Time Constraint Patterns for Event B Development

Dominique Cansell, Dominique Méry, Joris Rehm

To cite this version:

Time constraint patterns for event B development

Dominique Cansell[1,5], Dominique Méry[2,3,5], and Joris Rehm[2,4,5]

[1] Université de Metz
cansell@loria.fr
[2] Université Henri Poincaré Nancy 1
[3] mery@loria.fr
[4] rehm@loria.fr
[5] LORIA
BP 239
54506 Vandœuvre-lès-Nancy
France

August 13, 2008

Abstract Distributed applications are based on algorithms which should be able to deal with time constraints. It is mandatory to express time constraints in (mathematical) models and the current work intends to integrate time constraints in the modelling process based on event B models and refinement. The starting point of our work is the event B development of the IEEE 1394 leader election protocol; from standard documents, we derive temporal requirements to solve the contention problem and we propose a method for introducing time constraints using a pattern. The pattern captures time constraints in a generic event B development and it is applied to the IEEE 1394 case study.

Key-words: event B, pattern, distributed systems, refinement.

1 Introduction

1.1 Overview

In this article, we present work in progress on the modelling of time in event B using *patterns*. The concept of time is not predefined in B but using set theory we can effectively model it. Generally most formal models "implement" time in the first abstraction and they explicitly express time constraints in models of computations or automata; several notations propose solutions for expressing time and time constraints (timed automata of Alur and Dill [8]). We think that it is not a good idea to introduce time too early in the development process, because invariants on time (using natural numbers and arithmetic) introduce too much noise for proof assistants and consequently most proof obligations need interaction. Using abstraction without time we can solve and prove more easily important properties of the system and obtain a first scheduling of events. For us, time can be introduced later in a specific refinement where time is a global variable.

1.2 Motivations for integrating time constraints in event B development

Needs for integrating time constraints in event B development are motivated by observations on a case study developed by Jean-Raymond Abrial, D. Cansell and D. Méry [6] and the goal was to redevelop the leader election protocol and to provide a proof simpler and easier to understand than Devillers et al [16]. The IEEE 1394 leader election protocol works properly provided the network is acyclic; but it is sensitive to time constraints [20], since it may loop forever if no time constraint is taken into account. In fact, the problem appears when the algorithm is executed and leads to a situation where two nodes of a finite network a and b are in contention. Let us recall the problem using events. Node a sends a message to b and asks him to be its child, while node b is asking to a to be its child. The two nodes a and b have sent messages to each other and the messages were sent independently. No one wants to be the leader but no other one can be the leader, since they are the last nodes in the election process: the others nodes have already asked to be children or grand-children of a or b. This problem is called the *contention* problem. This problem can occur with only two nodes of the network (this can be proved using our models). In the "real" protocol the problem is "solved" by means of timers. As soon as a node a discovers a contention with node b, it waits for a very short delay in order to be sure that the other node b has also discovered the problem. The very short delay in question is at least equal to the message transfer time between nodes (such a time is supposed to be *bounded*). After this, each node randomly chooses (with probability 1/2) to wait for either a "short" or a "large" delay (the difference between the two is at least twice the message transfer time). After the chosen delay has passed each node sends a new request to the other *if it is in the situation to do so*. Clearly, if both nodes choose the same delay, the contention situation will reappear. However if they do not choose the same delay, then the one with the largest delay becomes the father of the other: when it wakes up, it discovers the request from the other while it has not itself already sent its own request, it can therefore send an acknowledgement and thus become the father. According to the *law of large numbers*, the probability for both nodes indefinitely choosing the same delay is zero. Thus, at some point, they will (in probability) choose different delays and one of them will thus becomes the father of the other.

Abrial et al. [6] present a partial formalisation of the contention problem and the idea is to introduce a *virtual channel* which is used to resolve the contention. Recently, J-R. Abrial et al propose a simpler (without acknowledgement and confirmation) algorithm [7]. In both models, the contention is solved abstractly and no time reference is used. The real algorithm uses time constraints to solve this contention. Our starting questions were:

- can we add time constraints in previous abstract models to facilitate more realistic refinement?
- can we do this in a systematic way using something similar to design patterns in object-oriented software development [18]?

The questions will be partly addressed in the next sections. However, the introduction of time is not the main issue and the next sub-section motivates the introduction of patterns in the B methodology.

1.3 B patterns

Designing models of a complex system using refinement and proofs are very hard and often not very well used. This proof-based modelling technique is not automatically done, since we need to extract some knowledge from the system in an incremental and clever way. The event B method allows one such development and many case studies have been developed, including sequential algorithms [5], distributed algorithms [6,14], parallel algorithms [4] or embedded systems for controlling train like METEOR [10] or Roissy Val [9]. The last example was developed faster because previous models were reused and a specific automatic refining tool - Edit B developed by Matra (now Siemens)- was utilised. EditB provides automatic refinement from an abstract B model, which can be proved more quickly and automatically using or adding specific rules in the B prover; EditB is a "private" tool and only Siemens uses it to develop systems. The interesting thing is that the engineer activity *(typing model)* is very much simplified. This tool seems to apply a similar technique to those used in design patterns. It is the application of well-engineered patterns for a specific domain.

Three years ago Jean-Raymond Abrial, D. Cansell and D. Méry worked on using genericity in event B [13,3]. When a designer develops a generic development (a list of models related by refinement) both modelling and proof are easily done. Models are more abstract and consequently the set of proof obligations can be discharged more automatically or in an interactive way (it is less noisy for the prover). The generic development can be reused by instantiation of the carrier sets and constants of the development (list of models). We obtain a new development (list of models) which is correct, if all instantiated sets and constants satisfy properties of the generic development. An interesting point is that we do not prove the proof obligation of the instantiated development. This technique is well known by mathematicians who prove abstract theorems and reuse these on specific cases reproving the instantiated axioms of the theorem to obtain (for free or without proof) the instantiated goal of the theorem.

Recently, Jean-Raymond Abrial has presented [2] patterns for the action/-reaction paradigm to systematically develop the mechanical press controller.

These contributions follows the same direction leading to reuse previous proof-based developments, to give guidelines for mechanical refinement in daily software development. In our opinion, a B pattern is an event B development which is proved, sufficiently generic and can be partially reused in another specific B development to produce automatically new refinement models: proofs are (partly) inherited from the B pattern.

1.4 Summary

Our paper proposes initial and partial answers to these questions. We do not give an exact definition of B patterns. It is too early to propose a standard definition as many works are converging to this B pattern concept. We describe a pattern with regard to time and how we can use it to produce other patterns or to solve a specific problem. The next section introduces the time constraint pattern and its construction. Section 3 presents an application of our pattern for a message passing system. Section 4 concludes the paper by the IEEE 1394 case study and future works.

2 Time constraint pattern

In order to express time and time constraints we introduce a new pattern. This pattern demonstrates our modelling choice and gives a general background to reason about things like time progression, clock or timer. The main idea is to guard events with a time constraint, therefore those events can be observed only when the system reaches a specific time. The time progress is also an event, so there is no modification of the underlying language of B. It is only a modelling technique instead of a specialised formal system. The variable *time* is in $\mathbb{N}$ but time constraints can be written in terms involving unknown constants or expressions between different times. Finally, the timed event observations can be constrained by other events which determine future activations.

2.1 Defining the pattern

We can explain our method through an example event-B model. Later this model can be used like a pattern to refine another model adding time considerations. As you can see below, the pattern has two variables:

```
MODEL
  $m0$
VARIABLES
  $time$, /* current time */
  $at$ /* Active Times */
INVARIANT
  $time \in \mathbb{N} \quad \wedge$
  $at \subseteq \mathbb{N} \quad \wedge$
  $(at \neq \emptyset \implies time \leq \mathsf{min}(at))$
INITIALISATION
  $time := 0 \mid\mid at := \emptyset$
EVENTS
...
```

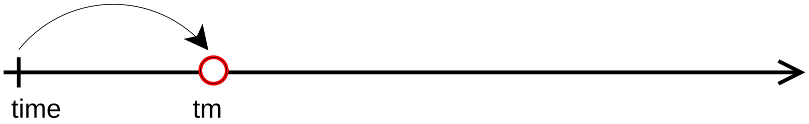
- *time* in $\mathbb{N}$ models the current time value. The incrementation of this value denotes the time progression.

- $at \subseteq \mathbb{N}$ is the known future active times of the system. Each active time stands for a future event activation. For example, a simple clock will have a set of active times like $\{time+1, time+2, ...\}$.

Since this pattern is very general, the invariant is simple and we have only to satisfy $at \neq \emptyset \Rightarrow time \leq min(at)$. This means that active times are in the future. As a consequence of this fact the time can not be moved beyond the first active time, this is intuitively correct because if time goes beyond one event activation, then we miss the right moment for observing it.

The three events represent three different temporal aspects. The first event is the creation of a new active time. In real system this can be the initialisation of a timer or the setting of a new time constraint. We denote this by "posting" new active times in the event "post_time". This event is needed when the activation of the system is dynamic. For our example of a regular clock the active times are known for every system so we have only to initialise the set *at* with ℕ. In this case, the event *post_time* is not required. For more complex cases like message passing in a network, the active times are determined by the message arrival so we need an event like *post_time* observed when a message is sent to constrain the system to receiving them some time later.

```
post_time  $\widehat{=}$
  any  $tm$  where
    $tm \in \mathbb{N} \quad \wedge$
    $tm > time$
  then
    $at := at \ \cup \ \{tm\}$
  end
```

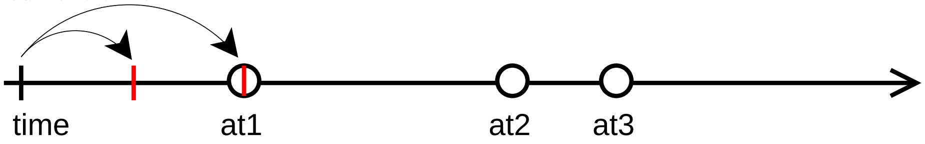


The event takes a new active time *tm* which is indeterminate in the most general case but it can be more specific like $time + delay$ with a constant *delay* in ℕ and greater than 0.

The second aspect is time progression. In this modelling approach, in a system state the time is frozen and it can go with an observation of the tick_tock event.

```
tick_tock  $\widehat{=}$
  any  $tm$  where
    $tm \in \mathbb{N}$  $\wedge$
    $tm > time$  $\wedge$
    $(at \neq \emptyset \implies tm \leq \mathsf{min}(at))$
  then
    $time := tm$
  end
```

This event simply takes a new value of time in the future and assigns it at the current time.

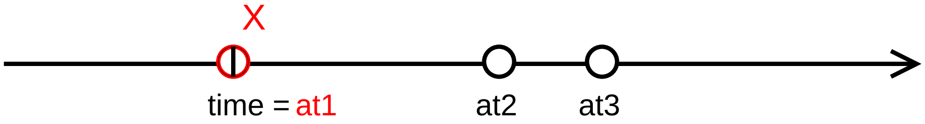


As we have already said, time progress is nondeterministic, the new value tm should only satisfy the invariant with $tm \leq min(at)$, if $at \neq \emptyset$. Otherwise time can take place everywhere and let the system trigger any event potentially. But as time is a natural and $tm > time$ we are sure that the system will reach the next active time if *tick_tock* is activated enough. Thanks to the set at, which is very general, this event can be copied without modification when we use the time constraint pattern.

Now we can look at the last aspect which is the goal of our work. With this event "process_time" we can consider events with time constraints.

```
process_time  $\widehat{=}$
  when
    $time \in at$
  then
    $at := at - \{time\}$
  end
```

The guard $time \in at$ and the invariant implies that $time$ has reached the first active time. The time can have made one or more step with one or more activation on *tick_tock* and other temporised events may have occurred.



The current active time is deleted from at therefore an active time can be used once and only once. After this removal the time and the system can continue to change.

2.2 Applying the time constraint pattern

This model can be used as a pattern, but it is very general and the invariant is limited. The pattern can be fit to time-sensitive systems in order to introduce time behaviour and prove invariants. Consequently such B patterns can be used as a systematic help to refine systems.

As this pattern represents a way to write time arguments it can not be used directly but needs to be adapted to a specific system (except for *tick_tock* which can be used directly).

At first, events of the system involving time must be present and written in a proved model. The idea is to use refinement and make an abstract model where time is implicitly controlled by events as usual. One can already reason about a model without time and prove general or abstract properties on the system. Next, the pattern should be adapted except *tick_tock*. The two aspects involved in *post_time* and *process_time* need to be identified from the modelled system. For instance, the beginning of a timer, sending of message or other initiation of non-instantaneous actions match the posting time event. Connected to that timer ending, message reception or finalisation of non-instantaneous actions match the processing time event. The result will be a set of adapted and renamed instances of *post_time* and *process_time* events.

When this aspect of events has been identified we can use it to refine the abstract model with a superposition of modified events; we refine the abstract model; for instance, we can substitute an abstract guard model by concrete time expression; in this way, we can prove that the time constraints implements required behaviour.

Some specific adaptation or improvement may be used:

- The two ideas of posting and processing active times have been presented separately for getting the essence of the concept. However they are often mixed in the same event. One can make a chain of reactions with time constraints between events.
- There is no contradiction to consider more than one time posting in an event which refines *post_time*. We can add a number of active times in one shot. Using the same idea, an infinite number of times can be posted using generalised substitution: for example, on the initialisation of the system.
- In addition to the set *at* we can add variables to express in a more specialised and meaningful way active times. These variables have common elements with *at*. For example, we can store sending time of messages. All these added variables allow one to write more specialised time constraints and give different categories of active times inside *at*.
- With the last remark, we can have different categories but we can not simply trigger more than one event at the same time because *at* is a set and it can not contain several identical values like in a multiset. To resolve this we can take several sets like *at* or take a function to index different sets. This index will represent different processes which can run at the same time. Of course this modification needs to be done in the same way for the rest of the model:

invariant and other events. With this adaptation we can represent different local clocks or several processes.

2.3 Comparisons with other methods

Our solution does not require a language with explicit time expression. Consequently it is difficult to compare this work with other solutions. A big part of other work uses timed automata [8] with model checkers such as Uppaal [11] or Kronos [15]. These automata allows one to write transition systems with time. Transitions between states are instantaneous and time can progress inside a state. One can use several clocks (variables in $\mathbb{R}$). Time constraints are comparisons of clocks with numeric constants and can be set both on state and transition. Automata can only stay in a state, if clock does not exceed time constraints. In the same way system can transit, if constraints are valid. One can reset clock to zero on transition, so the time may be cyclic.

We can point out fundamental differences with our time model using connections between event B models and automata. In our model it is a transition (event) which makes time progress under some condition instead of a state such that time can grow under a limit. For this reason we can have several event activations which are instantaneous. Usually we do not reset time to zero because we can infinitely add active times in the future.

The main difference comes with the use of the active times set which are not explicit in timed automata theory. The word "clock" does not fit very well with our approach because the variable *time* denotes the general time passing. For us a clock is a relation between time progression and known future active times. This set is also the main difference with the explicit-time description in [1] and [19] by L. Lamport et al. As a result our tick_tock event is more general because it only refers to elements of this set.

Properties certification for timed automata is done by a model-checker. The infinite number of states (because of time) is reduced to a finite set of partitions over vector space make by clocks. In our case, proof are made as usual with first order classical logic and set theory inside B tools.

Some other works related to real-time systems can be found in [17] by C.J. Fidge and A.J. Wellings, their approach is different than ours because they do not use instantaneous actions.

3 Message passing using the time constraint pattern

This section presents an application of our pattern. We design a system of two devices a and b. Device a can send a message to b. We prove that a timer triggered after sending ensure to a the effective reception of message by b (we do not take into account loss of message). The system is described by two models. The first model has basic elements and events sequencing. The second model refines the previous sequencing of events by time constraints.

3.1 Abstract model

As a first step we introduce the problem with an abstract model. The model consists of two constants a and b for the devices, four variables A, S, B and AB, and three events :

- sendA : a sends its message to b using connection AB
- recB : b receives it from the connection AB
- quA : when a knows that the message is received by b, it modifies one of its local variable S.

The invariant of the model is:

$$\boxed{\begin{array}{l} A \subseteq \{a\} \quad \wedge \\ B \subseteq \{b\} \quad \wedge \\ AB \subseteq \{a\} \quad \wedge \\ S \subseteq \{a\} \quad \wedge \\ (A \neq \emptyset \implies AB \neq \emptyset) \end{array}}$$

According to a distributed system, we consider that A and S are local variables for device a, B is a local variable for device b and AB is a global variable for the channel between the two devices. Similarly events sendA and quA are local to device a and event recB is local to device b. A denotes sending of the message if and only if A is not empty, similarly B denotes its reception and S denotes the state after execution of quA. All variables are booleans (empty or not). Next we define the three events:

```
sendA $\widehat{=}$
  when
    $A = \emptyset$
  then
    $A := \{a\} \parallel AB := \{a\}$
  end
```

```
recB $\widehat{=}$
  when
    $AB = \{a\}$
  then
    $B := \{b\} \parallel AB := \emptyset$
  end
```

Using A to express the sending of the message and B its reception, these events implicitly denote a delay between the execution of sendA and recB. After this delay, we make an action (the reception of the message) as a requirement. To specify this we explicitly ask the message to be received, in the guard of quA.

```
quA $\widehat{=}$
  when
    $B = \{b\}$
  then
    $S := \{a\}$
  end
```

In the abstract model we are "cheating" because event *quaA* is intended to be local to device a but it can see the variable B which is intended to be local to device b. It is as if device a can see local information of device b. In order to enforce the localisation property as we are moving towards implementation we refine this specification with the time constraint pattern.

3.2 Introducing time by the time constraint pattern

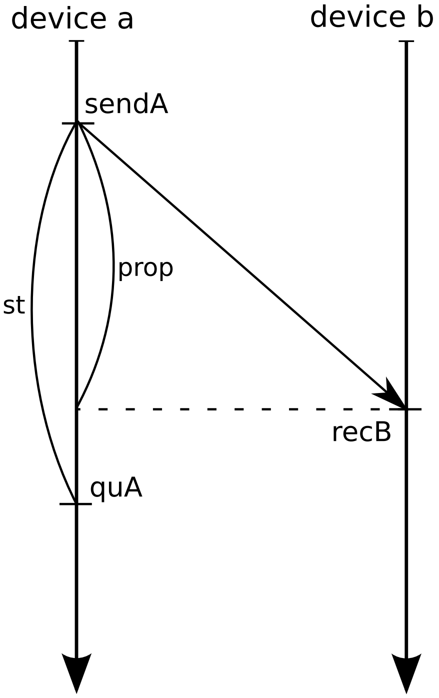
With the method already described we need to adapt the pattern and anchor events in time. For this we need two constants : *prop* is the propagation time needed for the message to transit from *a* to *b* and *st* is the sleeping time used in the timer. As an adaptation we need two new variables: *stm* is the "send time message" and *slp* the time when *a* will stop sleeping at the end of the timer. The tick_tock event can be copied as before and we have the two aspects of the pattern:

- Only event sendA posts two active times : $time + prop$ and $time + st$ (with the hypothesis $prop < st$).
- Events recB and quA are processing an active time.

Next we can write temporal aspects with a refinement of the abstract model.

```
sendA $\widehat{=}$
when
   $A = \emptyset$
then
   $A := \{a\}$ ||
   $AB := \{a\}$ ||
   $at := at \cup \{time + prop\} \cup$
      $\{time + st\}$ || /* added */
   $stm := time$ || /* added */
   $slp := time + st$ /* added */
end
```

```
recB $\widehat{=}$
when
   $AB = \{a\} \wedge$
   $time = stm + prop$ /* added */
then
   $B := \{b\}$ ||
   $AB := \emptyset$ ||
   $at := at - \{time\}$ /* added */
end
```



For these two events the refinement is just a superposition, i.e. some lines have been added without changing existing expressions. On sendA we can see the two new active times $time + prop$ and $time + st$ which are the future arrival time of messages and the awake time ending the timer. On the same event the informative variables *stm* and *slp* are set up. The second event is triggered by $stm + prop$ which is equal to the posted value $time + prop$, this active time is deleted from *at*, as in the pattern. Now the most interesting event:

```
quA $\widehat{=}$
when
  $A \neq \emptyset \ \wedge \ / * \mathit{changed\ to\ a\ local\ guard} * /$
  $\mathit{time} = \mathit{slp} \ / * \mathit{added} * /$
then
  $S := \{a\} \ ||$
  $\mathit{at} := \mathit{at} - \{\mathit{time}\} \ / * \mathit{added} * /$
end
```

Here the refinement is not just a superposition: the abstract guard was $B = \{b\}$ and is changed to a more concrete $A \neq \emptyset \wedge \mathit{time} = \mathit{slp}$. The use of the non local variable B has disappeared with the use of the local variable A and of the variable *time*. Variable *time* is universal and global so we can use it to get more information from the local state of distributed devices. In order to prove the refinement we need the following invariant:

$$\begin{array}{l}
\mathit{time} \in \mathbb{N} \quad \wedge \\
\mathit{stm} \in \mathbb{N} \quad \wedge \\
\mathit{slp} \in \mathbb{N} \quad \wedge \\
\mathit{at} \subseteq \mathbb{N} \quad \wedge \\
(A \neq \emptyset \implies \mathit{stm} + \mathit{prop} < \mathit{slp}) \quad \wedge \\
(A \neq \emptyset \wedge \mathit{time} \geq \mathit{stm} + \mathit{prop} \wedge \mathit{stm} + \mathit{prop} \notin \mathit{at} \implies B = \{b\}) \quad \wedge \\
(\mathit{at} \neq \emptyset \implies \mathit{time} \leq \mathit{min}(\mathit{at})) \quad \wedge \\
\mathit{at} \subseteq \{\mathit{stm} + \mathit{prop}, \mathit{slp}\} \quad \wedge \\
(A = \emptyset \implies \mathit{at} = \emptyset) \quad \wedge \\
(A \neq \emptyset \wedge \mathit{at} = \emptyset \implies \mathit{time} \geq \mathit{slp}) \quad \wedge \\
(A \neq \emptyset \wedge \mathit{at} \neq \emptyset \implies \mathit{slp} \in \mathit{at}) \quad \wedge \\
(A \neq \emptyset \wedge \mathit{at} = \{\mathit{slp}\} \implies \mathit{time} \geq \mathit{stm} + \mathit{prop})
\end{array}$$

We give explanations on the most interesting part of this invariant and a derived theorem:

- $(A \neq \emptyset \wedge \mathit{stm} + \mathit{prop} \notin \mathit{at} \wedge \mathit{time} \geq \mathit{stm} + \mathit{prop} \Rightarrow B = \{b\}$:
  This part of the invariant is important to prove the refinement of quA. In this expression if time is beyond $\mathit{stm}+\mathit{prop}$ and if the time constraint $\mathit{stm}+\mathit{prop}$ has already been processed then we are sure of the reception ($B = \{b\}$).
- $(A \neq \emptyset \wedge \mathit{at} = \{\mathit{slp}\} \Rightarrow \mathit{time} \geq \mathit{stm} + \mathit{prop}$:
  If active times set is only $\{\mathit{slp}\}$ and message is gone then current time is after the message reception.
- $(A \neq \emptyset \wedge \mathit{at} = \emptyset \Rightarrow \mathit{time} \geq \mathit{slp}$:
  This predicate is interesting if the message has already been sent ($A \neq \emptyset$) and if there is no more time constraints on process ($\mathit{at} = \emptyset$), in other words once all the events were observed. In this case, one can affirm that the current time exceeded the moment when a was awaken.

- $(A \neq \emptyset \implies stm + prop < slp)$:
  This invariant uses the fact that $prop < st$ because event *sendA* provides the following proof obligation: $\{a\} \neq \emptyset \implies time + prop < time + st$. This fact is a property on constants *st* and *prop* which expresses that the propagation time is less than the sleep time.

The abstract event quA is cheating, since it looks the variable B in its guard ($B = \{b\}$); the refined version is no more cheating, since the guard is local ($A \neq \emptyset$ it has sent the message) and the temporal guard $time = slp$. Only *time* is a global variable shared by each participant of the global system: it is local for each participant and everyone has the same time. We assume that the time is the same of everyone.

4 Concluding remarks and perspectives

4.1 On the contention problem

This work began with the time constraint problems inside the firewire protocol. The protocol, namely *IEEE 1394*, can be found in computers and devices like cameras or external hard-disks and is used to connect them together in a local network. When one or several devices are connected or disconnected, they are able to reconfigure themselves. The reconfiguration consists of the election of a network leader. The network is a symmetric acyclic graph, the algorithm of election orients edges to obtain a spanning tree rooted by a leader.

At each step of this distributed algorithm a device is submitted to another. The submitted device is a leaf node among non-submitted devices. The submission of the device is done by sending a message to the device next to it.

But, at the end of the process, a contention problem may occur with the last two devices (and only in this case). Since both devices are leaves, they can send submission almost at the same time. In this case, the two messages cross in the bi-directional channel between the last devices. The election can not be done because both devices are in a submission state.

We can see in the figure 1 an example of contention. X and Y devices are sending messages together with the arrow “1”. After the first sending there is a period “a”, in this period the other node can send a second message because the first is not received. The protocol has a special case for this problem, the chosen solution deletes the two submission messages and tries to choose a leader with a new message. Messages are not structured packets but constant signals, so a message can be put on or removed from the channel. To give a chance for resolving contention each device chooses a delay between a long and a short time. Then they sleep for the chosen delay. We can see a example below in figure 2 with the two delays “b” and “c” and the deletion of message with the arrow “2”.

When the device awakes, there are two possibilities:

- No message has arrived during the sleeping time, so the device can send a submission message.

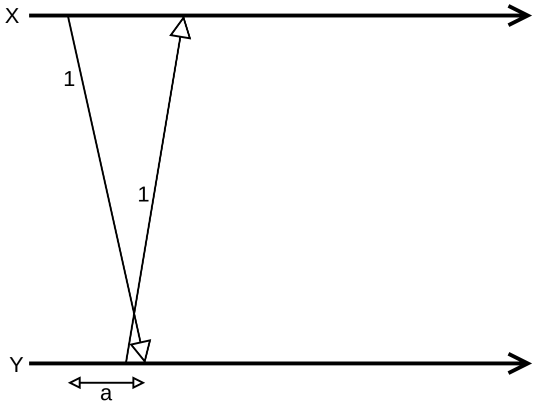


Figure 1. Example for the contention problem

- One message has arrived during the sleep, so the device has to receive it and becomes the leader.

This contention solving succeeds only if the chosen times are different, in this case only one message is re-sent and the receiver becomes the leader. We can see this discriminatory message labelled “3” in the figure 2. On the other hand if sleeping times are the same the contention problem occurs again and the same process begin.

As we have seen there are a lot of time issues in this part of the protocol. Time is needed to quantify the two different sleeping times and to represent the progression of the message signals over the channel.

4.2 Conclusion

The starting point of our work is the proof-based development of the IEEE 1394 leader election protocol and the observation of the partiality of the resulting proved solution [6]: the development does not take into account time constraints. Moreover, we paid attention to capture our modelling experience in a pattern called *time constraint pattern*. We give a light definition of patterns which are planned to give a systematic help for specialised refinement. Our work illustrates the use of a general and global time which interacts with a number of actives times. The time progression is abstract and nondeterministic; the concept of active times can be fit to various situations like simple clocks, timers or delays for messages. We have used our pattern on a realistic problem involving messages on channels and we have studied time constraints in contention problem of the IEEE 1394 protocol.

The contention problem is not yet completely solved. But we have enough elements and tools to solve the contention problem in the event B framework. With the help of refinement we can introduce time constraints that satisfies

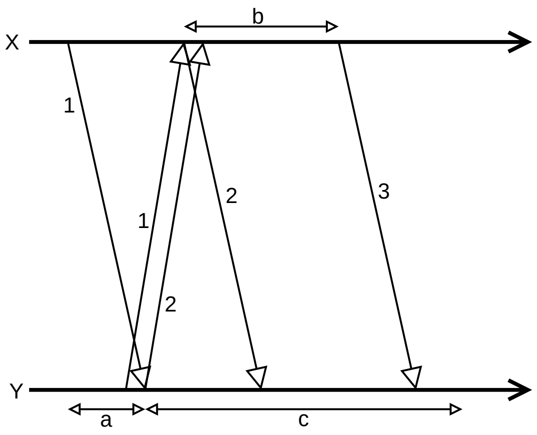


Figure 2. Example for the contention problem

a sequence of events. If a concrete system refines another system with time constraints we can prove the timing validity of the concrete system. For future work, we can enrich the library of patterns and we can study the applicability of such a process on others case studies.

References

1. Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.
2. J.-R. Abrial. Using Design Patterns in Formal Devlopments - Example: A Mechanical Press Controler. Journée scientifique du PPF IAEM Transversal - Développement incrémental et prouvé de systèmes, April 2006.
3. Jean-Raymond Abrial. B$^{\#}$: Toward a synthesis between z and b. In Bert et al. [12], pages 168–177.
4. Jean-Raymond Abrial and Dominique Cansell. Formal Construction of a Non-blocking Concurrent Queue Algorithm (a Case Study in Atomicity). *Journal of Universal Computer Science*, 11(5):744–770, May 2005.
5. Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Formal derivation of spanning trees algorithms. In Bert et al. [12], pages 457–476.
6. Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of IEEE 1394 tree identify protocol. *Formal Asp. Comput.*, 14(3):215–227, 2003.
7. Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A new IEEE 1394 leader election protocol. In *Rigorous Methods for Software Construction and Analysis Seminar N 06191,07.05.-12.05.06.* Schloss Dagstuhl, U. Glaser and J. Abrial, 2006.
8. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

9. Frédéric Badeau and Arnaud Amelot. Using B as a high level programming language in an industrial project: Roissy val. In Helen Treharne, Steve King, Martin C. Henson, and Steve A. Schneider, editors, *ZB*, volume 3455 of *Lecture Notes in Computer Science*, pages 334–354. Springer, 2005.
10. P. Behm, P. Benoit, A. Faivre, and J.-M.Meynadier. METEOR : A successful application of B in a large project. In *Proceedings of FM'99: World Congress on Formal Methods*, Lecture Notes in Computer Science, pages 369–387, 1999.
11. Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, pages 232–243, 1995.
12. Didier Bert, Jonathan P. Bowen, Steve King, and Marina A. Waldén, editors. *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, volume 2651 of *Lecture Notes in Computer Science*. Springer, 2003.
13. D. Cansell. *Assistance au développement incrémental et à sa preuve*. Habilitation à diriger des recherches, Université Henri Poincaré (Nancy 1), 2003.
14. D. Cansell and D. Méry. Formal and Incremental Construction of Distributed Algorithms:
On the Distributed Reference Counting Algorithm. *Theoretical Computer Science*, 2006. to appear.
15. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III: Verification and Control*, volume 1066, pages 208–219, Rutgers University, New Brunswick, NJ, USA, 22–25 October 1995. Springer.
16. Marco Devillers, W. O. David Griffioen, Judi Romijn, and Frits W. Vaandrager. Verification of a leader election protocol: Formal methods applied to ieee 1394. *Formal Methods in System Design*, 16(3):307–320, 2000.
17. Colin J. Fidge and Andy J. Wellings. An action-based formal model for concurrent real-time systems. *Formal Aspects of Computing*, 9(2):175–207, 1997.
18. Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In Oscar Nierstrasz, editor, *ECOOP*, volume 707 of *Lecture Notes in Computer Science*, pages 406–431. Springer, 1993.
19. L. Lamport. Real time is really simple. Technical Report MSR-TR-2006-30, March 2005.
20. Judi Romijn. A timed verification of the ieee 1394 leader election protocol. *Formal Methods in System Design*, 19(2):165–194, 2001.