Pattern Based Integration of Time applied to the 2-Slots Simpson Algorithm

Joris Rehm

Pattern Based Integration of Time applied to the 2-Slots Simpson Algorithm*

Joris Rehm

joris.rehm@loria.fr
LORIA - Université Henri Poincaré Nancy 1
BP 239 - 54506 Vandœuvre-lès-Nancy - France

Abstract. Event-B is a formal method used to do model driven engineering correct by construction. We propose a pattern to integrate time in this method. This pattern integrates elements from the theory of timed automata and event-clock automata. As experimentation of our ideas, we present a case study: an algorithm for asynchronous communication from H.R. Simpson. We prove this formal development with the software tool Rodin.

1 Introduction

Our goal in this work is to use formal methods with systems that have real-time aspects. The formal method that we wish to use is "Event-B". As Event-B does not explicitly handle real-time problems, we propose a pattern to handle time properties. This pattern integrates elements from the theory of timed automata [2] and event-clock automata [3].

In this paper, we present a case study to illustrate our ideas. This case study is a formal development of an algorithm for asynchronous communication. The algorithm that we use is a version of Simpsons algorithm [9] where two memory slots are used rather than four. This version with two slots is not fully asynchronous, but it requires less space memory than the full version.

The full version with four slots has also been studied [1] using the Event-B Method (no time properties are needed for the four slots algorithm, those two algorithms are in fact very different). The first and second models of our case study are equivalent to the first models of this case study [1]. Those two models describe the communication scheme independently of the algorithm therefore it is an example of reusing a model. After that, the next refinements model the algorithm itself and consequently are different. The common elements from [1] are not, at this time, publicly available; we show its in this paper.

As this two slots version of the algorithm is not fully synchronous, some behaviours for the communicating processes are not permitted. We use real time constraints to specify these restrictions. We use our pattern to specify some time properties on the system formed by this algorithm. With this specification,

* This work was supported by grant No. ANR-06-SETI-015-03 awarded by the Agence Nationale de la Recherche. Many thanks to Rosemary Monahan for the help.

we have verified that this 2-slots version of Simpsons algorithm with real-time constraints is correct.

The purpose of the algorithm is to allow a one-way asynchronous communication between two entities. As the communication is directional, we name one of the entities the “writer” and the other one the “reader”. Furthermore, the direction of communication goes from the writer towards the reader. At any time, the writer can send a new value, and the reader may or may not obtain it, in an (almost) independent way. This is implemented with variables (a memory) shared between both entities.

For example, we can imagine that the writer is an electronic thermometer that regularly updates the temperature and that the reader is another device that reads the current value of the temperature when it needs.

As usual in event-B development, we have a chain of models which refine each other. The first model is the most abstract specification. Here, we consider two atomic events named *read* and *write*. Note that, in the implementation and in the last model, the operations of reading and writing are not atomic (the size of the communicated data is not limited). To represent this characteristic in the first model, there will be a gap between the value that is read and the value that is written. However, the algorithm gives guarantees about the level of freshness of the read value and we have formalised the value of the gap. Informally, we can say that the read value is at least as recent as the last value written at the time of the previous reading.

The objective of this algorithm is to avoid writing two values in succession during the same reading. This would provoke two actions on the same slot of the (2 slots) memory, which is undesirable.

The sketch of the proof (and thus of the proved development) is: “the interval between successive writes is greater than the duration of any read” $\Rightarrow$ “do not write twice in a row during the same reading” $\Rightarrow$ “no memory access problem”. Every step of this sketch are represented by a refinement. We start by the basic specification “we want a communication algorithm” then we add the concept of memory, and finally we add the real-time issue.

In this case study, we applied our pattern in order to obtain the duration between the start and the end of the reading and writing operations.

In our previous works, we studied a leader election protocol [7], for that we used a pattern of calendar [4] for the time model. In the short paper [6] we sketched a preliminary study of the 2-slots algorithm. The study is here fully completed and described, in addition we changed the time model (in fact, the complexity of the former is not required by the study). Between the three papers [4, 6] and this one, the model of the time (the pattern used) are different. And we think that the pattern of this paper is more adequate for the Simpson algorithm.

About the 2-Slots Simpson Algorithm, in addition to the original description [9], we can find [5] which studies the feasibility conditions for scheduling and utilisation of this method of communication. The paper [8] gives an extension of this algorithm for preemptive scheduling.

This paper carry on in Section 2 by introducing the issues found in the studied algorithm. We continue in Section 3 with the description of our pattern for the time model. In Section 4 shows the formal development of the case study. Finally we conclude in Section 5.

2 Presentation of the Simpson Algorithm

In figure 1 we see an example of traces of values that are written (wv) and values that are read (rv). In this example, the reader supplies a new different value each time ($a, b, c, ...$). The reader can choose the same value during consecutive choices e.g. $rv(1) = rv(2) = a$, if no new write is available. It is also possible for the reader to miss some values, as $wv(3) = c$ which does not appear in rv. With

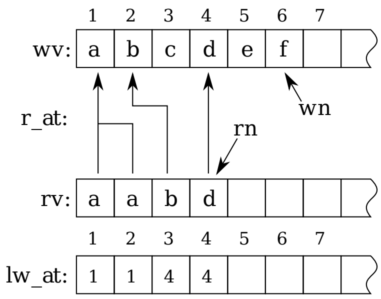

Fig. 1. Traces of reading and writing

r_at (Read AT) we can see what the link between the read and written values is. In the figure the written values are all different but this is not generally the case therefore we need the function r_at. We trace the value of wn (the number of writes) at each read using lw_at (Last Write AT). Using this information it is possible to quantify the lag of the reader. For example $rv(3)$ reads the value $wv(2)$ but the latest is $wv(4)$ (in fact $wv(lw\_at(3))$) which is only read at $rv(4)$. With these variables, we will show two important properties in the first model: "the order of the read values is the same than the written values" and "the freshness of the read value is guaranteed at some point". Those properties will be kept in the complete development through relating each model by refinement.

The two-slot asynchronous communication mechanism from Simpson [9] can be seen as the following pseudo-code:

```
read(){                      write(d){
 readi:=latest;               buffer(1-latest):=d;
 rr:=buffer(readi);           latest:=1-latest;
}                            }
```

where $buffer$ is the two-slot memory which is a function from $0, 1$ to $DATA$. We can see a graphical representation in Fig. 2. The write operation takes a value d (from the set $DATA$), writes it in the buffer and switches the value that is stored in the variable called *latest*. This variable *latest* has its value in $\{0, 1\}$, therefore $1 - latest$ switches the value from 0 (respectively 1) to 1 (resp 0). The read operation stores the value of *latest* and reads the corresponding value from the buffer to *rr* (the Read Result). The difficulty of the mechanism comes from the size of the elements of $DATA$ which can be arbitrary large. This means that the duration for reading or writing can be very long. On the contrary *latest* (and *readi*) has a small, fixed size and we can manipulate it value atomically. The read and write operations run in parallel. However, as the algorithm is not allowed to read and write concurrently at the same place in a memory, we need at least two different slots. While the reader uses one slot, the system can use the other one to write the new data. When new data is written the pointer *latest* is updated to point to the up-to-date slot of memory.

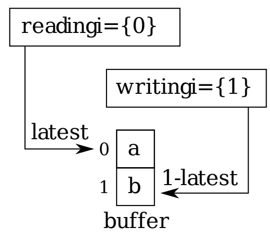

Fig. 2. Two-slots memory

Actually, two slots are not enough: if, while the reader copies the value from $buffer$ to *rr*, the system performs two writes, then it reads and writes at the some slot ($1 - latest$ becomes equal to *readi*). To handle this scenario: [9] has proposed two ways: add more slots (four slots for a full asynchronism) or keep two-slots and add real-time constraints. For that, [9] gives a condition: "the interval between successive writes is always greater than the duration of any read".

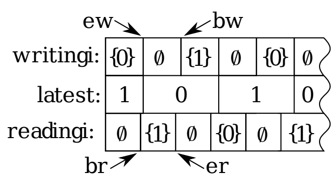

Fig. 3. Time line of the behaviour

This condition is illustrated in Fig. 3. The set *writingi* represents the memory slot currently written. And the set *readingi* represents the slot currently read. As before, the value *latest* denotes where the freshest value is in memory. This value is updated at the end of each write, and the reader uses this value to choose where to read. For the correct use of memory, the set *writingi* and *readingi* must be disjoint (at the same moment). In the worst case, the beginning of the read (denoted by *br* on the picture) starts just before the end of a write (*ew*). In this case, if the duration between *br* and *er* (End Read) is longer than the duration between *ew* and *bw* (Begin Write) then the system reads and writes at the same slot (the figure shows a correct behaviour).

3 Pattern

To represent these scheduling properties we need to model the time inside of our language and method. For that, we use clocks similar to clocks in timed automata [2]. We now show the pattern that we use in the final model of the Simpson mechanism. In this model, we have four clocks. In the pattern we use the function S which associates every element of the set E to a value in $\mathbb{N}$.

Sets E **Variables** S **Invariants** inv1:$S \in E \rightarrow \mathbb{N}$	**Event** init $\widehat{=}$ **begin** act1:$S := \{e \mapsto 0 \mid e \in E\}$ **end**

Initially, all the clocks are set to zero.

Event reset $\widehat{=}$ **any** e **where** grd1:$e \in E$ **then** act1:$S(e) := 0$ **end**	**Event** tic $\widehat{=}$ **any** s **where** grd1:$0 < s$ **then** act1: $S :=$ $\{e \cdot e \in E \mid e \mapsto S(e) + s\}$ **end**

Only two actions can be applied to the clock. We can reset one clock $S(e)$ to zero with the event *reset*. And the event *tic* can increment all the clocks by a positive non-null value s (Shift). In our models, we take $s = 1$ to simplify the proofs. This model is a pattern of our final model in the sense that the set of clocks will refine this behaviour.

But these clock will be used in a precise way. In fact, E will be a subset of actual events of a model. For an event e, the clock $S(e)$ (Since e) records the duration between the last execution of the event e and the present time in the system. Such a mechanism is similar to event-recording automata [3].

The clock can appear in the invariant, and we can also use it the guard of event. Therefore, we can express a lower or a upper time bound by adding inequality in the guard of an event. More interesting, we can represent a mandatory upper time bound by adding:

$$Guard(e) \Rightarrow S(e) + s \leq x$$

in the guard of the event *tic*, where e is a event; $Guard(e)$ its guard without time bound; and x expression of type $\mathbb{N}$ (the $\leq$ can also be $<$).

4 Two-Slots Asynchronous Communication Mechanism

This section is organised as a sequence of refined models. Each subsection shows a model which refines the previous (except for the first). Every model will focus on a particular subject or aspect of the system.

4.1 Specification of the Asynchronous Writer and Reader by Traces

The goal of this model is to specify the *read* and *write* events which manipulate elements of the *DATA* set. For that, we model the sequence of written values ($wv \in 1 \,..\, wn \rightarrow DATA$ with $wn \in \mathbb{N}_1$ the writes number) and similarly the sequence of read values ($rv \in 1 \,..\, rn \rightarrow DATA$ with $rn \in \mathbb{N}_1$ the reads number).

Of course, the read and written values are the same and the variable r_at ("Read AT" $r\_at \in 1 \,..\, rn \rightarrow 1 \,..\, wn$) gives the connection from the i^{th} read value to the $r\_at(i)^{th}$ written value, as we will see in the invariant.

Finally, the reader will try to access the most up-to-date written value, but this is not always possible. The model specifies this shift by recording, at each i^{th} read, the latest index of write (which is wn) in the function lw_at ("Last Write AT" $lw\_at \in 1 \,..\, rn \rightarrow 1 \,..\, wn$).

We can now deduce the two following events:

Event read $\widehat{=}$ **any** ri **where** grd1: $ri \in lw\_at(rn) \,..\, wn$ **then** act1: $rn := rn + 1$ act2: $r\_at(rn + 1) := ri$ act3: $lw\_at(rn + 1) := wn$ act4: $rv(rn + 1) := wv(ri)$ **end**	**Event** write $\widehat{=}$ **any** d **where** grd1: $d \in DATA$ **then** act1: $wn := wn + 1$ act2: $wv(wn + 1) := d$ **end**

Most of actions and the guard of the event *write* are self-explanatory but $grd1$ of event *read* is not. This guard expresses the obligation of the reader to be up-to-date. This means that at each read, the variable ri (Read Index) must be greater than the last written value known at the time of the last read ($lw\_at(rn)$). In other words, the read index is always incremented.

From this we can prove some properties using the invariant. As we already said, we have a relation between the read and written values. This relation is

$$rv = r\_at; wv$$

This tells us that the reader actually processes the written value rather than random values.

As we said, the reader can be "in late":

$$\forall i \cdot i \in 1 \,..\, rn \Rightarrow r\_at(i) \leq lw\_at(i)$$

But we know that the read value is as fresh as the last value written at the time of the previous reading

$$\forall i \cdot i \in 1 \,..\, rn - 1 \Rightarrow lw\_at(i) \leq r\_at(i+1)$$

This first model gives a general specification for a mechanism of asynchronous communication between the reader and writer. As we verified the basic properties of these communications, we removed some general variables which are not needed to work on the incoming issues. Fortunately, thanks to the refinement relation, all those properties still hold. Of course those properties are expressed on the abstract variables and the following models will use different concrete variables. But the "gluing" invariant between the abstract and the concrete variables ensures the transition of the abstract properties to the whole set of refined models.

4.2 Removing the Reader Trace

As we refine, we keep the variables wn and wv from the first model. The variables rn, r_at, lw_at and rv disappear, for the benefit of the new variables rr ("Read Result" $rr \in DATA$) and lw_at_lr ("Last Write AT Last Read" $lw\_at\_lr \in 1 \,..\, wn$). The variable rr represents the result of the *read* event:

$$rr = rv(rn)$$

and similarly we only need the last value of the sequence lw_at:

$$lw\_at\_lr = lw\_at(rn)$$

As you can see in the following event, this new set of variables is enough to express the system behaviour. While this changes the event *read*, the *write* event remains the same.

Event read $\widehat{=}$
any ri **where**
  grd1: $ri \in lw\_at\_lr \,..\, wn$
then
  act1: $lw\_at\_lr := wn$
  act2: $rr := wv(ri)$
end

We are now ready to introduce some parts of the algorithm.

4.3 The 2-Slots Memory: First Elements

In this refinement, the variable lw_at_lr disappears. The variable was used in the specification but now some part of the specification can now be fulfiled with the two new variables $reading$ ($reading \subseteq \mathbb{N}$) and $writing$ ($writing \subseteq \mathbb{N}$). We also add events in order to replace an atomic event of reading or writing by two events for each operation: one event for the beginning and one event for the end of the operation. The variable $reading$ gives the index of the values which are currently read, and the variable $writing$ gives the values currently written.

The event end_read refines $read$, and the event end_write refines $write$:

Event begin_read $\widehat{=}$ **when** grd1: $reading = \varnothing$ **then** act1: $reading := \{wn\}$ **end**	**Event** end_read $\widehat{=}$ **any** ri **where** grd1: $ri \in reading$ **then** act1: $rr := wv(ri)$ act2: $reading := \varnothing$ **end**

Event begin_write $\widehat{=}$ **when** grd1: $writing = \varnothing$ grd2: $reading \neq \varnothing \Rightarrow wn \in reading$ **then** act1: $writing := \{wn+1\}$ **end**	**Event** end_write $\widehat{=}$ **any** d, wi **where** grd1: $d \in DATA$ grd2: $wi \in writing$ **then** act1: $wn := wi$ act2: $wv(wi) := d$ act3: $writing := \varnothing$ **end**

The guard $grd2$ of the event $begin\_write$ needs explanation. In fact, the key of this version of the algorithm is not to write a value twice while doing one read, as we will see in the next refinements, the 2 slots memory is not able to handle this situation. Therefore, this guard can be read as: if a reading is running then the value currently read must be wn. This allows one write, afterwhich we have $reading = \{wn-1\}$ thus preventing another write from occuring. As soon as the reading is finished, the writer can act again so we can also do several readings or several writings.

The invariant will clarify this behaviour. We can see that the writer can only add the number $wn+1$ or be inactive:

$$writing \subseteq \{wn+1\}$$

For the reader, we have three possibilities: no reading; read the latest value wn; or read the value before the last ($wn-1$).

$$\exists x \cdot x \in \{wn, wn-1\} \wedge reading \subseteq \{x\}$$

While reading, the reader can only read the latest written value:

$$writing \neq \varnothing \Rightarrow reading \subseteq \{wn\}$$

To prove the refinement of guard $grd1$ of $read$ ($ri \in lw\_at\_lr..wn$) in the previous model, we need to know that $lw\_at\_lr \leq wn - 1$ if we read the value $wn - 1$ (which equals to ri):

$$wn - 1 \in reading \Rightarrow lw\_at\_lr \leq wn - 1$$

We have now modelled the main point of this algorithm: the constraints over the asynchronous behaviour.

4.4 The Actual 2-Slots Memory

Now, we can add more implementation elements, like the 2 slots memory. To do this we replace wv by function $buffer$ ($buffer \in \{0,1\} \rightarrow DATA$)which represents the memory. We also need the variable $latest$ ($latest \in \{0,1\}$) to store the location of the slot of the $buffer$ with the latest value. Finally, we replace $reading$ (respectively $writing$) by $readingi \subseteq \{0,1\}$ (resp. $writingi \subseteq \{0,1\}$) which stores the index of the memory instead of the index in terms of the number of $read$ (resp. $write$) events. As we will see at the end of the invariant, the main goal of the model is to show that the memory is correctly used (e.g. no read and write events on the same slot).

Event begin_read $\widehat{=}$ **when** grd1: $readingi = \varnothing$ **then** act1: $readingi := \{latest\}$ **end**	**Event** end_read $\widehat{=}$ **any** i **where** grd1: $i \in readingi$ **with** ri: $(i = latest \Rightarrow ri = wn) \wedge$ $(i \neq latest \Rightarrow ri = wn - 1)$ **then** act1: $rr := buffer(i)$ act2: $readingi := \varnothing$ **end**

Event begin_write $\widehat{=}$ **when** grd1: $writingi = \varnothing$ grd2: $readingi \neq \varnothing \Rightarrow$ $readingi = \{latest\}$ **then** act1: $writingi := \{1 - latest\}$ **end**	**Event** end_write $\widehat{=}$ **any** d, i **where** grd1: $d \in DATA$ grd2: $i \in writingi$ **with** wi: $wi = wn + 1$ **then** act1: $writingi := \varnothing$ act2: $buffer(i) := d$ act3: $latest := i$ **end**

With this version of the model, the reader uses the slot $latest$ in the $buffer$ while the writer uses the slot $1 - latest$ in the $buffer$ (which is the other one between the two possible slots). As you can see in $act3$ of end_write, updating the variable $latest$ is only done at the end of the writing, because the reader can access the updated slot as soon as variable $latest$ is changed..

We can see the witness of in the clause **with** of end_read. This witness defines how the variable ri of the abstract event (with the same name) is refined by the variables of this concrete event.

Now the invariant must explicitly relate the new (concrete) variables and the old (abstract) variables. The content of the memory $buffer$ is the wn^{th} and the $(wn + 1)^{th}$ written values, and we know which value is which using the index $latest$:

$$buffer(latest) = wv(wn)$$

$$wn \geq 2 \Rightarrow buffer(1 - latest) = wv(wn - 1)$$

The variable $writingi$ is almost equivalent to $writing$ but we know that writing can occur on the slot $latest - 1$ of memory:

$$writingi = \varnothing \Leftrightarrow writing = \varnothing$$

$$writingi = \{1 - latest\} \Leftrightarrow writing = \{wn + 1\}$$

In the same way, we know that the current reading can occur on $latest$ or on $1 - latest$:

$$readingi = \varnothing \Leftrightarrow reading = \varnothing$$

$$readingi = \{latest\} \Leftrightarrow reading = \{wn\}$$

$$readingi = \{1 - latest\} \Leftrightarrow reading = \{wn - 1\}$$

We have proved the theorem that the read or write operation never occurs on the same slot of the memory:

$$readingi \cap writingi = \varnothing$$

This can be deduced using the invariant. As now we have verified the crucial safety properties, we can move a step further towards a concrete model.

4.5 Toward Boolean Variables

In order to simplify the data-types of the model, we can use booleans rather than sets. We replace $readingi$ by two variables: $read$ and $readi$. The variable $read \in BOOL$ is true when $readingi$ is not empty. And, in this case, $readi \in \{0, 1\}$ gives the value inside $readingi$.

The set $writingi$ is replaced by $write \in BOOL$. We do not need another variable to store the value inside $writingi$ because this value is known (always $wn + 1$).

Event begin_read $\widehat{=}$ **when** grd1: $read = FALSE$ **then** act1: $read := TRUE$ act2: $readi := latest$ **end**	**Event** end_read $\widehat{=}$ **when** grd1: $read = TRUE$ **with** i: $i = readi$ **then** act1: $read := FALSE$ act2: $rr := buffer(readi)$ **end**

Event begin_write $\widehat{=}$ **when** grd1: $write = FALSE$ grd2: $read = TRUE \Rightarrow latest = readi$ **then** act1: $write := TRUE$ **end**	**Event** end_write $\widehat{=}$ **any** d **where** grd1: $d \in DATA$ grd2: $write = TRUE$ **with** i: $i = 1 - latest$ **then** act1: $write := FALSE$ act2: $buffer(1 - latest) := d$ act3: $latest := 1 - latest$ **end**

The witness of *end_read* says that the value of the abstract variable i (local to *end_read*) is now denoted by the value in the model variable *readi*. Similarly the variable i of *end_write* (which is another local variable with the same name and not the same variable) is refined by the constant value $1 - latest$.

The invariant of the model "glues" the three concrete variables (*read*, *write* and *readi*) with the abstract variables (*readingi* and *writingi*) which disappear. To express this "gluing invariant" we give the equivalence for the emptiness of the variables *readingi* and *writingi*:

$$read = FALSE \Leftrightarrow readingi = \varnothing$$

$$write = FALSE \Leftrightarrow writingi = \varnothing$$

Then the values for the case of non-emptiness can be easily deduced with the help of the invariant:

$$read = TRUE \Rightarrow readingi = \{readi\}$$

Finally, the proof of refinement, with the help of the witness clauses (part **with**), is trivial.

4.6 Real-time Constraints

When we actually use this algorithm, we do not want the writer to use a variable belonging to the reader (like *readi* in *grd*2 of *begin_write*) to check a running condition. Instead, we want to use real-time constraints to ensure this condition. The model that we present in this section models this requirement by replacing this abstract guard $grd2$ of *begin_write* by an adequate encoding of the real-time properties.

The model of time uses a set of clocks (all of type $\mathbb{N}$) which we call *sbr* (Since Begin Read), *ser* (Since End Read), *sbw* (Since Begin Write) and *sew* (Since End Write). Each clock is associated with an event. For example *sbr* is associated with *begin_read*. In the actions of the associated event, the clock is reset to zero. We want the clock to count how much time is elapsed since the last triggering of the associated event. For that, we also need to make time progress with the event *tic*. This event increments the clocks. Other events are not allowed to make the time progress. We also have a constant $c \in \mathbb{N}_1$.

Event begin_read $\widehat{=}$ **when** grd1: $read = FALSE$ **then** act1: $read := TRUE$ act2: $readi := latest$ act3: $sbr := 0$ **end**	**Event** end_read $\widehat{=}$ **when** grd1: $read = TRUE$ **then** act1: $read := FALSE$ act2: $rr := buffer(readi)$ act3: $ser := 0$ **end**

Event begin_write $\widehat{=}$ **when** grd1: $write = FALSE$ grd2: $c \leq sew$ **then** act1: $write := TRUE$ act2: $sbw := 0$ **end**	**Event** end_write $\widehat{=}$ **any** d **where** grd1: $d \in DATA$ grd2: $write = TRUE$ **then** act1: $write := FALSE$ act2: $buffer(1 - latest) := d$ act3: $latest := 1 - latest$ act4: $sew := 0$ **end**

Event tic $\widehat{=}$
when
grd1: $read = TRUE \Rightarrow sbr + 1 < c$
then
act1: $sbr := sbr + 1$
act2: $ser := ser + 1$
act3: $sbw := sbw + 1$
act4: $sew := sew + 1$
end

In this set of events, two important elements must be considered: the $grd2$ of *begin_write* (which replaces the abstract $grd2$ of the previous model) and the $grd1$ of *tic*. The $grd2$ (of *begin_write*) means the system waits at least c units of time before triggering *begin_write*. We count the time since the last execution of *end_write* (where *write* became $FALSE$) as this lower bound is applied to *sew*. For $grd1$ (of *tic*) we have an upper bound on *sbr* if $read = TRUE$. The progression of time is therefore limited with this condition. In fact the predicate $read = TRUE$ is the guard of *end_read*. This means that *end_read* is forced to happen before *sbr* reach c.

In the invariant we can prove a upper bound on *sbr* and a lower bound on *sew*:

$$read = TRUE \Rightarrow sbr < c$$
$$write = TRUE \Rightarrow c \leq sew$$

This means that the duration between *begin_read* and *end_read* is strictly lower than c, and the duration between *end_write* and *begin_write* is greather than c. The value itself of c does not matter, but it must be greater than zero.

Now in the invariant we must explain how it is possible to replace $grd2$ of *begin_write* from the previous model. Under the condition $read = TRUE \wedge write = FALSE$ and the time constraint that we must have $latest = readi$, we consider the following invariant:

$$sbr < sew \wedge read = TRUE \wedge write = FALSE \Rightarrow latest = readi$$

From the guard of *begin_write* we know $c \leq sew$. This fact, along with the first invariants implies that $sbr < sew$. Hence, we can deduce that the abstract guard of *begin_write* ($read = TRUE \Rightarrow latest = readi$).

Non-blocking In the Event-B method there is a proof obligation of non-blocking. This obligation shows that the system will never block. We prove this by proving the disjunction of the event's guard. Our algorithm describes a perpetual reactive system, we thus verified the theorem:

$read = FALSE \vee read = TRUE \vee (write = FALSE \wedge c \leq sew) \vee write = TRUE \ \vee \ (read = TRUE \Rightarrow sbr + 1 < c)$

It is indeed possible to introduce real-time constraints leading to a blocked state of the system. Therefore the real-time bounds and the guard of *tic* are also included in this verification.

Proof Obligations Details This proved development was conceived on the Rodin[1] software tool (from the European project of the same name) with the prover B4Free of the ClearSy company. All the proof obligations (PO) were cleared. The following table gives the details of the number of proof obligation by models:

Model	Total	Auto	Inter
m0	30	21	9
m1	12	12	0
m2	27	21	6
m3	43	33	10
m4	21	19	2
m5	28	13	15

[1] http://www.event-b.org

The label "Auto" means done without user intervention, and "Inter" means done with an interactive session of proving. We found the interactive proofs quite easy and short.

5 Conclusion and Perspective

In this paper, we proved a model of asynchronous communication. This mechanism comes from [9]. Our proof is structured in a sequence of six models, refining each-other in a proved development

First, a general specification of the properties of this kind of communication between a writer and a reader is provided. This specification uses traces of reading and writing elements to express how the reader can follow or miss the written value, we also express how the reader can be in late regarding the writer (freshness of the read value).

This specification is prepared in our case study by refining the general notation. We then study the properties of the algorithm.

We study how the reader or the writer can interleave and how late the reader can be in this algorithm. In fact the communication is not totally asynchronous (for this version with 2-slot) we formalised this condition of the scheduling.

In the next refinement, we add the 2-slot buffer and show it is safely used. Next to simplify the model, we removed some set-theory notations by replacing them with boolean values.

Finally, we proved that the time constraints correctly implement the scheduling condition and we verified that the system does not contains deadlock.

The reader may wonder why we use so many refinements, the reason is that each refinement is fit to express a particular property. And it would be harder to express and validate all the verified properties in only one big model, although it is possible. For example, the correct use of the memory $readingi \cap writingi = \varnothing$ would be harder to express one level below with the boolean variable *read* and *write*. And with the refinement, we can also verify our models (the invariant and the refinement proof obligations) at each step.

This development, and its proof, was achieved using the Event-B method. In addition, we integrated the theory of event-recording automata [3] with a pattern of refinement. We found that this integration works smoothly.

In future work, we will formalise an augmented version of Event-B models with time-bounds. For example, we can add a lower bound to the event *end_read* and a mandatory upper bound to *begin_write*.

Event end_read $\widehat{=}$	**Event** begin_write $\widehat{=}$
when	**when**
grd1: $read = TRUE$	grd1: $write = FALSE$
time-bounds	**time-bounds**
bnd1: $Since(begin\_read) < c$	bnd2: $c \leq Since(end\_write)$
then	**then**
...	...

Such models should abstract the inner mechanism of our pattern and can be used to generate the models shown in this paper. It should be also possible to export the models for another formal method or tools with a real-time support.

References

1. Jean-Raymond Abrial and Dominique Cansell. Formal development of simpson's 4-slot algorithm. Technical report, Private communication, March 2006.
2. Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
3. Rajeev Alur, Limor Fix, and Thomas A. Henzinger. Event-clock automata: A determinizable class of timed automata. *Theor. Comput. Sci.*, 211(1-2):253–273, 1999.
4. Dominique Cansell, Dominique Mèry, and Joris Rehm. Time constraint patterns for event B development. In *B 2007: Formal Specification and Development in B*, volume 4355/2006, pages 140–154. Springer, January 17-19 2007.
5. J. Chen and A. Burns. Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties. In *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, page 236, Washington, DC, USA, 1999. IEEE Computer Society.
6. Joris Rehm. A Duration Pattern for Event-B Method. In *2nd Junior Researcher Workshop on Real-Time Computing - JRWRTC 2008*, Rennes France, 10 2008. ANR-06-SETI-015.
7. Joris Rehm and Dominique Cansell. Proved Development of the Real-Time Properties of the IEEE 1394 Root Contention Protocol with the Event B Method. In Frederic Boniol Yamine Aït Ameur and Virginie Wiels, editors, *RNTI ISoLA 2007 Workshop On Leveraging Applications of Formal Methods, Verification and Validation*, volume RNTI-SM-1, pages 179–190, Poitiers-Futuroscope France, 12 2007. Cépaduès.
8. Norman Scaife and Paul Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *ECRTS*, pages 119–126. IEEE Computer Society, 2004.
9. H.R. Simpson. Four-slot fully asynchronous communication mechanism. *Computers and Digital Techniques, IEE Proceedings* -, 137(1):17–30, Jan 1990.