Proved Development of the Real-Time Properties of the IEEE 1394 Root Contention Protocol with the Event B Method

Joris Rehm

Noname manuscript No.
(will be inserted by the editor)

Proved Development of the Real-Time Properties of the IEEE 1394 Root Contention Protocol with the Event B Method

Root Contention Protocol with Event B

Joris Rehm

Received: date / Accepted: date

Abstract We present a model of the IEEE 1394 Root Contention Protocol with a proof of Safety. This model has real-time properties which are expressed in the language of the event B method: first-order classical logic and set theory. Verification is done by proof using the event B method and its prover, we also have a way to model-check models. Refinement is used to describe the studied system at different levels of abstraction: first without time to fix the scheduling of events abstracly, and then with more and more time constraints.

Keywords Formal method · Real-time · Event-B method · Theorem proving

1 Introduction

In this paper, we present a model of the IEEE 1394 Root Contention Protocol with a proof of safety and of real-time properties. We already described the pattern of our model of time, applied in a simple case study, in [8] as a pattern of refinement for the event B method. We show here how this pattern works over a proven development of the IEEE case study. Many different models for real-time already exist. Our goal is to find a model of time adapted to allow proof by invariant with refinement over systems of events. We also argue that is better to start a proven development by an abstract model without time and to use refinement to add real-time properties. Therefore our model of time must allow us to use refinement.

This work was supported by grant No. ANR-06-SETI-015-03 awarded by the Agence Nationale de la Recherche.

Joris Rehm
Université Henri Poincaré Nancy 1 - LORIA
BP 239 - 54506 Vandœuvre-lès-Nancy - France
E-mail: joris.rehm@loria.fr

The IEEE 1394, also known as FireWire, is used to connect devices like external hard-disks or movie cameras. Devices are able to configure themselves by the IEEE 1394 leader election protocol. This protocol takes the network as an acyclic graph and orients edges to obtain a spanning tree rooted by a leader. This scenario has already been modelled with the event B method in [4]. This work extends this result to the following case: at the end of the algorithm, or when only two devices are connected, the general algorithm can fail. In this case the signals can cross in the bi-directional channel between the two devices if they send signals at almost the same send time. It is possible because there are separate cables in both directions. Consequently the IEEE 1394 Root Contention Protocol takes place in order to choose a leader between the two devices. The algorithm is probabilistic and uses a random choice between a short and a long waiting time. This sleeping time and signal sending between devices leads to a (probable) election. We do not take into account probabilistic properties (we replace them by non-determinism) nor the loss of signals. To model this system we need to quantify the two different sleeping times and the progression of signals over the channels. We want to prove safety properties on this algorithm. And we want to be compatible with the existing B model ([4]) as we want to keep the possibility to use the refinement relation between models. Furthermore, we want to use the language and tools of B without modifications.

The language of the B method is based on the first order classical logic with set theory. This method can be used for specifying, designing and coding software systems. B models of system are accompanied by mathematical proofs. Proofs validate an invariant over the events of the system and validate the refinement relation between models. The goal of the refinement is

to connect an abstract specification to a more concrete model. And step by step we can reach a precise model of the implementation. Of course, in this paper, every formal descriptions are followed by textual explanations.

The language of the B method does not contain specific real-time or distributed features but we can model them. The idea is to guard events with a time constraint like a timeout or an alarm. We say that events are linked to an "activation time" (AT). We have several sets of AT, one for each constrained timed event which corresponds to encoding a multiset of ATs.

To represent the real-time progression we use a global clock represented by the variable *time*. Our time is discrete so $time \in \mathbb{N}$, but we can use unknown constants or logical expressions between different times. The time progression is expressed by an event called *tick*. No events except *tick* make the time progress therefore several events can trigger in the same clock granule. This event nondeterministicly increases the variable *time* between $time+1$ and the first activation time (if any). So we have in invariant: $(at \neq \varnothing \Rightarrow time \leq \min(at))$ where the variable *at* is the union of all different AT sets. As *time* is a natural number we are sure that the system will reach the next active time if *tick* is activated often enough. Finally, when time reaches an AT value we have $time \in at$; in other words, $time = min(at)$. Therefore the event linked to this AT can trigger, do its work and remove the reached AT from its AT set. After this suppression, *time* is free to reach the next AT, or simply increase if $at = \varnothing$.

This paper is organised as follows. In section 2, we introduce the Event-B method; and in section 3 a real-time model for Event-B. In section 4, 5, 6 and 7 we show the model of the case-study. We follow the methodology of refinement and we introduce respectively: the goal of the system, the details of the system without time, the real-time properties of the signals passing and the real-time properties of the sleeping times. In Section 8, we show the verification of the model. Finally in Section 9, we show related-work and we conclude.

2 Overview of event-B development by step-wise refinement

2.1 Event-based modelling

This event-driven approach [2] is based on the B notation. It extends the methodological scope of basic concepts in order to take into account the idea of *formal models*. Roughly speaking, a formal model is characterised by a (finite) list x of *state variables* possibly modified by a (finite) list of *events*; an invariant $I(x)$ states properties that must always be satisfied by the variables x and *maintained* by the activation of the events. In the following, we briefly recall definitions and principles of formal models and explain how they can be managed by tools [3,9].

Generalised substitutions are borrowed from the B notation. They provide a means for expressing changes to state variable values. In its simple form, $x := E(x)$, a generalised substitution looks like an assignment statement. In this construct, x denotes a vector built on the set of state variables of the model, and $E(x)$ a vector of expressions. However, the interpretation we shall give here to this statement is not that of an assignment statement. We interpret it as a *logical simultaneous substitution* of each variable of the vector x by the corresponding expression of the vector $E(x)$. There exists a more general normal form, denoted by the construct $x : |P(x, x')$. This should be read: "x is modified in such a way that the predicate $P(x, x')$ holds", where x' denotes the *new value* of the vector and x denotes its *old value*. This is clearly non-deterministic in general.

An event has two main parts: a *guard*, which is a predicate built on the state variables, and an *action*, which is a generalised substitution. An event can take one of the three normal forms. The first form ($evnt \mathrel{\widehat{=}}$ BEGIN $x : |P(x, x')$ END) represents an event that is not guarded: it is thus always enabled and is semantically defined by $P(x, x')$. The second ($evt \mathrel{\widehat{=}}$ WHEN $G(x)$ THEN $x : Q(x, x')$ END) and third ($evt \mathrel{\widehat{=}}$ ANY t WHERE $G(t, x)$ THEN $x : R(x, x', t)$ END) forms are guarded by a guard which states the necessary condition for these events to occur. Such a guard is represented by WHEN $G(x)$ in the second form, and by ANY t HERE $G(t, x)$ (for $\exists\, t \cdot G(t, x)$) in the third form. We note that the third form defines a possibly non-deterministic event where t represents a vector of distinct local variables. The, so-called, before-after predicate $BA(x, x')$ associated with each of the three event types, describes the event as a logical predicate expressing the relationship linking the values of the state variables just before (x) and just after (x') the "execution" of event *evt*.

Proof obligations are produced from events in order to state that an invariant condition $I(x)$ is preserved. Their general form follows immediately from the definition of the before-after predicate, $BA(x, x')$, of each event:

$$\boxed{I(x) \wedge BA(x, x') \;\Rightarrow\; I(x')}$$

Note that it follows from the two guarded forms of the events that this obligation is trivially discharged when the guard of the event is false.

2.2 Model Refinement

The refinement of a formal model allows us to enrich a model in a *step-by-step* approach, and is the foundation of our *correct-by-construction* approach. Refinement provides a way to strengthen invariants and to add details to a model. It is also used to transform an abstract model into a more concrete version by modifying the state description. This is done by extending the list of state variables, by refining each abstract event into a corresponding concrete version, and by adding new events. The abstract state variables, x, and the concrete ones, y, are linked together by means of a, so-called, *gluing invariant* $J(x, y)$. A number of proof obligations ensure that (1) each abstract event is correctly refined by its corresponding concrete version, (2) each new event refines *skip*, (3) no new event takes control for ever, and (4) relative deadlock-freeness is preserved. Details of the formulation of these proofs follows.

We 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, corresponding to proof obligation (1):

$$\boxed{\begin{array}{l} I(x) \wedge J(x,y) \wedge BAC(y,y') \\ \quad \Rightarrow \exists x' \cdot (BAA(x,x') \wedge J(x',y')) \end{array}}$$

Now, proof obligation (2) states that $BA(y, y')$ must refine *skip* ($x' = x$), generating the following simple statement to prove (2):

$$\boxed{I(x) \wedge J(x,y) \wedge BA(y,y') \;\Rightarrow\; J(x,y')}$$

For the third proof obligation, we formalise the notion of the system advancing in its execution; a standard technique is to introduce a variant $V(y)$ that is decreased by each new event (to guarantee that an abstract step may occur). This leads to the following statement to prove (3):

$$\boxed{I(x) \wedge J(x,y) \wedge BA(y,y') \;\Rightarrow\; V(y') < V(y)}$$

Finally, to prove that the concrete model does not introduce additional deadlocks, we give formalisms for reasoning about the event guards in the concrete and abstract models: $\mathsf{grds}(AM)$ represents the disjunction of the guards of the events of the abstract model, and $\mathsf{grds}(CM)$ represents the disjunction of the guards of the events of the concrete model. Relative deadlock freeness is now easily formalised as the following proof obligation (4):

$$\boxed{I(x) \wedge J(x,y) \wedge \mathsf{grds}(AM) \;\Rightarrow\; \mathsf{grds}(CM)}$$

To review, refinement guarantees that the set of traces of the refined model contains (modulo stuttering) the traces of the resulting model.

3 Real-time modelling in Event-B

The formal method Event-B is untimed, we proposed in [8] extending it in order to model time with a pattern. For the sake of completeness we recall some content from [8] in a updated form and we show how this generic pattern is used in this case study. This pattern shows generic forms for event suited to express the different aspect of real-time. We call it a pattern because it represents a general domain or aspect (like real-time properties or constraints) and not a particular properties of a system. Thus we argue that pattern can be re-use in several systems study of the same domain. We use the variables now in $\mathbb{N}$ to represent the current time, hence we use a discrete time. And the variable at (stands for Activation Times) is a function from a set $labels$ of real-time activity labels to a subset of $\mathbb{N}$: : $at \in labels \rightarrow \mathbb{P}(\mathbb{N})$. This fonction at is used to give for every label in $dom(at)$, a set of activation times in the future, like in a calendar. Therefore, we have in invariant:

$$\forall e \cdot (e \in evts \wedge at(e) \neq \varnothing \Rightarrow now \leq min(at(e)))$$

We now give the pattern which shows how to write an event-B model of a real-time system. To do that each event of the new model will refine one (or maybe several) event(s) of the pattern. The resulting events will inherit the real-time properties of the pattern.

Let is start by the initialisation:

initialisation $\widehat{=}$
BEGIN
act1: $now := 0$
act2: $at :\in labels \rightarrow \mathbb{P}(\mathbb{N})$
END

The current time now is set to zero and at can be set to any total fonction from $labels$ to $\mathbb{P}(\mathbb{N})$. In practice, in this case study, at will be set to $\{l \mapsto \varnothing | l \in labels\}$.

The event **add** shows how to add a new future activation time $ntime$ for the activity e:

add $\widehat{=}$
ANY
e
$ntime$
WHERE
grd1: $e \in dom(at)$
grd2: $now < ntime$
THEN

act1: $at(e) := at(e) \cup \{ntime\}$
END

In the event **use** the activity e is activated, at the current time now, but only if this activation has been planned in the set $at(e)$ ($now \in at(e)$); in this case we remove the current time from $at(e)$.

use $\widehat{=}$
ANY
e
WHERE
grd1: $e \in dom(at)$
grd2: $now \in at(e)$
THEN
act1: $at(e) := at(e) \setminus \{now\}$
END

Finally the event **tick** represents the time progression, we increase the current time at least to $now + 1$ and at most to the smallest time of the calendar (if any).

tick $\widehat{=}$
ANY
n_now
WHERE
grd1: $now < n\_now$
grd2: $\forall e \cdot e \in dom(at) \wedge at(e) \neq \varnothing$
$\Rightarrow n\_now \leq min(at(e))$
THEN
act1: $now := n\_now$
END

This initialisation and the three events **add**, **use**, and **tick** are in a general form and now we can refine them for the case study. We need two activities *pass* and *awake* for two devices a and b. Hence we choose for the set *labels*:

$$labels = \left\{ \begin{array}{l} a\_pass, \\ b\_pass, \\ a\_awake, \\ b\_awake \end{array} \right\}$$

In the next sections we develop step by step the final model of the system. Each steps will be a refinement between two models. First we introduce the untimed aspect of the system: the basic specification and the components like channel. This leads to create (untimed) events and secondly we introduce the real-time constraints over those events. Each events which adds a real-time delay for another event will use the pattern events **add**. And each events which happen when the delay is exhausted will use the pattern events **use**. Finally **tick** is added as it is to the model.

One last remark to conclude: instead of using directly the function at we can use the four labels as subset of $\mathbb{N}$, for example for the label a_pass instead of $at(a\_pass)$ we use at_a_pass (with $at\_a\_pass \in \mathbb{N}$). In the following we call those sets "Activation Time" (AT) because that is equivalent and sets are more friendly in this set-based language.

4 First Model

This first model is the most abstract specification of our system. The general behaviour is to choose (elect) one device in the set $N = \{a, b\}$. The only variable *leader* is a subset of N and contains the chosen device (if not empty). Apart from this, the invariant states that the set *leader* is $\{a\}$ or $\{b\}$ or $\varnothing$. In the initialisation of the system the variable *leader* is set to $\varnothing$ (because no leader is elected).

Finally the transitions of this abstract system are given by the event *accept*:

accept $\widehat{=}$
ANY x WHERE
x ∈ N ∧
$leader = \varnothing$
THEN
leader := {x}
END

This event occurs when the set *leader* is empty and fills it with a device. After this transitions the guard of *accept* is false so no more transitions are possible. All of the following models in the paper will refine this behaviour.

5 First Refinement

We now introduce, through refinement, the local state of devices and two communication channels between device a and b. Almost all behaviour of the system can already be expressed abstractly at this level of abstraction. Communication will be asynchronous. In other words a signal from a to b can cross a signal from b to a. So the system can progress in two ways: if only one signal is sent, a leader will be elected; if two signals cross, a situation called "contention" will appear. In this situation the election is impossible, so the two devices will remove their signals. After a contention the devices wait for a random length of time before retrying the election process. We note that real-time properties will be added later and that model is a specification for the future real-time properties. For example, we will add a precise propagation time for the signal progress in a channel.

Devices communicate with two $SIGNALS$: IDL and PN. IDL is the initial and idle signal. PN (for

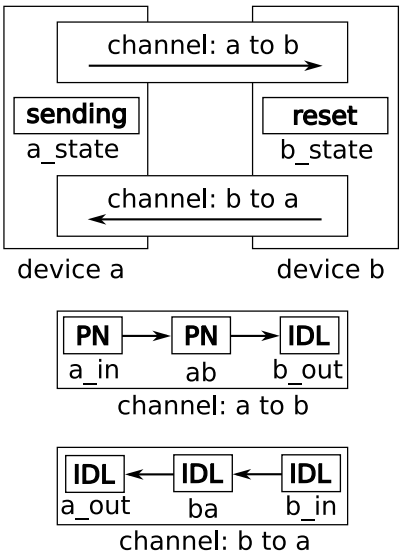

Fig. 1 Devices and channels of the system

Parent Notify) means that the sender does not want to be the leader. We have four different $STATES$ for the two devices a and b, devices are in the state: *reset* when they start; *sending* when they are sending the signal PN; *sleeping* when they wait after a contention; and *accepting* when they accept to be the leader. In the refinement, the variable *leader* disappears through data refinement and we add nine new variables: variables *a_state* and *b_state* for devices and three variables for each channels *a_in*, *ab*, *b_out* and *b_in*, *ba*, *a_out*. (The names of variables are chosen with the intuition that the signal go into the channel, not outside the devices). We need three variables for a channel because we want asynchronous communication and we can have a maximum of two changes of signals at the same time. The three variables of a channel act in a "first in first out" way and we have an event to make the values progress inside the channel from the input to the output. Finally, the variable *case* does not take place in the behaviour of the system but is used to denote a special case in the invariant. We can see a graphical representation in Fig. 1, which shows the very first sending of the signal PN.

5.1 Invariant

The first point to specify in an invariant is the type of the new variables: all variables of the channels are members of the set $SIGNALS$; Variables *a_state* and *b_state* are in $STATES$; and *case* is a boolean.

In this refinement the variable *leader* is not required anymore because we can deduce the leader of the election from the state of the devices. So we can replace the abstract variable *leader* by the concrete variables *a_state* and *b_state*. We call this a "data refinement". For this we need a "gluing invariant" that relates the value of the abstract variable with the value of the concrete variables. Here we want ($leader = \{a\} \Leftrightarrow a\_state = accepting$) and the same thing for b. We also know that if the PN signal is present in both channels then $leader = \varnothing$. Furthermore, if one device is accepting then the other is sending (the signal PN) and all signals have been received. (For device a: leader = {a} ⇒ a_state = accepting ∧ b_state = sending ∧ ab = IDL ∧ b_out = IDL ∧ ba = PN ∧ a_out = PN.)

This part of the invariant is the most important, but there are more things to express. First of all, in the initial state of the system devices are *reset*. With that condition all variables of the channel from this device are equal to IDL, for device a: a_state = reset ⇒ (a_in = IDL ∧ b_out = IDL ∧ ab = IDL). We also know that if a device is reset the other one is either reset or sending. When a device, for instance a, is sending then the beginning of the channel is set to PN, for instance $a\_in = PN$. We have the equivalence: ($a\_state = sending \Leftrightarrow a\_in = PN$). We are also sure that if a device is reset and this device receives the PN signal then the other must be sending. In this case, the receiving device can safely accept to be the leader. Consider the case where the receiving device is also in the state *sending*: the election is now impossible and we are in a situation of "contention". The device discovers this situation and sets its state to *sleeping*. In this state we have for device a: $a\_in = IDL$ and $b\_state \in \{sending, sleeping\}$ (and the symmetric case for b: b_state = sleeping ⇒ b_in = IDL ∧ b_state ∈ {sending, sleeping}). So the signal PN will be erased by IDL. After this, each device will go back to the *sending* state and the previous part of the invariant is used to describe the state. The level of modelling is quite abstract and we will see in the final refinement many more (real-time) statements over states, especially in the situation of contention.

Next, we express a simplification in the use of the channels: if we will have only one change of signal progressing in a channel we can directly put the progression at the final step. Because it is only important for the environment, the devices do not have anything special to do. For example, for the channel from a to b we can have: ($a\_in = ab \wedge ab = b\_out$) or ($a\_in = ab \wedge ab \neq b\_out$) or ($a\_in \neq ab \wedge ab \neq b\_out$) but not ($a\_in \neq ab \wedge ab = b\_out$). Therefore we add: if $a\_state = sending$ then $ab = PN$, and the same for b.

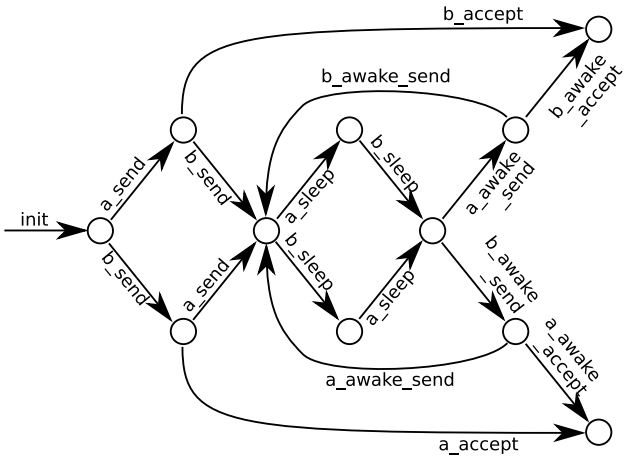

Fig. 2 Transitions of the two devices

Finally the *case* boolean variable is true if and only if: one of devices is sending and the other is sleeping: case = TRUE ⇒ (a_state = sending ∧ b_state = sleeping) ∨ (b_state = sending ∧ a_state = sleeping); and the device currently sending was previously in the state *sleeping*: property ensure by the events.

The variable *case* expresses this with relations between values of *case* and values of other variables. This is the key to the algorithm because when a device goes back to the state *sending* after *sleeping*, it can elect a leader if the duration of this case is long enough. We will see the utility of *case* in the last refinement. In other words, if the second device awakes a long time after the first: then the first device has enough time to send their signal. Otherwise, the contention reappears.

5.2 Events

Events give properties over the transitions of the system. We can see on the Fig. 2 the transition graph of devices, transitions about the progression of message over channel variables are not represented. Here we have four kinds of events: *send*, *pass*, *accept* and *sleep*. From now on we will describe the model only from the point of view of the device a. As the system is totally symmetric between devices a and b, the reader can easily fill in the blanks.

init $\widehat{=}$
BEGIN
$a\_in, b\_in, a\_out, b\_out, ab, ba :=$
$IDL, IDL, IDL, IDL, IDL, IDL \|$
$a\_state, b\_state := reset, reset \|$
$case :=$ FALSE
END;

a_send $\widehat{=}$
WHEN
$a\_state = reset \wedge$
$a\_out = IDL$
THEN
$a\_state := sending \|$
$a\_in := PN \|$
$ab := PN$
END;

b_send $\widehat{=}$ ...

ab_pass_out $\widehat{=}$
WHEN
$ab \neq b\_out \wedge$
$(b\_state \neq sending \vee b\_out \neq PN)$
THEN
$b\_out := ab \|$
$ab := a\_in$
END;

ba_pass_out $\widehat{=}$ ...

pass_out $\widehat{=}$
WHEN
$ab \neq b\_out \wedge$
$ba \neq a\_out$
THEN
$b\_out := ab \|$
$ab := a\_in \|$
$a\_out := ba \|$
$ba := b\_in$
END;

a_accept $\widehat{=}$
REFINES accept WHEN
$a\_state = reset \wedge$
$a\_out = PN$
THEN
$a\_state := accepting$
END;

b_accept $\widehat{=}$ ...

a_sleep $\widehat{=}$
ANY new_ab WHERE
$a\_state = sending \wedge$
$a\_out = PN \wedge$
$new\_ab \in SIGNALS \wedge$
$(ab = b\_out \Rightarrow new\_ab = IDL) \wedge$
$(ab \neq b\_out \Rightarrow new\_ab = PN)$
THEN
$a\_state := sleeping \|$
$a\_in := IDL \|$
$ab := new\_ab$
END;

b_sleep $\widehat{=}$ ...

a_awake_send $\widehat{=}$
WHEN
$a\_state = sleeping \wedge$
$a\_out = IDL \wedge$
$ab = IDL \wedge$
$b\_out = IDL$
THEN
$a\_state := sending \|$
$a\_in := PN \|$
$ab := PN \|$
$case := \neg case$
END;

b_awake_send $\widehat{=}$ ...

a_awake_accept $\widehat{=}$
REFINES accept WHEN
$a\_state = sleeping \wedge$
$a\_out = PN \wedge$
$b\_state = sending \wedge$
$ab = IDL \wedge$
$b\_out = IDL$
THEN
$a\_state := accepting \|$
$case := \neg case$
END;

b_awake_accept $\widehat{=}$ ...

As written with the keyword "REFINES", four events refine the abstract event *accept*. : *a_accept*, *b_accept*, *a_awake_accept* and *b_awake_accept*.

The guard of the event *a_send* express that *a* has never sent anything and is not receiving a signal. With this condition we send a signal to the device *b*. The substitution: $ab := PN$ comes with a simplification in the use of the channel.

The *ab_pass* event shows how a signal progresses in the channel from *a* to *b*. We know that ($a\_in \neq ab \Rightarrow ab \neq b\_out$). Therefore when there is at least one change of signal in the channel we know that $ab \neq b\_out$. Then we advance values from *a_in* to *ab* and from *ab* to *b_out*. However, when there are changes of signals in both channels we can make the values pass in both channels at the same time. This is done by the event *pass_out*. Without this event, in this abstract model, one channel can take priority over the other, for example with several activations of *ab_pass_out* without activations of *ba_pass_out*. This is not realistic behaviour, but in model with real-time properties this problem is solved. In the last line of the guard of *ab_pass_out* we can see an expression of the priority of the event *b_sleep*. To express priority, we take the guard (or a crucial part of the guard) of the event with the higher priority and put its negation in the guard of the event with the lower priority. A part of the guard is crucial if it negation is enought to prevent the execution of the event in any case. Hence both events can not trigger in the same state. Without this priority the system could execute the sequence: *a_send*, *b_send*, *pass_out*, *a_sleep*, *ab_pass*, *a_accept*. However, such a sequence is not allowed by the standard as a device has to have discovered the contention situation. In this sequence the device *b* failed to discover it (i.e. to execute *b_sleep*). In this refinement the problem is solved abstractly and in later models we will use real-time constraints.

The event *a_accept* triggers when the situation of contention never occurred (impossible when a state is still *reset*) and a device receives a signal PN.

In contrast, a device can discover a situation of contention when it is sending and it has received a signal PN. In this case it goes to sleep and starts to remove its signal PN. This is the only state where we can have two changes of signal in the same channel. It's achieved when we already have $ab \neq b\_out$. Therefore the new state of the channel will be ($a\_in = IDL \wedge ab = PN \wedge b\_out = IDL$).

Finally, there are two events left: *a_awake_send* and *a_awake_accept*. They contain the same behaviour as *a_send* and *a_accept* plus the management of the variable *case* and some extra conditions.

In addition to the guard of *a_send*, the event *a_awake_send* must check that the previous signal PN of device *a* is erased from the channel from *a* to *b*. This can be seen in the last two lines of the guard of *a_awake_send*.

In addition to the guard of *a_accept*, the event *a_awake_accept* must check that the signal is erased in the same way. It has also to check if the device *b* is actually in the *sending* state. This is done by the third line of the guard. Otherwise one device would accept to be the leader with the other device in state *sleeping* because $a\_out = PN$ does not imply that $b\_state = sending$; and it would lead to an incorrect election situation.

In conclusion, this model is enough to express all safety properties abstractly and now we can introduce the real-time.

6 Second Refinement

In this second refinement, we add a precise propagation time for the propagation of a signal inside a channel. This constant, called *prop*, is in $\mathbb{N}$ and is not equal to zero. We can see on the Fig. 3 an example timeline of a communication, each event activations are indicated with the name of the event. This model contains three new variables: *time*, *at_a_pass* and *at_b_pass*. The variable *time* is a global clock, the value of *time* represents

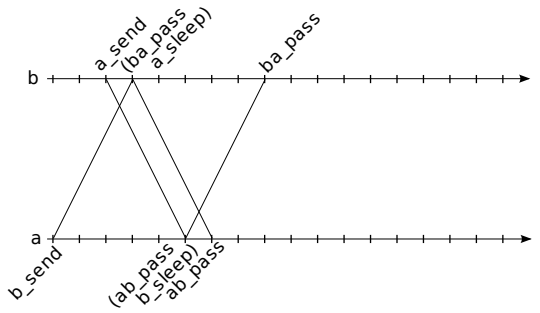

Fig. 3 Timeline of the system with $prop = 3$

"now" the current time. The algorithm itself does not require a global clock or a synchronisation between devices. But in the B method we need closed systems, i.e. we need to model the environment. Time is a part of the environment of our devices and it plays a central role in the invariant. The two other variables are two sets of "Activation Time" (AT) i.e. a set of timeouts or alarms. Each AT set is linked to some particular event. Here the set at_a_pass (respectively at_b_pass) is linked to ab_pass_out (ba_pass_out). Therefore, they are both linked to $pass\_out$. The meaning is: the AT set contains the time in the future when the linked event will be triggered. Of course, other events can fill the AT set of another event. In a very natural way, we let the time progress in a non-deterministic way betwen $time{+}1$ and the first AT. If there is no AT then time progression is not limited. When time reaches an activation time, the linked event can be triggered and we remove the AT from the set in the related event. With this model, we are sure that time is progressing, and every time constraint events linked to a AT set will also be triggered at the right moment. This model of time has already been described in our article [8].

6.1 Invariant

The typing of the new variables are ($time \in \mathbb{N}$) and ($at\_a\_pass \subseteq \mathbb{N}$) and ($at\_b\_pass \subseteq \mathbb{N}$). We use a discrete time. As the system is symmetric we only show invariants concerning the device a. The time can not go beyond an activation time, as we don't want to miss the timeout of an event:

$$at\_a\_pass \cup at\_b\_pass \neq \varnothing \Rightarrow time \leq \min(at\_a\_pass \cup at\_b\_pass)$$

As at_a_pass represents the time of the reception of a signal, and signals take the propagation time $prop$ to progress in the channel, then this AT set is bound by $time + prop$:

$$\forall x \cdot (x \in at\_a\_pass \Rightarrow x \leq time + prop)$$

The set at_a_pass is finite and its cardinality reflects the number of signal changes travelling in the channel. In the computer model we use a formula with quantification instead of a cardinality because it is more convenient for the interactive proof.

$$b\_in = ba \wedge ba = a\_out \Leftrightarrow at\_a\_pass = \varnothing$$
$$a\_in = ab \wedge ab \neq b\_out \Leftrightarrow card(at\_a\_pass) = 1$$
$$b\_in \neq ba \wedge ba \neq a\_out \Leftrightarrow card(at\_a\_pass) = 2$$

A device can not start to send after the reception of a signal PN so we have:

$$\forall (x,y) \cdot \begin{pmatrix} x \in at\_a\_pass \wedge y \in at\_b\_pass \\ \Rightarrow |x - y| < prop \end{pmatrix}$$

If cardinality of at_b_pass is two, then the difference of members are strictly under $prop$ because they are both bound by $time{+}prop$ and because the $pass$ events have higher priority than $sleep$ events.

$$\forall (x,y) \cdot \begin{pmatrix} x \in at\_b\_pass \wedge y \in at\_b\_pass \\ \Rightarrow |x - y| < prop \end{pmatrix}$$

The time continues to progress after a sending and if contention is reached we have:

$$b\_in = PN \wedge b\_out = PN \Rightarrow time + prop \notin at\_a\_pass$$

The events $pass$ have higher priority than the events $send$:

$$time \in at\_a\_pass \cup at\_b\_pass \Rightarrow time + prop \notin at\_a\_pass \cup at\_b\_pass$$

This part allows us to prove the refinement of the event ab_pass_out:

$$ab{\neq}b\_out \wedge time \in at\_b\_pass - at\_a\_pass \Rightarrow b\_state{\neq}sending \vee b\_out = IDL$$

6.2 Events

In this refinement we have a new event $tick$. This event makes the time progress. In almost all guards, we add an extra clause to model the priority between events. The main reason for the use of priorities is the fact that the environment must act before the devices react. Otherwise, the behaviour is not always consistent. Here, the environment is the three $pass$ events, so we add $time \notin (at\_a\_pass \cup at\_b\_pass)$ in guards to let $pass$ events trigger before the other. Of course, between the three $pass$ events, the simultaneous passing event $pass\_out$ has higher priority. Finally, the real-time constraints model the propagation time of signals. In the next description of events, we show only the differences between events of the previous refinement. For that we mark new lines with a $\oplus$ and removed lines with a $\ominus$. All new lines of guards are connected with "$\wedge$" and lines of substitution with "$\|$".

init $\widehat{=}$
BEGIN
$\oplus time := 0$

$\oplus at\_a\_pass, at\_b\_pass := \varnothing, \varnothing$
END;

a_send $\widehat{=}$
WHEN
$\oplus time \notin (at\_a\_pass \cup at\_b\_pass)$
THEN
$\oplus at\_b\_pass := at\_b\_pass \cup \{time + prop\}$
END;

b_send $\widehat{=} \ldots$

ab_pass_out $\widehat{=}$
WHEN
$\ominus(b\_state \neq sending \vee b\_out \neq PN)$
$\oplus time \in at\_b\_pass - at\_a\_pass$
THEN
$\oplus at\_b\_pass := at\_b\_pass - \{time\}$
END;

ba_pass_out $\widehat{=} \ldots$

pass_out $\widehat{=}$
WHEN
$\oplus time \in at\_a\_pass \cap at\_b\_pass$
THEN
$\oplus at\_a\_pass := at\_a\_pass - \{time\}$
$\oplus at\_b\_pass := at\_b\_pass - \{time\}$
END;

a_accept $\widehat{=}$
WHEN
$\oplus time \notin (at\_a\_pass \cup at\_b\_pass)$
THEN
$\ldots$
END;

b_accept $\widehat{=} \ldots$

a_sleep $\widehat{=}$
WHERE
$\oplus time \notin (at\_a\_pass \cup at\_b\_pass)$
THEN
$\oplus at\_b\_pass := at\_b\_pass \cup \{time + prop\}$
END;

b_sleep $\widehat{=} \ldots$

a_awake_send $\widehat{=}$
WHEN
$\oplus time \notin (at\_a\_pass \cup at\_b\_pass)$
THEN
$\oplus at\_b\_pass := at\_b\_pass \cup \{time + prop\}$
END;

b_awake_send $\widehat{=} \ldots$

a_awake_accept $\widehat{=}$
WHEN
$\oplus time \notin (at\_a\_pass \cup at\_b\_pass) \wedge$
THEN
$\ldots$
END;

b_awake_accept $\widehat{=} \ldots$

tick $\widehat{=}$
ANY tm WHERE
$tm \in \mathbb{N} \wedge$
$tm > time \wedge$
$((at\_a\_pass \cup at\_b\_pass) \neq \varnothing$
$\Rightarrow tm \leq \min(at\_a\_pass \cup at\_b\_pass)) \wedge$
$(a\_state \neq sending \vee a\_out \neq PN) \wedge$
$(b\_state \neq sending \vee b\_out \neq PN)$
THEN
$time := tm$
END;

7 Third Refinement

This final refinement removes all abstract conditions in the guards and adds the two different sleep times. As we have already explained devices try to go out of the situation of contention by waiting a random time between a short and a long time. So we have two new constants st (short time) and lt (long time) both in $\mathbb{N}$ and non-zero. Their values have the two following properties: $st \geq prop \times 2$ and $lt \geq prop \times 2 + st - 1$. Properties are chosen in order to leave enough time for the devices to react. The whole invariant of this paper is a proof of that. In this refinement we have four new variables: at_a_awake, at_b_awake, a_sleept, b_sleept. Variable at_a_awake (respectively at_b_awake) are a AT set linked to the events a_awake_send and a_awake_accept (b_awake_send and b_awake_accept). Variables a_sleept and b_sleept contain the chosen sleep time. We can see in the Fig. 4 a timeline showing a typical situation of contention with values $prop = 3$, $st = 6$ and $lt = 11$. The election will succeed if devices chose two different delays between st and lt. In that case, we can see in the Fig. 4 that the difference between the two awake times of devices will be enough to transmit a signal. The invariant discussed in the following section will express this formally.

7.1 Invariant

Both new AT sets at_a_awake and at_b_awake are a subset of $\mathbb{N}$. Values of variables a_sleept and b_sleept are in $\{st, lt\}$. We have the same property as in the previous invariant about $time$ and values of AT set: $time$ can not go after the first timeout. The new AT sets contain zero or one value:

$$a\_state \neq sleeping \Leftrightarrow at\_a\_awake = \varnothing$$

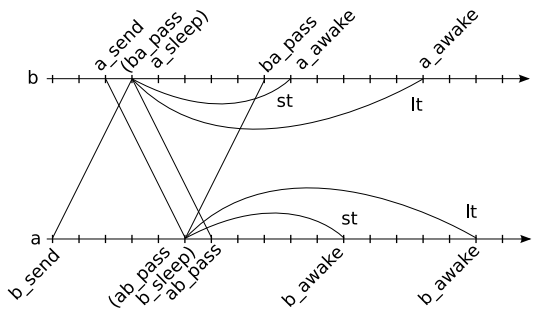

Fig. 4 Timeline of the system with $prop = 3$, $st = 6$ and $lt = 11$

$$a\_state = sleeping \Leftrightarrow card(at\_a\_awake) = 1$$

We have a general upper bound for the new AT set and a special case:

$$\forall x \cdot (x \in at\_a\_awake \Rightarrow x \leq time + a\_sleept)$$

$$time \in at\_a\_awake \wedge a\_sleept = b\_sleept \Rightarrow \forall x \cdot (x \in at\_b\_awake \Rightarrow x < time + prop)$$

And we have several kinds of lower bound with conditions:

$$a\_in = PN \wedge \begin{pmatrix} a\_out = PN \\ \vee (a\_out = IDL \wedge ba = PN) \end{pmatrix} \Rightarrow \forall x \cdot \begin{pmatrix} x \in at\_b\_awake \\ \Rightarrow time + b\_sleept - prop < x \end{pmatrix}$$

$$a\_in = PN \wedge a\_out = PN \Rightarrow \forall x \cdot (x \in at\_b\_awake \Rightarrow time + prop < x)$$

$$case = \text{FALSE} \wedge b\_state = sending \Rightarrow \forall x \cdot (x \in at\_a\_awake \Rightarrow time + prop < x)$$

$$time \in at\_a\_awake \wedge a\_sleept \neq b\_sleept \Rightarrow \forall x \cdot (x \in at\_b\_awake \Rightarrow time + prop \leq x)$$

This invariant restricts the possible values of at_a_awake:

$$ab = PN \wedge b\_out = IDL \Rightarrow time + prop \notin at\_a\_awake$$

Here we can see that, in two different cases, the signal is received before awake time:

$$case = \text{FALSE} \Rightarrow \forall (x, y) \cdot \begin{pmatrix} (x \in at\_a\_pass \wedge y \in at\_a\_awake \\ \Rightarrow x < y) \end{pmatrix}$$

$$a\_sleept \neq b\_sleept \Rightarrow \forall (x, y) \cdot \begin{pmatrix} x \in at\_a\_pass \wedge y \in at\_a\_awake \\ \Rightarrow x \leq y \end{pmatrix}$$

In the following conditions the awake time is before signal reception:

$$case = \text{TRUE} \wedge a\_sleept = b\_sleept \Rightarrow \forall (x, y) \cdot \begin{pmatrix} x \in at\_a\_awake \wedge y \in at\_a\_pass \\ \Rightarrow x < y \end{pmatrix}$$

After the discovering of the contention, device a erases its signal at a propagation time before awake time:

$$\forall (x, y) \cdot \begin{pmatrix} x \in at\_a\_pass \wedge y \in at\_b\_awake \\ \Rightarrow x + prop \leq y \end{pmatrix}$$

If chosen delays are equal, then devices do not have the time to transmit a signal:

$$a\_sleept = b\_sleept \Rightarrow \forall (x, y) \cdot \begin{pmatrix} x \in at\_a\_awake \wedge y \in at\_b\_awake \\ \Rightarrow |x - y| < prop \end{pmatrix}$$

If chosen delays are different, then devices have the time to transmit a signal. This formula shows why this algorithm works when chosen delays are different.

$$a\_sleept \neq b\_sleept \Rightarrow \forall (x, y) \cdot \begin{pmatrix} x \in at\_a\_awake \wedge y \in at\_b\_awake \\ \Rightarrow prop \leq |x - y| \end{pmatrix}$$

If the cardinality of at_a_pass is two then we have $b\_sleept - prop$ between awake time and the reception time of the first signal.

$$b\_in \neq ba \wedge ba \neq a\_out \Rightarrow \forall x \cdot \begin{pmatrix} x \in at\_b\_awake \\ \Rightarrow \min(at\_a\_pass) + \\ b\_sleept - prop < x \end{pmatrix}$$

Finally, these formulae ensure the refinement of events a_awake_send and a_awake_accept:

$$time \in at\_a\_awake \Rightarrow ab = IDL \wedge b\_out = IDL$$

$$time \in at\_a\_awake \wedge (a\_out = PN \vee ba = PN) \Rightarrow b\_state = sending$$

$$case = FALSE \wedge time \in at\_a\_awake \Rightarrow b\_state = sleeping$$

7.2 Events

Again we show only the difference, we mark new lines with a $\oplus$ and removed lines with a $\ominus$. If an event is not present then it does not have any differences or it is symmetric.

With the real-time properties of awake events we can remove all abstract conditions in guard of these events. The properties, expressed with the new AT sets, ensure this requirement as we can see in the last part of the invariant.

init $\widehat{=}$
BEGIN
$\oplus at\_a\_awake, at\_b\_awake := \varnothing, \varnothing$
$\oplus a\_sleept, b\_sleept := st, st$
END;

a_sleep $\widehat{=}$
ANY $\oplus sleep$
WHERE $\oplus sleep \in \{st, lt\}$
THEN $\oplus at\_a\_awake :=$
$at\_a\_awake \cup \{time + sleep\}$
$\oplus a\_sleept := sleep$
END;

b_sleep $\widehat{=} \ldots$

a_awake_send $\widehat{=}$
  WHEN
    $\oplus time \in at\_a\_awake$
    $\ominus a\_state = sleeping$
    $\ominus ab = IDL$
    $\ominus b\_outs = IDL$
  THEN
    $\oplus at\_a\_awake := at\_a\_awake - \{time\}$
  END;

b_awake_send $\widehat{=} \ldots$

a_awake_accept $\widehat{=}$
  WHEN
    $\oplus time \in at\_a\_awake$
    $\ominus a\_state = sleeping$
    $\ominus b\_state = sending$
    $\ominus ab = IDL$
    $\ominus b\_out = IDL$
  THEN
    $\oplus at\_a\_awake := at\_a\_awake - \{time\}$
  END;

b_awake_accept $\widehat{=} \ldots$

tick $\widehat{=} \ldots$
  ANY
    tm
  WHERE
    tm $\in \mathbb{N} \wedge$
    tm $>$ time $\wedge$

$$\left(\begin{pmatrix} at\_a\_pass \cup at\_a\_awake \cup \\ at\_b\_pass \cup at\_b\_awake \end{pmatrix} \neq \varnothing \right.$$
$$\left. \Rightarrow tm \leq \min \begin{pmatrix} at\_a\_pass \cup \\ at\_a\_awake \cup \\ at\_b\_pass \cup \\ at\_b\_awake \end{pmatrix} \right) \wedge$$

    (a_state $\neq$ sending $\vee$ a_out $\neq$ PN) $\wedge$
    (b_state $\neq$ sending $\vee$ b_out $\neq$ PN)
  THEN
    time := tm
  END

8 Verification by Proof and Model-checking

For us, the primary way of verification is done by mechanical proof. But we also have used model-checking with ProB ([13]) in order to make partial verifications and to find counter-examples. In order to have a finite number of transitions with our models, we can not let the variable *time* increase indefinitely. Therefore, we define another version for the event *tick* for model-checking. This version of the event *tick* lets the variable *time* always be zero but decreases the values inside the AT sets. For instance, the **tick** event from the second refinement becomes:

tick $\widehat{=}$
  ANY $shift$ WHERE
    $shift \in \mathbb{N} \wedge$
    $0 < shift \wedge$

$$\left(\begin{array}{l} (at\_a\_pass \cup at\_b\_pass) \neq \varnothing \\ \quad \Rightarrow shift \leq \min(at\_a\_pass \cup at\_b\_pass) \end{array} \right) \wedge$$

    $(a\_state \neq sending \vee a\_out \neq PN) \wedge$
    $(b\_state \neq sending \vee b\_out \neq PN)$
  THEN
    at_a_pass := { x| x+shift$\in$ at_a_pass}
    at_b_pass := { x| x+shift$\in$ at_b_pass}
  END;

prop	reachable states
1	25
2	51
3	81
4	117
5	159
6	207

Fig. 5 Number of reachable states: m2

prop	*st*	*lt*	reachable states
1	2	3	54
2	4	7	186
3	6	11	376
4	8	15	624
5	10	19	930
6	12	23	1294

Fig. 6 Number of reachable states: m3

With ProB it is not possible to do parametric model-checking, therefore we need to give a value (which verifies all hypotheses) to *prop*, *st*, and *lt*. With the new **tick** and those valuations, the number of transitions of the system is finite

Therefore we were able to check all models (invariant included) with ProB. Fig 5 and 6 give the reachable number of states for the second refinement model m2 and the third refinement m3, and the used valuation of constants. The first model and first refinement are trivially checked with 4 states and 24 states respectively.

But as *time* always equals zero and constants are valued, an invariant, determined to be correct with the model-checker, will not always be correct for the proof. In any case, model-checking provides a convenient way to discover invariants and to test them before the proof.

With the B tool the proof is cut into small "Proof Obligations" (PO). Some of those PO are automatically discharged and some need user interactions. All proofs of the first model are done automatically; for the first refinement 59 PO are interactive; for the second refinement 124; and for the last refinement 222. Of course all the proofs are done with the tool of the B method.

The first refinement is very easy to prove, the proofs of second and third refinement are short but numerous.

As the model is symmetric between the two devices a and b, there are two similar versions of each event and of each invariant. Therefore the proof is also symmetric. For instance: the invariant ($a\_state = sending \Leftrightarrow a\_in = PN$) need to be verified for the events **send_a** and **send_b** and again for the invariant ($b\_state = sending \Leftrightarrow b\_in = PN$) with **send_a** and **send_b**. Thus we have 4 proofs which may be very repetitive or totally symmetric. All versions are taken into consideration inside the number of PO.

The procedure of proof, with an invariant, leads to the discovery of an invariant strong enough to be inductive. We can always start with a small invariant containing types of variables and some requirements for the refinements of data or events. If we start to do the proof with an invariant that is too weak then the proof will fail with an impossible interactive proof. With this failure we see the missing piece of information about the system state: we add it to the invariant and retry to prove.

In the case of real-time systems, we can split PO in two parts: PO coming from the event *tick*; and PO coming from all other events. The first kind of PO requires the invariant to be inductive with the progression of the time. Every invariant which contains the time variable *now* are non-trivial to prove for the event *tick* because this event increments *now*. The second requires the invariant to be inductive with the transitions of devices states, as usual for B models. This leads to different kinds of proof and different constraints over the invariant but the two invariants are dependant. Thanks to the encoding of the time properties in the language of the B method, the proof was done with the normal tools and ways of the B method.

9 Related-works and Conclusion

Many of other works about IEEE Root Contention Protocol (RCP) use model-checking over timed automata [6]. We can find a comparative study of works about RCP in [14], this work extends thoses results by another approach. Our approach of verification is primary focused on proof by invariant. It's clear that an interactive proof takes more time than verification by model-checker or proof with decision procedures. This is the price to pay for an expressive general language based on set theory. But the tool cuts the verification proof in small and quite easy parts. And we plan to work on the rules of the tool in order to reduce the number of interactive steps, as the proofs show a repetitive scheme a lot of improvements can be done. The idea of using a variable to model the time is shown in [1], authors call this: explicit-time specification. Here the model of time (AT sets) is different. They focus on worst-case upper and lower bounds on real-time delays. An absolute timer, which can be a lower-bound or a upper-bound, imposes timing bound on actions. A volatile δ-timer counts how much time an action has been continuously enabled. A persistent δ-timer is the same as a volatile δ-timer without the continuous condition over the activation.

For the (classical) B method (which is the "parent" of the Event-B method used here) an extension using duration calculus is described in the thesis [10] and in the article [11]. Our work here is different because classical B principally uses operations, which take a certain amount of time to run, whereas an event is an instantaneous action. This work can be link to the time aware system refinement for Action Systems in [15] where actions also take a certain time.

Similarly to our pattern, Dutertre and Sorea in [12] model and verify an explicit-time specification of a distributed algorithm of election, where timing constraints are modeled as a timeout and calendar. The calendar gives the future times of execution of some events, and this notion is close to our pattern for Event-B in [8]. But the authors do not use a non-deterministic "tick" event which makes the time progress. Instead the time goes directly from one event activation to the next event activation. This model prevent the use of clocks which vary continuously and this allows authors to use a symbolic model-checker. In our work, we use mainly a verification by theorem proving and thus our pattern does not need this particular time progression.

Another work can be found in [7] where authors extend the Abstract State Machines method with First Order Timed Logic in order to verify the Root Contention Protocol with a specific decision procedure. In this paper we use a general tool to reason about a specific domain. In fact, in many cases real-time systems arise in specialised areas like distributed computing, for this reason it is a interesting work to study how to extend an existing specification with specific properties like real-time properties.

The work presented in this paper starts with this situation: we can see in [5] a distributed election algorithm, where most of the problem studied does not require time to be taken into account. But the final phase of this algorithm uses timing constraints, so we found it useful to find a practical way to verify those timing aspects using the same formal method: the Event-B method.

With our pattern, one can model a real-time system within the B method. The refinement relation between

models can be used in order to introduce time in a model and can be used again to add real-time properties step by step. If a concrete model with time refines an abstract model we can prove the timing validity of the concrete model. At every step of refinement, we can verify the model and its invariants by model-checking it first and then by a computer-assisted proof. This paper shows how our pattern of refinement of real-time constraints works on the IEEE 1394 RCP. The safety proof of RCP is done and we can see from the last invariant how the election works when chosen delays are different:

If chosen delays are different ($a\_sleept \neq b\_sleept$), then devices have the time to transmit a signal:

$$\forall(x,y)\cdot \begin{pmatrix} x \in at\_a\_awake \wedge y \in at\_b\_awake \\ \Rightarrow prop \leq |x-y| \end{pmatrix}$$

But if chosen delays are equal ($a\_sleept = b\_sleept$), then devices do not have enough time:

$$\forall(x,y)\cdot \begin{pmatrix} x \in at\_a\_awake \wedge y \in at\_b\_awake \\ \Rightarrow |x-y| < prop) \end{pmatrix}$$

Where at_a_awake and at_b_awake are sets of natural and represent the time when devices will stop to wait. And *prop* is the propagation time needed by the signal to pass from one device to another. Notice when devices are not in the state *sleeping* and do not plan to awake then the corresponding set is empty.

Our method can be used without changes to the language of B and therefore we can extend existing results and developments in B with real-time. As the proofs about passing of time are specific, we could consider a way of handling this specificity. Our time is discrete but in our proof the most important property used is the order over natural numbers. The time model used involves a global time which interacts with a number of time of activation stored in several sets as in a global multi-set. As the studied algorithm does not require synchronisation, the global time is not a problem but we can think about localising this into several distributed clocks for other case-studies.

References

1. Abadi, M., Lamport, L.: An old-fashined recipe for real-time. ACM Trans. Program. Lang. Syst. **16**(5), 1543–1571 (1994)
2. Abrial, J.R.: B$^{\#}$: Toward a synthesis between Z and B. In: D. Bert, J.P. Bowen, S. King, M.A. Waldén (eds.) ZB, *Lecture Notes in Computer Science*, vol. 2651, pp. 168–177. Springer (2003)
3. Abrial, J.R., Cansell, D.: Click'n prove: Interactive proofs within set theory. In: D.A. Basin, B. Wolff (eds.) TPHOLs, *Lecture Notes in Computer Science*, vol. 2758, pp. 1–24. Springer (2003)
4. Abrial, J.R., Cansell, D., Méry, D.: A mechanically proved and incremental development of IEEE 1394 tree identify protocol. Formal Asp. Comput. **14**(3), 215–227 (2003)
5. Abrial, J.R., Cansell, D., Méry, D.: A mechanically proved and incremental development of ieee 1394 tree identify protocol. Formal Asp. Comput. **14**(3), 215–227 (2003)
6. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
7. Beauquier, D., Crolard, T., Prokofieva, E.: Automatic parametric verification of a root contention protocol based on abstract state machines and first order timed logic. In: K. Jensen, A. Podelski (eds.) TACAS, *Lecture Notes in Computer Science*, vol. 2988, pp. 372–387. Springer (2004)
8. Cansell, D., Mèry, D., Rehm, J.: Time constraint patterns for event B development. In: B 2007: Formal Specification and Development in B, vol. 4355/2006, pp. 140–154. Springer (2007)
9. ClearSy, Aix-en-Provence (F): B4FREE (2004). Http://www.b4free.com
10. Colin, S.: Contribution à l'intégration de temporalité au formalisme b : Utilisation du calcul des durées en tant que sémantique temporelle pour b. Ph.D. thesis, Universitée de Valenciennes et du Hainaut-Cambrésis (2006)
11. Colin, S., Mariano, G., Poirriez, V.: Duration calculus: A real-time semantic for b. In: Z. Liu, K. Araki (eds.) ICTAC, *Lecture Notes in Computer Science*, vol. 3407, pp. 431–446. Springer (2004)
12. Dutertre, B., Sorea, M.: Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In: Y. Lakhnech, S. Yovine (eds.) FORMATS/FTRTFT, *Lecture Notes in Computer Science*, vol. 3253, pp. 199–214. Springer (2004)
13. Leuschel, M., Butler, M.J.: Prob: A model checker for b. In: K. Araki, S. Gnesi, D. Mandrioli (eds.) FME, *Lecture Notes in Computer Science*, vol. 2805, pp. 855–874. Springer (2003)
14. Stoelinga, M.: Fun with FireWire: Experiments with verifying the ieee1394 root contention protocol. In: J.R. S. Maharaj C. Shankland (ed.) Formal Aspects of Computing (2002)
15. Westerlund, T., Plosila, J.: Time aware system refinement. Electr. Notes Theor. Comput. Sci. **187**, 91–106 (2007)