 2*i+2*
THEN
*i := 2*i+2* || *l := l+1*
END;

cross_left = SELECT
i $\in$ *INTERNAL_N* $\wedge$
l $<$ *n* $\wedge$
*path_level(l+1) = 2*i+1*
THEN
*i := 2*i+1* || *l := l+1*
END;

The function *path_level* concerning the current parcel is updated "on the fly" by two events : *path_right*, *path_left*. The *select_parcel* gives the opportunity to initialize *path_level(0)* with 0. All three events are part of the system control.

path_right = SELECT
i$\in$ *INTERNAL_N* $\wedge$
l $<$ *n* $\wedge$
*adr_level(adr(pe),l+1) = 2*i+2*
THEN
*path_level(l+1) := 2*i+2*
END;

path_left = SELECT
i $\in$ *INTERNAL_N* $\wedge$
l $<$ *n* $\wedge$
*adr_level(adr(pe),l+1) = 2*i+1*
THEN
*path_level(l+1) := 2*i+1*
END;

select_parcel = ANY
p **Where** *p* $\in$ *UNSORTED* $\wedge$
pe $\in$ *NOPARCELS*
THEN
pe := p || *i := 0* || *l := 0* ||
path_level(0):= 0
END;

The system has been verified by the B toolkit and some proofs obligations have been verified automatically and others interactively. The last onces were difficult to prove and arithmetic onces were not easily to prove and this is way, we have added some functions in the clause properties. For example, we have added the function power and for some interactive proofs, we have added hypothesis in the verification process.

PROPERTIES
power2 $\in$ NATURAL $\rightarrow$ NATURAL $\wedge$
power2(0) = 1 $\wedge$
$\forall$ *n .(n* $\in$ NATURAL $\Rightarrow$ *power2(n+1)= 2 * power2(n))*

For verification purpose, we have added the following properties in the clause assertions. These properties are verified only once by the proof system.

ASSERTIONS
$\forall$ *n .(n* $\in$ NATURAL1 $\Rightarrow$ power2(n)= 2 * power2(n-1)) $\wedge$
(i $\in$ *INTERNAL_N* $\Rightarrow$ *2*i+1* $\in$ *(0..(power2(n+1)-2)))* $\wedge$
$\forall$ *PP .(PP* $\subseteq$ NATURAL $\wedge$
0 $\in$ *PP* $\wedge$
succ[PP] $\subseteq$ *PP*
$\Rightarrow$
NATURAL $\subseteq$ *PP)*
;
$\forall$ *n .(n* $\in$ NATURAL $\Rightarrow$ (n$>=$ 1 $\Rightarrow$ *power2(n)*$>=$ 2) $\wedge$ (n = 0 $\Rightarrow$ *power2(n)* = 1))

4.5. Second refinement : an even more concrete model of the operative part

This refinement consists in adding details to the nodes. A node is provided with the following sensors : an input sensor, two outputs sensors and a gate which can be set to open the exit to the left pipe or to the right pipe. The position of the gates is represented by a variable function *gate* returning for each node a value in : *R,L*.

The input sensor of a node is represented by a Boolean variable *in* which is initially set to *false*. The outputs sensors are represented by the Boolean variables *out_R* and *out_L*. These variables become true when a parcel is detected and as we cannot have two outputs at the same time, the outputs sensors must verify the property $\neg$ (*out_R* $\wedge$ *out_L*).

Once the parcel is in a node, three events can be started consecutively, the first is *set_gate_left*, which was added to refine control part, this event models gate opening on the left, by putting *gate(adr_level(adr(pe), l+2)):=L*, where *adr_level(adr(pe), l+2)* indicates the current node. The same behaviour but towards right side is modelled by the event *set_gate_right*. The second executed

event is *crossing_left*, respectively *crossing_right*, modelling the node crossing to go towards the left pipe, respectively right pipe, resulting from this node. The third event concerns the pipe crossing and it is modelled by *cross_right*, indicating that the parcel passed towards the next left node, respectively towards the right following node. The variable *gate_set*, is used to guarantee that the parcel crossing is possible only if the node gate is positioned in advance. It is changed at truth, by the event *cross_left* which makes it to false.

REFINEMENT **Parcel_routing2**
REFINES **Parcel_routing1**
SETS
Result = { *R* , *L* , *ind* }
PROPERTIES
$\forall$ *(p, l)* .($p \in$ PARCELS $\wedge$ $l \in$ *NATURAL* $\wedge$ $l < n \Rightarrow$ *adr_level(adr(p), l)*$\in$ *INTERNAL_N*
VARIABLES
arrived, channel, pe, sorted, sorting, ready_to_sort , path_level, i, l, in, out_L , out_R , gate, gate_set
INVARIANT
in $\in$ *Node* $\rightarrow$ *BOOL* $\wedge$ *dom(in)= Node* $\wedge$
out_R $\in$ *INTERNAL_N* $\rightarrow$ *BOOL* $\wedge$ *out_L* $\in$ *INTERNAL_N* $\rightarrow$ *BOOL* $\wedge$ *gate* $\in$ *Node* $\rightarrow$ *sortie* $\wedge$ *gate_set* $\in$ *BOOL* $\wedge$ $\forall$ *i* .(*i* $\in$ *INTERNAL_N* $\Rightarrow$ *l* $\neg$ *(out_R(i)=TRUE* $\wedge$ *out_L(i)=TRUE))*
DEFINITIONS
UNSORTED == PARCELS - sorted ;
NOPARCELS == PPARCELS - PARCELS ;
NBasket == Node - Baskets ;

set_gate_right = SELECT
l <= n-2 $\wedge$
pe $\in$ *PARCELS* $\wedge$
*adr_level(adr(pe),l+2) = 2*adr_level(adr(pe),l+1)+2*
THEN
gate_set := TRUE ||
gate(adr_level(adr(pe),l+1)) := R
END;

set_gate_left = SELECT
l <= n-2 $\wedge$
pe $\in$ *PARCELS* $\wedge$
*adr_level(adr(pe),l+2) = 2*adr_level(adr(pe),l+1)+1*
THEN
gate_set := TRUE ||
gate(adr_level(adr(pe),l+1)) := L
END;

The events *path_left*, respectively *path_right* having for effect to calculate, the next node to be visited by checking that the gate is opened in the good orientation *(gate(ii) = L (left), gate(ii)=R (right))*.

path_right = SELECT
i$\in$ *INTERNAL_N* $\wedge$
$l < n$ $\wedge$
*adr_level(adr(pe),l+1) = 2*i+2* $\wedge$
in(i) = TRUE $\wedge$
gate(i) = R
THEN
*path_level(l+1) := 2*i+2*
END;

path_left = SELECT
i $\in$ *INTERNAL_N* $\wedge$
$l < n$ $\wedge$
*adr_level(adr(pe),l+1) = 2*i+1* $\wedge$
in(i) = TRUE $\wedge$
gate(i) = L
THEN
*path_level(l+1) := 2*i+1*
END;

crossing_right = SELECT
i = INTERNAL_N $\wedge$
in(i) = TRUE $\wedge$
gate(i) = R
THEN
out_L(i) := TRUE ||
out_R(i) := FALSE ||
in(i) := FALSE
END;

crossing_left = SELECT
i = INTERNAL_N $\wedge$
in(i) = TRUE $\wedge$
gate(i) = L
THEN
out_L(i) := TRUE ||
out_R(i) := FALSE ||
in(i) := FALSE
END;

passage_right = SELECT
i = INTERNAL_N $\wedge$
out_L(i) = TRUE $\wedge$
$l < n$ $\wedge$
gate_set = TRUE $\wedge$
*path_level(l+1) = 2*i+2*
THEN
*i := 2*i+2* ||
l := l+1 ||
gate_set := FALSE ||
*in(2*i+1) := TRUE*
END;

passage_left = SELECT
i = INTERNAL_N ∧
out_L(i) = TRUE ∧
l < *n* ∧
gate_set = TRUE ∧
*path_level(l+1) = 2*i+1*
THEN
*i := 2*i+1* ||
l := l+1 ||
gate_set := FALSE ||
*in(2*i+1) := TRUE*
END;

The event *select_parcel*, is also refined by taking into account the sensor which models the parcel presence on the level of the first entry basket. The event *cross_parcel*, concerning the operative part is refined by adding to the level of each node the sensor which detects the entry of parcels.

select_parcel = ANY *p* **Where**
p ∈ *UNSORTED* ∧
pe ∈ *NOPARCELS*
THEN
pe := p ||
i := 0 ||
l := 0 ||
path_level(0):= 0 ||
in(0) := TRUE
END;

cross_parcel = SELECT
sorting = busy ∧
l = n-1
THEN
arrived(pe) := basket_node (adr_level(adr(pe,n-1)) ||
sorted := sorted ∪ { *pe*} ||
pe :: NOPARCELS ||
sorting := free ||
in(basket_node(channel)) := TRUE
END;

The fact that almost every clause of the invariant has a universal quantification increased considerably the complexity of the proof obligations. The total number of proofs amounts to 90 and 74 where automatically proved by the Atelier-B prover. The others proof obligations are interactive and where difficult to prove. Nevertheless, all were proved.

5. Conclusion

In this paper, we have proposed the first steps of a formal method for the development of hybrid systems using the B event-based approach. We have discarded time features and will deal with in future work. The main contribution of the paper consists in building a model of the expected automated system starting from a model of the physical operative part. In the stepwise approach, the first model consists of an abstract model of the operative part with the guards of the event kept as week as possible to prevent unsafe behaviours. The only property one can expect from a system, we have called permissive, is that it does not prevent the expected behaviour of the final automated system to occur. If it is the case, we deduce that the system is controllable and that is is possible to strengthen the guards in order to obtain an abstract model of the required controlled system. The controlled system is then refined in the usual way. The refinement, indeed, concerns the operative part. At the end of the process, we have a detailed model of the operative part. The validation of the operative part, using the animator has been experienced in another work.

In the future, we will deal with temporal properties using, the temporal logic of actions TLA+ [12, 13] as B is limited to invariant properties and does not allow to deal easily with fairness properties.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] J.-R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habrias, editor, *Proceedings of the 1st Conference on the B method*, pages 169–191, Nov. 1996.

[3] J.-R. Abrial. Event driven circuit construction. MATISSE project, Aug. 2000.

[4] R.-J. Back and K-Sere. Stepwise refinement of action systems. In *Mathematics of Program Construction.*, pages 115–138, Berlin - Heidelberg - New York, June 1989. Springer.

[5] R.-J. Back and J. v. Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[6] ClearSy. Atelier b. Technical Note Version 3.6, Aix-en-Provence(F), 2002.

[7] E. Dijkstra. *A Discipline of Programming*, chapter 14. Prentice-Hall, Englewood Cliffs, N. J., 1976.

[8] E. Dijkstra and C. Schweten. *Predicate Calculus and Programm Semantics*. Springer Verlag, New York, 1990.

[9] R.-M. Dijkstra. An experiment with the use of predicate transformers in UNITY. *Information Processing Letters*, 53(6):329–332, Mar. 1995.

[10] D. C. J-R. Abrial and G. Laffitte. Higher-order mathematics in b. In H. Habrias, editor, *Formal Specification and Development in Z and B*, pages 237–270, Nov. 1996.

[11] J. JARAY and A.Mahjoub. Une mthode itrative de construction d'un modle de systme ractif . *TSI*, 15, 1996. .

[12] L. Lamport. The Temporal Logic of Actions. Technical Report 79, Digital Equipment Corporation, Systems Research Centre, Dec. 1991.

[13] L. Lamport. Hybrid systems in TLA^{+}. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 77–102. Springer-Verlag, 1993.

[14] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, 1987.

[15] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE; Special issue on Dynamics of Discrete Event Systems*, 77, 1:81–98, 1989.

[16] J.-M. Spivey. Understanding Z, A Specification Language and its Formal Semantics. *Tracts in Theoretical Computer Science*, 3, 1988. Cambridge University Press.