

Event Driven Electronic Circuit Construction

by J.-R. Abrial

August 2001

Version 5

Event Driven Electronic Circuit Construction

1. Introduction

In this paper, I shall present a simple methodology supporting the *progressive proved development* of synchronized circuits. For this, I shall use the B Method [1] and, more precisely, its, so-called, *event-driven approach* [2] [3]. Some examples illustrate the presentation: they are entirely mechanically treated with Atelier B, which is the tool associated with B. Another outcome of this paper is the notion of *design by proof*: we shall see how the proofs, done during the design process of our circuits, will definitely influence it.

1.1 Synchronized Circuits

A synchronized circuit is viewed as a *box*, within which an *input* line is entering, and out of which an *output* line is emerging. Of course, there might be more input and output lines.

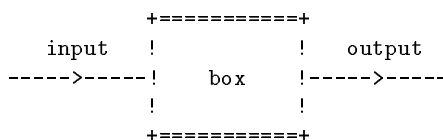


Fig. 1. A typical circuit

The circuit is supposed to be synchronized by a clock, which pulses regularly between two alternative positions: *low* and *high*

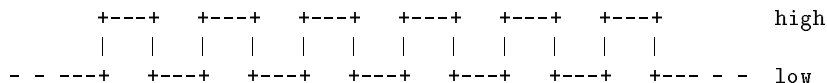


Fig. 2. Clock pulses

Synchronization means this: (1) when the clock is *low*, the *box* and the *output* line are supposed to be idle, only the *input* line may be changed, and conversely (2) when the clock is *high*, the *input* line is supposed to stay idle whereas the *box* may be modified as well as the *output* line. From now on, we consider that the *box* and *output* line together form the *circuit*, whereas the input line constitutes its *environment*. Note that the environment may also comprise some “internal” *state*.

1.2. Coupling the Circuit with its Environment

With this view of circuit and environment in mind, the notion of clock can be made more abstract by simply saying that it gives us two alternative ways of *observing the closed system* made of the circuit and its environment.

We can thus consider that we have two *modes* of observation: one, *env*, corresponds to observing the environment independently from the circuit, and another one, *cir*, consisting in observing the circuit independently from the environment. Such modes alternate for ever. From now on, we shall follow that view and forget about the clock. This has the consequence that we shall never develop a circuit in isolation, but always *together with its environment*. Such a *coupling* is shown in the following schema:

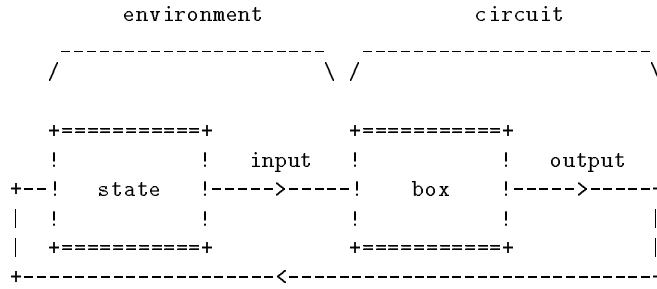


Fig. 3. A circuit coupled with its environment

1.3. Dynamic View of the Coupling

There is an important distinction to be made between the way the *input* line and environmental *state* are changed when the *mode* is *env*, and the *output* line and *box* are changed when the *mode* is *cir*. The modification of the environment may follow some specific rules but in no case is it influenced by the *box* (it may be influenced by the *output* however). Conversely, the change in the circuit may depend on the *input* line and on the *box* (but not on the *state* of the environment however). This can be formalized by the following “non-deterministic assignments”:

$$\begin{array}{l}
 \text{mode} = \text{env} \implies \left(\begin{array}{l} \text{state, input} \in F(\text{output, state}) \parallel \\ \text{mode} := \text{cir} \end{array} \right) \\
 \text{mode} = \text{cir} \implies \left(\begin{array}{l} \text{box, output} \in G(\text{input, box}) \parallel \\ \text{mode} := \text{env} \end{array} \right)
 \end{array}$$

The conditions $\text{mode} = \text{env}$ or $\text{mode} = \text{cir}$ situated on the left hand side of the long arrow “ \implies ” are the *guards* of these assignments: they represent the necessary conditions for the multiple assignments situated on the right hand side of “ \implies ” to be “executed”. The symbol “ \in ” is a non-deterministic generalization of the classical assignment symbol “ $:=$ ”. The symbol “ \parallel ” corresponds to a generalization of multiple assignment. What is formalized here is just “one step” of the alternating behaviors of the environment and of the circuit.

The formula $F(\text{output, state})$ above represents the set of possible next values for the environment (depending on the *output* of the circuit and the *state* of the environment), it specifies the way the *environment* may evolve: it must clearly be totally outside the control of the *box* of the circuit. This assignment is non-deterministic because, most of the time, the environment moves in a way that cannot be known in advance (although not totally erratically). Notice that a possibility is one by which the environment stays *unchanged*. Another possibility is one by which the environment is not influenced by the *output* of the circuit: in this case, the coupling

between the two is rather loose.

Likewise, the formula $G(input, box)$ represents the set of possible next values for the circuit (depending on $input$ and on the previous value of box). It specifies the way the circuit *reacts* to the uncontrolled behavior of its environment and possibly to its own past (represented by its internal status box). Here the assignment might be non-deterministic because, in an abstract view of the circuit, we do not need to make its behavior completely known in all circumstances: we can accept a certain laxism in the specification, leaving the final details of various alternative implementations for a further design phase. Notice that, as for the environment, one of the various possibilities is that the circuit does not change: in this case, it simply is *reactionless* in front of the environment.

Also notice that, in an abstract view of our circuit and environment, the status of the *input* and *output* lines and of *box* and *state* are not necessarily represented by boolean values (which will probably be the case in a refined implementation). For instance, in an abstract specification, *state* can very well carry the entire *history* of what has happened since the interaction between the circuit and its environment has started.

1.4. Static View of the Coupling

So far we have only envisaged a very *operational* (although abstract) view of our circuit and environment: we have just described how these entities behave dynamically while time is passing, but we have not at all explained *why* they should behave like this. Another completely *independant* approach is one by which a *static view* is presented by means of some conditions C and D describing the way these entities are *permanently* related to each others. These conditions express the way the circuit is *coupled* with its environment.

$mode = env \Rightarrow C((state, input), (box, output))$ $mode = cir \Rightarrow D((state, input), (box, output))$

Condition C states what the circuit should establish (for the environment) provided it behaves in a situation where D holds. Conversely, condition D states what the environment should establish (for the circuit) provided it behaves in a situation where C holds.

1.5. Consistency Conditions

Nothing guarantees however that the dynamics envisaged above and the statics we have just described are coherent: this is something that has to be proved rigorously. It can be stated as follows:

$C((state, input), (box, output)) \wedge (s', i') \in F(output, state) \Rightarrow D((s', i'), (box, output))$ $D((state, input), (box, output)) \wedge (b', o') \in G(input, box) \Rightarrow C((state, input), (b', o'))$

Informally, this means that when *mode* is *env* and the static condition C holds, then D must hold after any accepted modifications s' and i' of *state* and *input* made by the environment. Likewise, when *mode* is *cir* and the static condition D holds, then C must hold after any accepted modifications o' and b' of *output* and *box* made by the circuit.

1.6. Decomposing the Circuit and the Environment

The above statements to prove are rather heavy because of the inherent non-determinacy. The idea is then to *decompose* the behaviors of each partner (environment and circuit) in various independent *deterministic pieces*, which we call *events*. For instance, the non-deterministic behavior of the partners can be decomposed in the following deterministic events, where conditions $P_1(output, state), \dots$, denotes the guards of the environmental events, whereas $Q_1(input, box), \dots$ denotes the guards of the circuit events:

Environment	$\left\{ \begin{array}{l} mode = env \wedge P_1(output, state) \implies state, input, mode := F_1(output, state), cir \\ \dots \end{array} \right.$
Circuit	$\left\{ \begin{array}{l} mode = cir \wedge Q_1(input, box) \implies box, output, mode := G_1(input, box), env \\ \dots \end{array} \right.$

Notice that the P guards may overlap, thus leading to the non-determinacy of the environment. Likewise, the Q guards may overlap, hence the non-determinacy of the circuit.

1.7 Consistency Conditions Revisited

Our consistency conditions are now transformed in the following collection of more tractable formulae:

$C((state, input), (box, output)) \wedge P_1(output, state) \implies D(F_1(output, state), (box, output))$ \dots
$D((state, input), (box, output)) \wedge Q_1(input, box) \implies C((state, input), G_1(input, box))$ \dots

1.8. A Warning

Note that this formulation corresponds to what we must obtain towards the *end of a formal development* where there should exist a very clear distinction between the circuit and the environment. During the development however, such a distinction is not necessarily as strict. For instance, we might allow for the possibility of the environment to access the previous *input* and even to access the *box* of the circuit. Likewise, we accept to have the circuit accessing its previous *output* and even the entire *state* of the environment. What must still be clearly followed however, even in an abstraction, is the limitation of modification: the environment modifies the *input* and the *state* only, whereas the circuit modifies the *box* and the *output* only.

One of the objective of the design of a circuit is precisely that of making the circuit and environment communicating eventually through the input and output lines only. For this, we have to *localize* their respective states.

2. A First Illustrating Example

As the previous discussion may appear to be rather dry, we shall now illustrate our approach by describing a little example of circuit specification and design.

2.1. Specification

The circuit we propose to study is a well-known benchmark that has been analysed in different contexts: it is called the *Single Pulser* (Pulser for short). Here is a first informal specification taken from [4]:

We have a debounced push-button, on (true) in the down position, off (false) in the up position. Devise a circuit to sense the depression of the button and assert an output signal for one clock pulse. The system should not allow additional assertions of the output until after the operator has released the button.

Here is another related specification [4], which is given under the form of three properties concerning the input I and the output O of the circuit:

1. *Whenever there is a rising edge at I , O becomes true some time later.*
2. *Whenever O is true it becomes false in the next time distance and it remains false at least until the next rising edge on I .*
3. *Whenever there is a rising edge, and assuming that the output pulse doesn't happen immediately, there are no more rising edges until that pulse happens (There can't be two rising edges on I without a pulse on O between them).*

My subjective impression after reading these specifications is that they are rather difficult to understand. I'd prefer to plunge the circuit to specify within a possible environment as follows:

1. We have a button that can be depressed and released by an operator. The button is connected to the input of the circuit.
2. We have a lamp that is able to be lit and subsequently turned down. The lamp is connected to the output of the circuit.
3. The circuit, situated between the button and the lamp, must make the lamp always flashing as many times as the button is depressed and subsequently released.

Here is a schematic representation of this closed system.

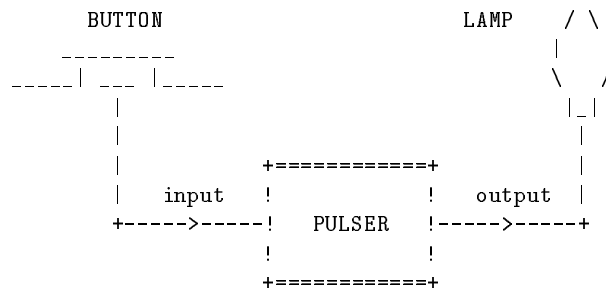
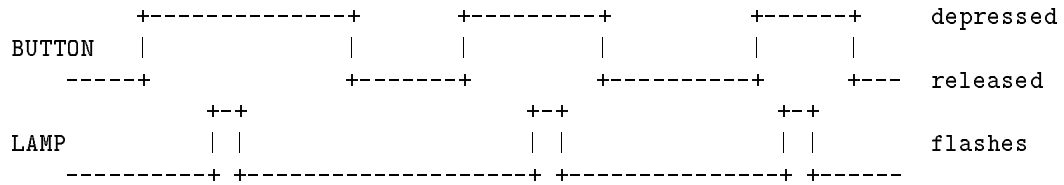


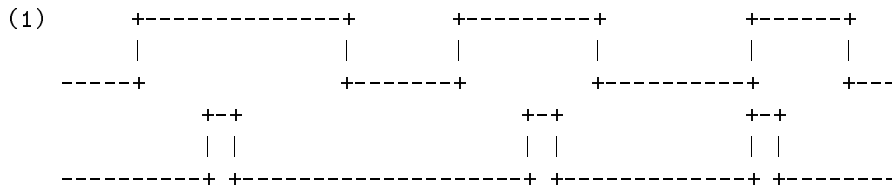
Fig. 4. The Pulser circuit in its environment

Note that the scenario we have described can be *observed* by an external witness: we can count the number of times the button is depressed by the operator and also the number of times the lamp flashes and we can

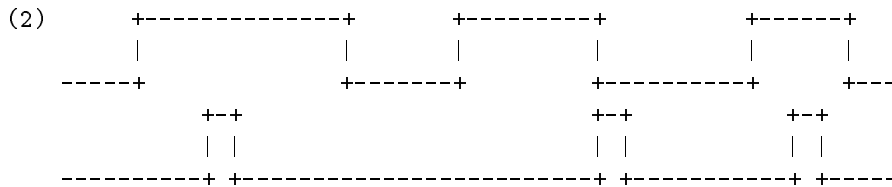
compare these numbers. For example, the following schemata shows two wave diagrams: the first one represents a succession of releases and subsequent depressions of the button, and the second shows various corresponding flashes of the lamp.



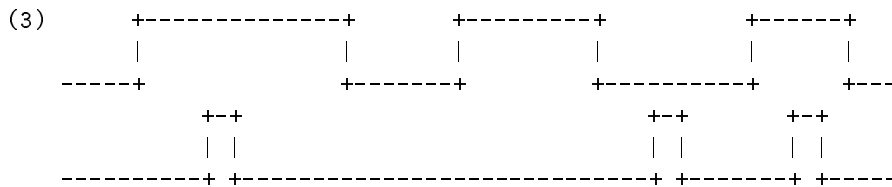
Notice that situations like the next ones can be encountered and are correct: in all cases we can observe that the number of lamp flashes always follows the number of depressings and subsequent releases of the button. In this first case, the third flash occurs as early as possible just after the depressing of the button



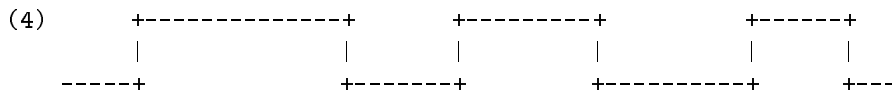
Next is a case where the second flash occurs just after the release of the button



In the diagram to follow, the second flash occurs well after the release of the button: this is correct (but perhaps risky) since the constraint concerning *push* and *flash* is indeed observed.

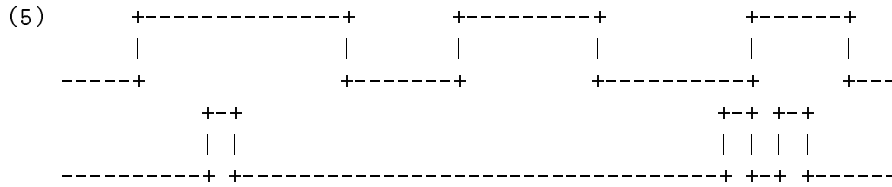


Next is a combination of cases (1) and (3) above.

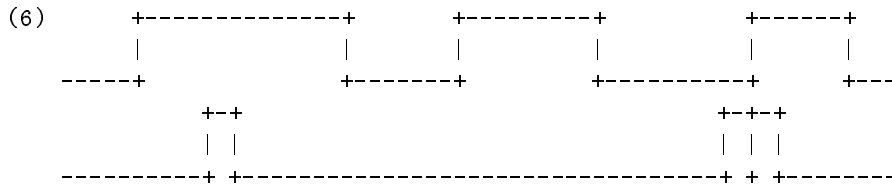




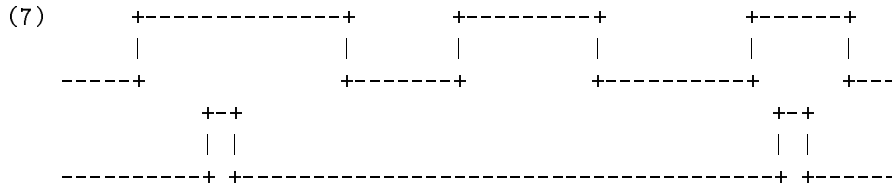
In the next case, the second flash occurs at the last possible moment. How does the lamp know that it has to flash now? Not clear.



On the other hand, situations like the next two ones are not correct: the number of flashes does not always follows the number of depressions and subsequent releases of the button. In the next case, we can observe that the lamp does not turn down. In fact, two successive flashes are glued together.



Next is a case where one flash is clearly missing.



Rather than representing directly the environment by the concrete input line and the circuit by the concrete output line (and probably some concrete internal state), we consider an *abstraction* where the environment is represented by two natural numbers, *push* and *pop*, denoting respectively the number of times the button is depressed and the number of times it is released (since the system has started). This yields the following environmental properties, stating quite naturally that *pop* is at least as *push* and at most one more than *push*: the button is released then you later push it (the button being obviously released when the circuit is started):

$push \in \mathbb{N}$ $pop \in \mathbb{N}$ $push \leq pop \leq push + 1$
--

The abstract circuit is “represented” by a single variable $flash$ denoting the number of times the lamp flashes. We have then the following properties showing the *coupling* between the abstract environment and the abstract circuit: $push$ is at least as $flash$ and at most one more than $flash$: you push the button then the lamp later flashes (the lamp being turned down when the circuit is started):

$$\begin{array}{l} flash \in \mathbb{N} \\ flash \leq push \leq flash + 1 \end{array}$$

The dynamics of the environment is straightforward: we have three events corresponding respectively to pushing the button (event `env1`), releasing it (event `env2`) and finally doing nothing (event `env3`). Clearly, we can depress the button only when pop is one more than $push$, and we can release it when pop and $push$ are identical, finally we can do nothing in all circumstances (the construct `SELECT C THEN $x := E$ || $y := F$ END` is a key-word version of the more concise $C \implies x, y := E, F$):

<pre>env1 ≐ SELECT mode = env ∧ pop = push + 1 THEN mode := cir push := push + 1 END</pre>	<pre>env2 ≐ SELECT mode = env ∧ pop = push THEN mode := cir pop := pop + 1 END</pre>	<pre>env3 ≐ SELECT mode = env THEN mode := cir END</pre>
---	---	--

The dynamics of the abstract circuit is apparently equally straightforward. There are two events corresponding to flashing the lamp (event `cir1`) or doing nothing (event `cir2`): we can flash the lamp when $push$ is one more than $flash$, and we can do nothing in all circumstances.

<pre>cir1 ≐ SELECT mode = cir ∧ push = flash + 1 THEN mode := env flash := flash + 1 END</pre>	<pre>cir2 ≐ SELECT mode = cir THEN mode := env END</pre>
---	--

The proof of consistency between the static properties and the events are straightforward except that *one of the proof fails*. This is the one concerning the maintenance of the invariant $push \leq flash + 1$ by the event `env1` (depressing the button). This is not surprising since, in this event, $push$ is incremented. Thus $push$ must be at most equal to $flash$ before the incrementation in order to preserve the condition $push \leq flash + 1$ after the incrementation (in the case where $push$ is already equal to $flash + 1$ before the incrementation, this property will obviously be violated after the incrementation). In fact, the formal statement to prove shows exactly the situation, namely $mode = env \wedge pop = push + 1 \implies push + 1 \leq flash + 1$, that is $mode = env \wedge pop = push + 1 \implies push \leq flash$. It reduces to the following after eliminating $push$:

$$mode = env \implies pop \leq flash + 1$$

Since, it cannot be proved, let us consider that this condition constitutes a *new static property*, which, by the way, is quite comprehensible, since it says that the number of button releases, denoted by pop , is at most one more than the number of lamp flashes, denoted by $flash$, *as soon as the circuit has reacted*, that is when the condition $mode = env$ holds.

Equipped with this new invariant, we resume our proofs. They all succeeds, again *except one* concerning, this time, the event $cir2$ (the one that makes the circuit doing nothing). Our new invariant causes the problem: since event $cir2$ only changes $mode$ from cir to env , its preservation requires that the condition $pop \leq flash + 1$ must hold *before* the event takes place, that is when $mode = cir$ holds. In other words, we must prove the following, which is not possible:

$$mode = cir \Rightarrow pop \leq flash + 1$$

What happens here is that the guard $mode = cir$ of event $cir2$ is *too weak*. In fact, it is not true that the circuit can do nothing in all circumstances. This can be done in certain circumstances only, more precisely in those making the condition $pop \leq flash + 1$ holding. We have thus no choice but to modify our event $cir2$ by incorporating the extra guard $pop \leq flash + 1$ into it (this is clearly the weakest condition we can imagine):

```

cir2 ≐
  SELECT
    mode = cir ∧
    pop ≤ flash + 1
  THEN
    mode := env
  END

```

We thus redo all our proofs (for the third time) and now they are *all discharged successfully*. Let us try to transform equivalently the new guard $pop \leq flash + 1$ of event $cir2$. The only properties of that kind we already have are the following

$$\begin{aligned}
push &\leq flash + 1 \\
pop &\leq push + 1
\end{aligned}$$

By comparing these conditions with $pop \leq flash + 1$, it appears that two possible alternative guards are $push = pop$ or $push = flash$. Conversely, it can be proved that the condition $pop \leq flash + 1$ implies this disjunction. In other words, we have

$$pop \leq flash + 1 \Leftrightarrow push = pop \vee push = flash$$

Since the condition $push = flash + 1$ is also here equivalent to $push \neq flash$, we can eventually rewrite as follows both events of the circuit:

```

cir1 ≐
SELECT
  mode = cir ∧
  push ≠ flash
THEN
  mode := env ||
  flash := flash + 1
END

```

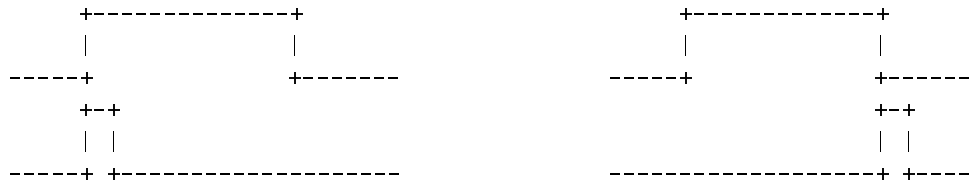
```

cir2 ≐
SELECT
  mode = cir ∧
  push = pop ∨ push = flash
THEN
  mode := env
END

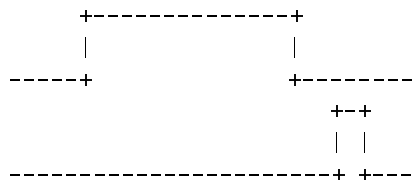
```

As can be seen, when $push \neq flash$ holds the circuit can either choose (event cir1) to flash the lamp or to postpone the flashing (event cir2), but the latter case is only possible provided $push = pop$ holds, in other words when the button is being depressed. As soon as the button is released (when $push \neq pop$ holds) and if the flash has not yet happened (when $push \neq flash$ still holds), then the guard of event cir2 becomes false and the circuit has no choice left (it is not possible to postpone the flashing any longer) except to fire the event cir1, which indeed flashes the lamp.

To summarize, what we have discovered is that the circuit can flash the lamp *at the earliest* just after the depressing of the button by the environmental event env1 (when $push \neq flash$ just holds), or *at the latest* just after the release of the button by the environmental event env2 (when $pop = push + 1$ just holds). Between these two events, the circuit is non-deterministic: again, it can choose either to flash the lamp or to do nothing. The following schemas show these two extreme situations:



Interestingly enough, a situation like the next one, although correct, is *potentially dangerous* as the button can be depressed again *at any time* as soon as it has been released. notice that his solution has been “naturally” rejected by our design:

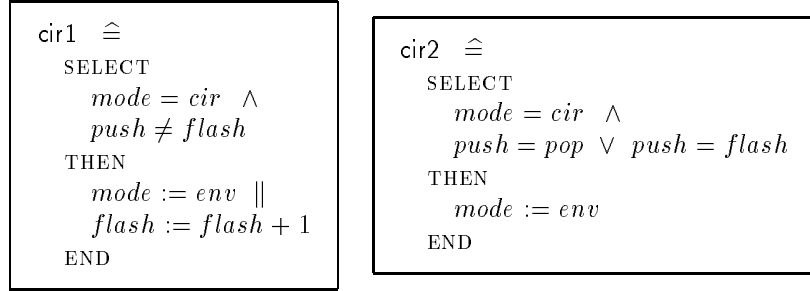


What we have shown with the development of this little example is that *the necessity of proving* has forced us to deeply understand the nature of our problem and has led us towards a correct solution: it has shown the concept of *design by proof* at work. But, of course, we have not yet a circuit. For the moment, we just have an abstraction of it (as well as one of our environment). The next step is to try to derive one or more deterministic circuits from our formal specification. But before that, we have to make a little detour through the concept of refinement.

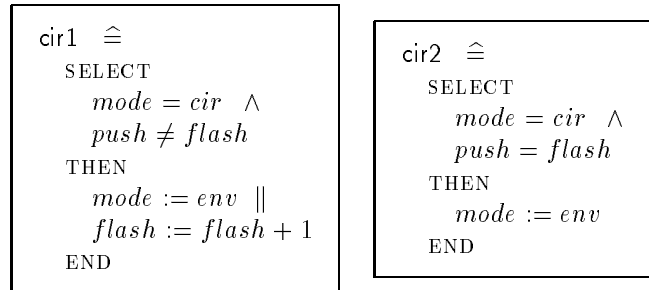
2.2. Refining the Circuit by Diminishing its Non-determinacy

In this section, we shall present a first way of refining our circuit. This corresponds to removing some (or all) of its possible non-deterministic behaviors.

When a circuit is made of various events whose guards are *overlapping* (i.e. can hold together), a possible refinement consists in having these guards made stronger so that they do not overlap any more, but are still covering all situations so that no deadlock is possible (this would happen should all guards of the circuit be false at the same time). Let us reconsider the two events of our circuit:



The guards, clearly, may overlap when $push \neq flash$ and $push = pop$ hold simultaneously. We can see immediately that there are two different ways of making this system of events deterministic without introducing deadlock: (1) by replacing the guard of *cir2* by $push = flash$, and (2) by adding the extra guard $push \neq pop$ to that of the event *cir1*. The net effect, in both cases, is to make each guard the negation of the other. Interestingly enough, these two solutions also corresponds exactly to the two special cases we have discussed in the previous section. The first solution, which we call **PULSER1**, corresponds to flashing the lamp as early as possible:



In this case, we have the extra invariant property stating that once the circuit has reacted (i.e. when $mode = env$ holds) then the number of flashes is always equal to the number of button depressions.

$$mode = env \Rightarrow push = flash$$

The second solution, which we call **PULSER2**, corresponds to flashing the lamp as late as possible.

```

cir1 ≐
  SELECT
    mode = cir ∧
    push ≠ pop ∧ push ≠ flash
  THEN
    mode := env ||
    flash := flash + 1
  END

```

```

cir2 ≐
  SELECT
    mode = cir ∧
    push = pop ∨ push = flash
  THEN
    mode := env
  END

```

In this case, we have the extra invariant property stating that the number of button releases is always different from the number of flashes

```

pop ≠ flash

```

2.3. Refining the Circuits by Changing the Data Space

The two circuits PULSER1 and PULSER2 we have obtained, although now completely deterministic, are still rather abstract. We would like to converge now towards some “real” circuits. In particular, the input and output lines should be defined, and the abstract numbers *push*, *pop*, and *flash* should be abandoned. The purpose of this section is to show how refinement allows one to change our data space.

We have two new variables *inp* and *oup*, which corresponds to the input and output lines respectively. These variables take their value within the “boolean” set {0, 1}

```

inp ∈ {0,1}
oup ∈ {0,1}

```

The variable *inp* is an environmental variables: it is modified by both events *env1* and *env2*. The abstract variable *push* is supposed to denote the number of times the variable *inp* moves from 0 to 1. Likewise the abstract variable *pop* is supposed to denote the number of times the variable *inp* moves from 1 to 0. This leads to the following new events *env1* and *env2*:

```

env1 ≐
  SELECT
    mode = env ∧
    inp = 0
  THEN
    mode := cir ||
    inp := 1
  END

```

```

env2 ≐
  SELECT
    mode = env ∧
    inp = 1
  THEN
    mode := cir ||
    inp := 0
  END

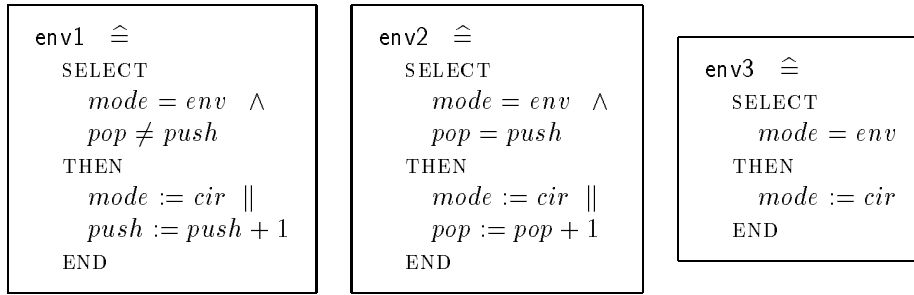
```

```

env3 ≐
  SELECT
    mode = env
  THEN
    mode := cir
  END

```

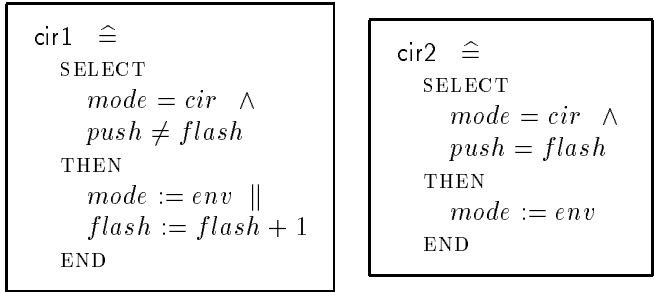
For these events to be correct refinements of their abstract counterparts, each concrete guard must imply the corresponding abstract guard. Here is a copy of the abstractions (where we have replaced the guard *pop = push + 1* of event *env1* by the equivalent condition *pop ≠ push*):



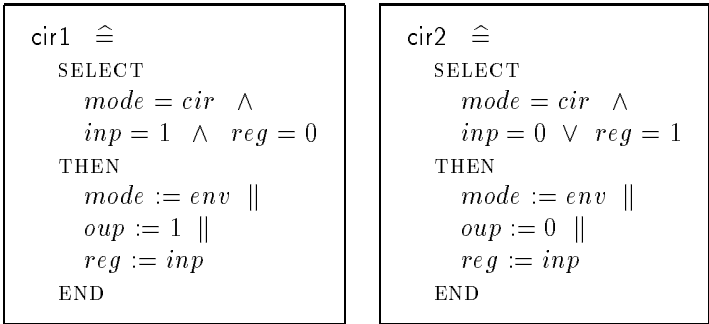
The correct refinement thus clearly involves proving the following relationship between the concrete environmental space and the abstract one (this proof is straightforward):

$$inp = 1 \Leftrightarrow pop = push$$

Let us now turn to the implementation of the abstract circuit **PULSER1**. We have the following abstract circuit events:



The abstract circuit variable *flash* has to disappear. It counts the number of time the concrete variable *oup* moves from 0 to 1. For this, the guard of the concrete event *cir1* must check that the abstract variable *push* has just been modified by the environment. As we know, this is when the input line *inp* moves from 0 to 1. Clearly, we can access the actual value of *inp*, but certainly *not its previous value*. We have no choice then but to introduce a register, *reg*, internal to our circuit, and whose rôle is to "store" the previous value of *inp*. This leads to the following tentative "implementation" of the events *cir1* and *cir2*



The concrete guards must imply the abstract ones. We also clearly have an equality between *reg* and *inp* when *mode = env* holds. All this leads to the following properties to be maintained (which are both easily provable):

We do not know whether it is possible to build such a circuit. We do not know either whether such a circuit, supposedly constructed, is free from any deadlock in some situations.

3.2. Formal Static Specification

In the formal specification, we shall abstract from the boolean input and output lines as described in the previous section. We consider that in the environment, we can count the numbers $r1$ and $r2$ of requirements made by each user and the corresponding numbers $a1$ and $a2$ of acknowledgements made by the circuit. The constraint of the informal specification imposes the following straightforward permanent invariant where it is stated that the number of requests is at most one more than the number of acknowledgements:

$$\begin{array}{l} r1 \in \mathbb{N} \ \wedge \ r2 \in \mathbb{N} \\ a1 \in \mathbb{N} \ \wedge \ a2 \in \mathbb{N} \\ \\ a1 \leq r1 \leq a1 + 1 \\ a2 \leq r2 \leq a2 + 1 \end{array}$$

The “waiting time” of any requiring user is at most equal to 1. This can be formalized by two counters $n1$ and $n2$ that are incremented as soon that a requesting user is not served and reset in case it has just been served:

$$n1 \in \{0, 1\} \ \wedge \ n2 \in \{0, 1\}$$

When the *mode* is *cir*, we have a situation that has just been established by the environment for the circuit. In this case, one of the waiting time at least should be equal to 0 (otherwise the circuit would have to grant the resource to both users, which is not allowed). We have thus:

$$mode = cir \ \Rightarrow \ (n1 = 0 \ \vee \ n2 = 0)$$

Finally, it should be clear that users having a positive waiting time are requiring users, yielding

$$\begin{array}{l} mode = cir \ \Rightarrow \ (n1 \neq 0 \ \Rightarrow \ r1 \neq a1) \\ mode = cir \ \Rightarrow \ (n2 \neq 0 \ \Rightarrow \ r2 \neq a2) \end{array}$$

When the *mode* is *env*, we have a situation that has just been established by the circuit for the environment. The fact that the circuit does something as soon as it can is formalized by saying that one of the requests at least is satisfied. In other words the number of requests is equal to the number of acknowledgements for at least one user. Formally:

$$mode = env \ \Rightarrow \ (r1 = a1 \ \vee \ r2 = a2)$$

3.3. Formal Dynamic Specification of the Environment

The environment first systematically resets or increments the “waiting counter” of each user depending on the situation of the request (if no request is pending the counter is reset, if some request is pending, it is incremented). This yields the following fragment, which appears in each environmental event:

```
IF  $r1 = a1$  THEN  $n1 := 0$  ELSE  $n1 := n1 + 1$  END ||
IF  $r2 = a2$  THEN  $n2 := 0$  ELSE  $n2 := n2 + 1$  END
```

The various events correspond to new requests being posted either individually (`env1` and `env2`) or simultaneously `env3`, or to the environment doing nothing (`env0`).

```
env1  $\hat{=}$ 
SELECT
  mode = env  $\wedge$   $r1 = a1$ 
THEN
  mode := cir ||
  r1 := r1 + 1 ||
  IF  $r1 = a1$  THEN  $n1 := 0$  ELSE  $n1 := n1 + 1$  END ||
  IF  $r2 = a2$  THEN  $n2 := 0$  ELSE  $n2 := n2 + 1$  END
END
```

```
env2  $\hat{=}$ 
SELECT
  mode = env  $\wedge$   $r2 = a2$ 
THEN
  mode := cir ||
  r2 := r2 + 1 ||
  IF  $r1 = a1$  THEN  $n1 := 0$  ELSE  $n1 := n1 + 1$  END ||
  IF  $r2 = a2$  THEN  $n2 := 0$  ELSE  $n2 := n2 + 1$  END
END
```

```
env3  $\hat{=}$ 
SELECT
  mode = env  $\wedge$ 
  r1 = a1  $\wedge$  r2 = a2
THEN
  mode := cir ||
  r1, r2 := r1 + 1, r2 + 1 ||
  IF  $r1 = a1$  THEN  $n1 := 0$  ELSE  $n1 := n1 + 1$  END ||
  IF  $r2 = a2$  THEN  $n2 := 0$  ELSE  $n2 := n2 + 1$  END
END
```

```
env0  $\hat{=}$ 
SELECT
  mode = env
THEN
  mode := cir ||
  IF  $r1 = a1$  THEN  $n1 := 0$  ELSE  $n1 := n1 + 1$  END ||
  IF  $r2 = a2$  THEN  $n2 := 0$  ELSE  $n2 := n2 + 1$  END
END
```

3.4. Formal Dynamic Specification of the circuit

The events of the circuit are very simple. In case an input line is 1 (in events `cir1` and `cir2`), the event increments the acknowledgement counter. It does so, however, provided the other user has not itself required the resource for more than one clock pulse. When no request is made (in event `cir0`), the event does nothing:

```
cir1  $\hat{=}$ 
SELECT
  mode = cir  $\wedge$ 
  r1  $\neq$  a1  $\wedge$  n2 = 0
THEN
  mode := env ||
  a1 := a1 + 1
END
```

```
cir2  $\hat{=}$ 
SELECT
  mode = cir  $\wedge$ 
  r2  $\neq$  a2  $\wedge$  n1 = 0
THEN
  mode := env ||
  a2 := a2 + 1
END
```

```
cir0  $\hat{=}$ 
SELECT
  mode = cir  $\wedge$ 
  r1 = a1  $\wedge$  r2 = a2
THEN
  mode := env
END
```

3.5. Proving Consistency and Strengthening the Invariant

When proving the closed system consistency with Atelier B, 72 “proof obligations” (in our jargon, this means the statements of the little lemma to be proved) were generated, out of which only 4 (that is 5.5%) were not discharged by the automatic prover. These are the following (two times each):

$$\begin{aligned} mode = env \wedge r2 = a2 \wedge r1 \neq a1 &\Rightarrow n1 + 1 \in \{0, 1\} \\ mode = env \wedge r1 = a1 \wedge r2 \neq a2 &\Rightarrow n2 + 1 \in \{0, 1\} \end{aligned}$$

The first one has been generated by environmental events `env0` and `env2` trying to preserve the invariant $n1 \in \{0, 1\}$ while incrementing $n1$. The second proof obligation denotes a similar situation with $n2$, it is generated by events `env0` and `env1`. We have the feeling that, in the first one, $r2$ plays no rôle, and similarly with $r1$ in the second one. The consequent $n1 + 1 \in \{0, 1\}$ could be equivalently replaced by $n1 = 0$ since we already have $n1 \in \{0, 1\}$, similarly with $n2 + 1 \in \{0, 1\}$. These statements could then be replaced by the following slightly stronger statements:

$$\begin{aligned} mode = env &\Rightarrow (r1 \neq a1 \Rightarrow n1 = 0) \\ mode = env &\Rightarrow (r1 \neq a2 \Rightarrow n2 = 0) \end{aligned}$$

As we have absolutely no way of proving these statements, we have no choice left but to suppose that they are invariant. In fact these statements are quite intuitive: they say that in the environment a requiring user ($r1 \neq a1$) cannot have its “waiting counter” positive since otherwise this would precisely mean that it is now waiting for more than one clock pulse. After incorporating these new invariants and reproofing our system, Atelier B generates 81 proof obligations, which are now all discharged automatically.

3.6. Proving Deadlockfreeness

Nothing guarantees, of course, that the circuit events are not stuck because their guards are simultaneously false. We have thus to prove the following, stating that while in the *cir* mode, the disjunction of the guards of the circuit always hold, (this is easily discharged being just a consequence of the invariant):

$$mode = cir \Rightarrow (r1 \neq a1 \wedge n2 = 0) \vee (r2 \neq a2 \wedge n1 = 0) \vee (r1 = a1 \wedge r2 = a2)$$

Note however that the circuit is still non-deterministic: this is the case when both users are just requiring the resource simultaneously (thus $n1 = 0$ and $n2 = 0$ hold simultaneously). In this case, both events `cir1` and `cir2` can be fired.

3.7. Generating Proper Binary Outputs from the Circuit

In section 3.4, the circuit events `cir1` and `cir2` incremented directly the acknowledgement counters $a1$ and $a2$. These counters both formed the abstract outputs of our circuit. We shall now postpone this incrementation and have the circuit only generating a 1 or 0 offset, the proper incrementation itself being done by the environment on two slightly time-shifted counters, say $b1$ and $b2$. This refinement introduces thus four variables typed as follows:

$$\begin{array}{l}
b1 \in \mathbb{N} \quad \wedge \quad b2 \in \mathbb{N} \\
o1 \in \{0, 1\} \quad \wedge \quad o2 \in \{0, 1\}
\end{array}$$

The “gluing” invariant that holds between the abstract counters $a1$ and $a2$ and the new concrete variables we have just introduced is the following:

$$\begin{array}{l}
mode = cir \quad \Rightarrow \quad a1 = b1 \\
mode = cir \quad \Rightarrow \quad a2 = b2 \\
\\
mode = env \quad \Rightarrow \quad a1 = b1 + o1 \\
mode = env \quad \Rightarrow \quad a2 = b2 + o1
\end{array}$$

The last two statements indicate that, while we are observing the environment (just after the reaction of the circuit), the abstract counters ai are already incremented (by the abstract circuit) while the concrete counters bi are not. In fact, they will be incremented in the environment thanks to the contents of the output oi . On the other hand, the first two statements indicate that while observing the circuit, the abstract and concrete counters are now “in phase”.

The environmental events are all modified in a straightforward way. We shall just show one of them (together with its old version presented first) since the other ones are modified in very much the same way:

```

env1  $\hat{=}$ 
SELECT
  mode = env  $\wedge$  r1 = a1
THEN
  mode := cir ||
  r1 := r1 + 1 ||
  IF r1 = a1 THEN n1 := 0 ELSE n1 := n1 + 1 END ||
  IF r2 = a2 THEN n2 := 0 ELSE n2 := n2 + 1 END
END

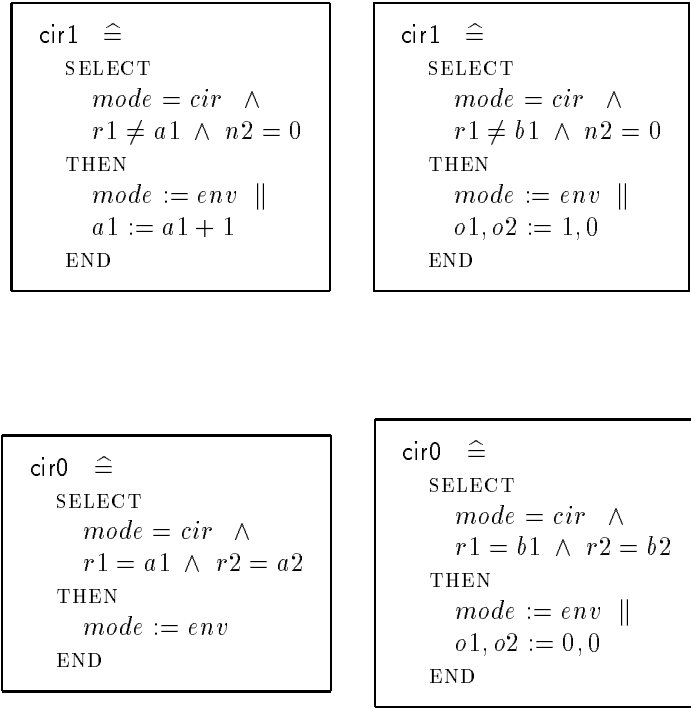
```

```

env1  $\hat{=}$ 
SELECT
  mode = env  $\wedge$  r1 = b1 + o1
THEN
  mode := cir ||
  r1 := r1 + 1 ||
  IF r1 = b1 + o1 THEN n1 := 0 ELSE n1 := n1 + 1 END ||
  IF r2 = b2 + o2 THEN n2 := 0 ELSE n2 := n2 + 1 END ||
  b1, b2 := b1 + o1, b2 + o2
END

```

The circuit events are modified accordingly. We present in what follows the old versions of $cir1$ and $cir0$ to the left of the new ones:



Atelier B generated 94 proof obligations for proving the correct refinements, out of which 6 (6.4%) were not discharged automatically. They were discharged interactively very easily.

3.7. Generating Proper Binary Inputs from the Environment

The inputs to the circuit, rather than being the number r_i of requests and the number b_i of acknowledgements could very well be only their *difference* which is at most 1, as we know. For this, we introduce two new binary variables i_1 and i_2 , glued as follows to r_i and b_i

$$\begin{aligned}
mode = cir &\Rightarrow r1 = b1 + i1 \\
mode = cir &\Rightarrow r2 = b2 + i2
\end{aligned}$$

The modification of the environment events are very simple. As for the output in previous section, we shall only present one such event together with its previous version:

```

env1 ≐
SELECT
  mode = env ∧ r1 = b1 + o1
THEN
  mode := cir ||
  r1 := r1 + 1 ||
  IF r1 = b1 + o1 THEN n1 := 0 ELSE n1 := n1 + 1 END ||
  IF r2 = b2 + o2 THEN n2 := 0 ELSE n2 := n2 + 1 END ||
  b1, b2 := b1 + o1, b2 + o2
END

```

```

env1 ≐
SELECT
  mode = env ∧ r1 = b1 + o1
THEN
  mode := cir ||
  r1, i1 := r1 + 1, 1 ||
  IF r1 = b1 + o1 THEN n1 := 0 ELSE n1 := n1 + 1 END ||
  IF r2 = b2 + o2 THEN n2, i2 := 0, 0 ELSE n2, i2 := n2 + 1, 1 END ||
  b1, b2 := b1 + o1, b2 + o2
END

```

The circuit events are very simply modified by replacing the guard $r1 \neq b1$ by $i1 = 1$ in cir1 and similarly in cir0 (again we present the new versions together with the old ones):

```

cir1 ≐
SELECT
  mode = cir ∧
  r1 ≠ b1 ∧ n2 = 0
THEN
  mode := env ||
  o1, o2 := 1, 0
END

```

```

cir1 ≐
SELECT
  mode = cir ∧
  i1 = 1 ∧ n2 = 0
THEN
  mode := env ||
  o1, o2 := 1, 0
END

```

```

cir0 ≐
SELECT
  mode = cir ∧
  r1 = b1 ∧ r2 = b2
THEN
  mode := env ||
  o1, o2 := 0, 0
END

```

```

cir0 ≐
SELECT
  mode = cir ∧
  i1 = 0 ∧ i2 = 0
THEN
  mode := env ||
  o1, o2 := 0, 0
END

```

Atelier B generated 24 proof obligations for proving the correct refinements, out of which 6 (25%) were not discharged automatically. They were discharged interactively very easily.

3.8. Localizing some Environment Variables in the Circuit

The previous version of the circuit, besides $i1$ and $i2$, is still using another input from the environment, namely the “waiting counters” $n1$ and $n2$. The idea is to “move” these counters from the environment to the circuit. For this, we introduce two circuit “registers”, $p1$ and $p2$ typed as follows:

$$\begin{array}{l} p1 \in \{0, 1\} \\ p2 \in \{0, 1\} \end{array}$$

The glue between the p and n variables is that they are ideed equal as seen from the circuit (we suspect that there are also some laws concerning these variables, as seen from the environment, but we do not know which by now), namely:

$$\begin{array}{l} made = cir \Rightarrow p1 = n1 \\ made = cir \Rightarrow p2 = n2 \end{array}$$

The environmental events are not modified, except that the variables $n1$ and $n2$ could be removed. The circuit events are modified as follows

```

cir1 ≐
SELECT
  mode = cir ∧
  i1 = 1 ∧ n2 = 0
THEN
  mode := env ||
  o1, o2 := 1, 0
END

```

```

cir1 ≐
SELECT
  mode = cir ∧
  i1 = 1 ∧ p2 = 0
THEN
  mode := env ||
  o1, o2, p1 := 1, 0, 0 ||
  IF i2 = 1 THEN p2 := 1 END
END

```

```

cir0 ≐
SELECT
  mode = cir ∧
  i1 = 0 ∧ i2 = 0
THEN
  mode := env ||
  o1, o2 := 0, 0
END

```

```

cir0 ≐
SELECT
  mode = cir ∧
  i1 = 0 ∧ i2 = 0
THEN
  mode := env ||
  o1, o2 := 0, 0 ||
  p1, p2 := 0, 0
END

```

Atelier B generates 38 proof obligations for proving this refinement. 18 were not proved by the automatic prover. The ones that were not proved are all concerned with the environmental events. A rapid inspection showed that two invariants were missing, namely:

$$\begin{array}{l} made = env \Rightarrow (r1 = b1 + o1 \Leftrightarrow p1 = 0) \\ made = env \Rightarrow (r2 = b2 + o1 \Leftrightarrow p2 = 0) \end{array}$$

By reproofing our system with this new invariants, Atelier B generated 52 proof obligations, out of which only 2 (3.8%) needed an easy interactive proof.

3.9. Reducing non-determinacy

The previous circuit we have obtained is now complete and simple, but still non-deterministic. When $i1$ and $i2$ are both equal to 1 with $p1$ and $p2$ both equal to 0, the circuit can choose to raise $o1$ or $o2$. In order to make the circuit completely deterministic, we decide that, in this case, $o1$, say, will be the winner. We also make all events *having the same "action part"*. This yields the following:

```

cir1 ≐
SELECT
  mode = cir ∧
  i1 = 1 ∧ p2 = 0
THEN
  mode := env ||
  IF i1 = 1 ∧ p2 = 0          THEN o1 := 1 ELSE o1 := 0 END ||
  IF ¬(i1 = 1 ∧ p2 = 0) ∧ i2 = 1 THEN o2 := 1 ELSE o2 := 0 END ||
  IF i2 = 1 ∧ i1 = 1 ∧ p2 = 0   THEN p2 := 1 ELSE p2 := 0 END
END

```

```

cir2 ≐
SELECT
  mode = cir ∧
  ¬(i1 = 1 ∧ p2 = 0) ∧ i2 = 1
THEN
  mode := env ||
  IF i1 = 1 ∧ p2 = 0          THEN o1 := 1 ELSE o1 := 0 END ||
  IF ¬(i1 = 1 ∧ p2 = 0) ∧ i2 = 1 THEN o2 := 1 ELSE o2 := 0 END ||
  IF i2 = 1 ∧ i1 = 1 ∧ p2 = 0   THEN p2 := 1 ELSE p2 := 0 END
END
END

```

```

cir0 ≐
SELECT
  mode = cir ∧
  i1 = 0 ∧ i2 = 0
THEN
  mode := env ||
  IF i1 = 1 ∧ p2 = 0          THEN o1 := 1 ELSE o1 := 0 END ||
  IF ¬(i1 = 1 ∧ p2 = 0) ∧ i2 = 1 THEN o2 := 1 ELSE o2 := 0 END ||
  IF i2 = 1 ∧ i1 = 1 ∧ p2 = 0   THEN p2 := 1 ELSE p2 := 0 END
END
END

```

Atelier B generates 5 proof obligations, 1 of them (20%) requiring an easy interactive proof.

3.11. Revisiting Deadlockfreeness

The interesting and fundamental last statement to prove is that the events of the circuit are deadlockfree. For this, we have to prove that, under the hypothesis $mode = cir$, the disjunction of the guards of the circuit events are true (the interactive proof of this statement is easy), namely:

$$mode = cir \Rightarrow (i1 = 1 \wedge p2 = 0) \vee (\neg(i1 = 1 \wedge p2 = 0) \wedge i2 = 1) \vee (i1 = 0 \wedge i2 = 0)$$

3.11. Building the Final Circuit

By removing references to the $mode$ in the circuit events and putting them together in an obvious fashion (the guards vanishing as their disjunction is true), we obtain the following

```

cir012 ≐
BEGIN
  IF i1 = 1 ∧ p2 = 0          THEN o1 := 1 ELSE o1 := 0 END ||
  IF ¬(i1 = 1 ∧ p2 = 0) ∧ i2 = 1 THEN o2 := 1 ELSE o2 := 0 END ||
  IF i2 = 1 ∧ i1 = 1 ∧ p2 = 0    THEN p2 := 1 ELSE p2 := 0 END
END

```

This leads to the following circuit (where the “ \ominus ” symbol means complementation):

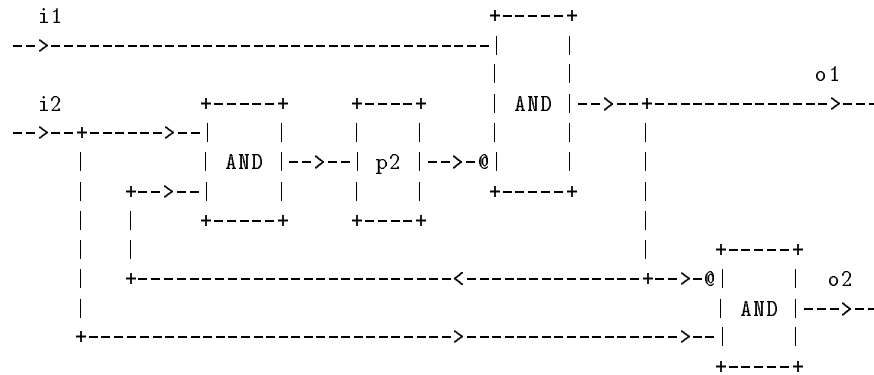


Fig. 6. The final circuit

Note that the overall development generated 271 proof obligations, among which 17 (6.3%) needed an (easy) interactive proof.

4. A More Elaborate Example

The example we develop in this section is one where a complete (but still simple) system is considered with the circuit aimed at controlling a physical environment by reacting appropriately. It is taken from Mead and Conway [?]. In this example, we also experiment with the idea of *connecting* various circuits together.

4.1 System Specification

One intends to install a traffic light at the crossing between a main road and a small road. The idea is to have these lights behaving in such a way that the traffic on the main road is somehow given a certain advantage over that on the small road. The corresponding policy is explained (and commented) in the following informally stated rules:

- Rule 1** When the light controlling the main road is green, it only turns orange (and subsequently red) when some cars are present on the small road (the presence of such cars is detected by appropriate sensors). As a consequence, when no cars are present on the small road, the traffic on the main road is not disturbed.
- Rule 2** This potential loss of priority on the main road is however only possible provided that road has already kept the priority for at least a certain (long) fixed delay. In other words, within that delay, the main road keeps the priority even if there are cars waiting on the small road. As a consequence, when there are frequently coming cars on the small road, the traffic on the main road is still rather smoothly flowing.
- Rule 3** On the other hand, the small road, when given priority, keeps it as long as there are cars willing to cross the main road.
- Rule 4** This keeping of the priority by the small road is however only possible provided a (long) delay (the same delay as for the main road) has not passed. When the delay is over, the priority systematically returns to the main road even if there are still some cars present on the small road. As a consequence, when there is a big amount of cars on the small road, these cars cannot block the main road for too long a period of time.
- Rule 5** As already alluded above, a green light does not turn red immediately. An orange color appears as usual for a (small) amount of time before the light definitely turns red. This sequential behavior is the same on the lights of both roads.
- Rule 6** As usual, the safety of the drivers is ensured by the fact that the light, when green or orange on one road, is exactly red on the other one, and vice-versa. Safety is also ensured, of course, provided the drivers obey the law of not trespassing a red light (but this is another matter, not under the responsibility of the circuit!).

4.2 A Separation of Concern Approach

By reading the previous informal requirements, it appears that there are apparently *two separate questions* in this problem: (1) one is dealing with the modification of the priority from the main to the small road and vice-versa (this corresponds to **Rule 1** to **Rule 4** above), and (2) another one is dealing with the realization of that change of priority in a way that is meaningful to drivers (this corresponds to **Rule 5** and **Rule 6**): this concerns the modification of the colors of each light (from successively, say, green to orange, then to red, and then to green again, etc), and the obvious non-contradiction between the lights governing each road (no two green lights at the same time, etc).

It seems that these two questions are rather “orthogonal” in that a modification in the road priority policy should not affect the proper behaviors of the lights, and vice-versa. Clearly, a modification in the light classical behaviour is not something that one would reasonably envisage as it is rather universal. On the other hand, a modification in the priority policy is a possibility that could not be rejected a priori. In that case, we would like to have the circuit built in such a way that this modification could be done in an easy way (sub-circuit replacement).

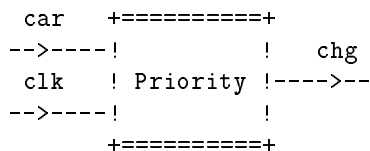
One should also notice that the first of these two questions deals with the essential *function* of this system, namely to alternate the priority between two roads in an unbalanced way. On the other hand, the second

question rather deals with the *safety* and possible *progress* of the users. In other words, we must ensure that drivers: (i) are always in a safe situation provided they obey the usual conventions indicated by the colors of the lights, and (ii) are also not blocked indefinitely (everybody has once experienced a situation where, for instance, both lights are red!).

Our initial idea is thus to make the design of *two distinct circuits*, which will be eventually connected. One is the **Priority** circuit, and the other is the **Light** circuit. The **Priority** circuit delivers a signal to the **Light** circuit telling that the priority has to be changed from one road to the other. In this way, the latter can translate this “priority” information in terms of a corresponding “traffic light” information.

4.3 The (simplified) Priority Circuit

The simplest **Priority** circuit we can think of is one with an external boolean entry, *car*, corresponding to the information elaborated by the car sensors disposed on the small road, and with a boolean exit, *chg*, yielding the information concerning a *change* in the priority. It also has another boolean entry, *clk*, which is an alarm coming from an external “timer”.



This timer sends an alarm on the boolean entry *clk* when (and as long as) the long delay described above is over. The circuit “decides” to possibly change the priority depending on three factors: (1) the actual priority (main road or small road), stored somehow in the circuit, (2) the presence of cars on the small road, and (3) the state of the alarm coming from the timer. The external wires of the circuit are thus declared as follows:

$$car, clk, chg \in \{0, 1\} \times \{0, 1\} \times \{0, 1\}$$

We have the following conventions: (1) *car* valued to 1 means that some cars are waiting on the small road, (2) *clk* valued to 1 means that the long delay is over, and (3) *chg* valued to 1 means that the priority has to change.

4.3.1 An “Off the Shelf” Timer

Before engaging more deeply in the specification and design of our **Priority** circuit, we could have a small look at the timer alluded above. A **Timer** circuit has a boolean entry, *rst*, which, when valued to 1, resets an internal counter, *tck*, to a positive pre-defined constant value *tmp*. And it has a boolean exit, *alm*, which, when valued to 1, is the alarm “ringing” as soon (and as long) as the internal counter *tck* has reached the null value.

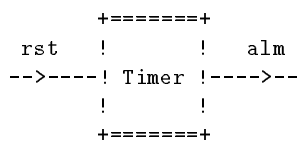


Fig. 7. External Interface of the Timer

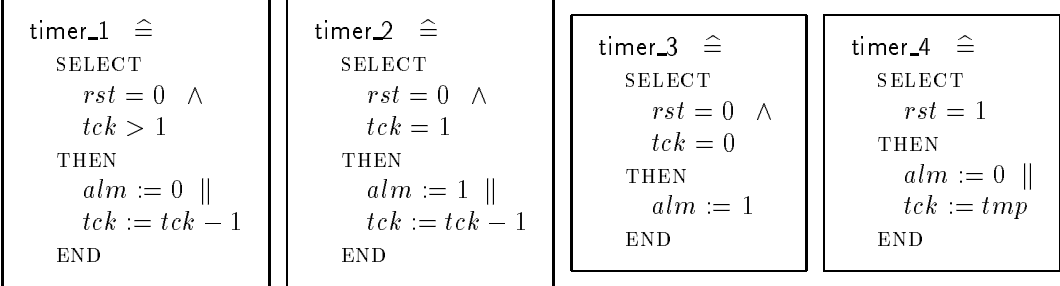
The constant *tmp* and the variables *rst*, *tck*, and *alm* are formally defined as follows:

```

tmp ∈ ℕ
tmp > 0
rst ∈ {0, 1}
alm ∈ {0, 1}
tck ∈ 0 .. tmp
alm = 1 ⇔ tck = 0

```

The last invariant states that *alm* is exactly set when the counter *tck* is null. The straightforward events of Timer are as follows:



As can be seen, when the Timer is not reset (*rst* = 0), then the counter *tck* is decremented if positive (in *timer_1* and *timer_2*) until it reaches the value 0 (in *timer_2* and *timer_3*), in which case the exit *alm* is set to 1. When the Timer is reset (*rst* = 1), then the counter *tck* is set to the positive value *tmp* and *alm* is reset to 0 (in *timer_4*).

4.3.2 Putting the Priority and Timer circuits together

Our next step is to inter-connect the two previous circuits by identifying the exit *chg* of Priority with the entry *rst* of Timer and the entry *clk* of Priority with the exit *alm* of Timer. This is so because a change of priority clearly implies resetting the Timer. The entry *car* and the Timer circuit together form the environment of the Priority circuit. Conversely, we could consider that the Priority circuit forms the environment of the Timer circuit. As can be seen, the notion of environment is *relative*. This yields the following schemata. The dynamic

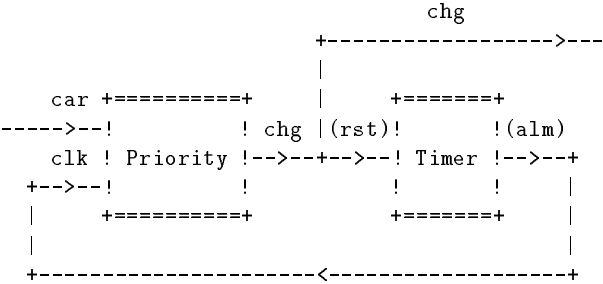
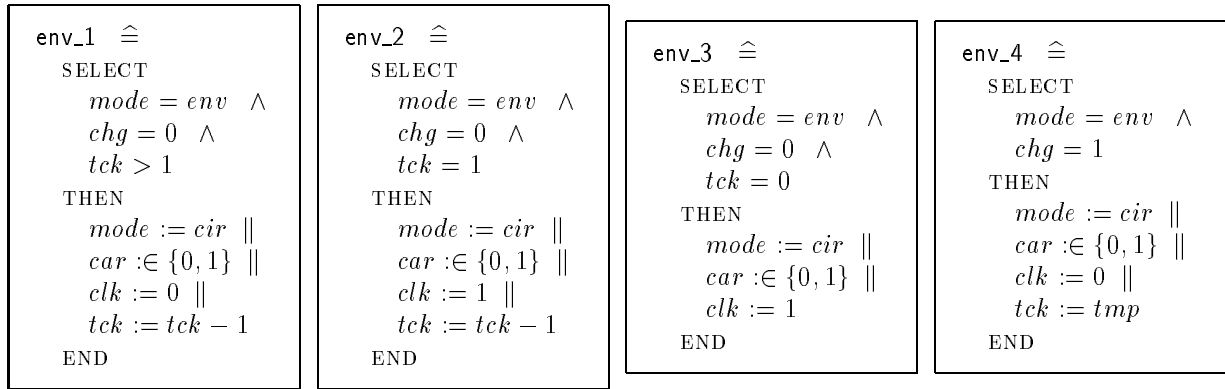


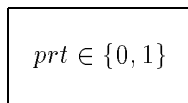
Fig. 8. Coupling the Timer and Priority circuits

environment of the Priority circuit is made by “repainting” the events of Timer (*rst* replaced by *chg* and *alm* replaced by *clk*) and by non-deterministically assigning the *car* entry (*car* ∈ {0, 1}).



4.3.3 Specifying the (simplified) Priority Circuit

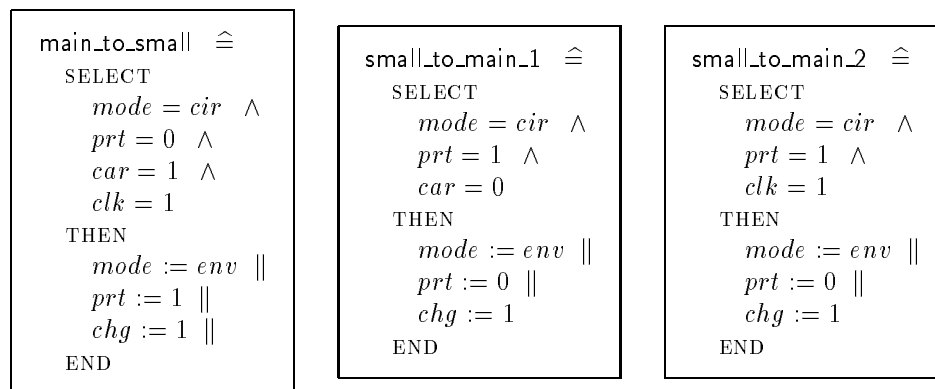
The Priority circuit has an internal register, *pri*, holding the actual priority. It is valued 0 (priority on main road) or 1 (priority on small road).



The events of the Priority circuit elaborate priority changes. We have three such events, called *main_to_small*, *small_to_main_1*, and *small_to_main_2*. Their guards formally state under which circumstances the priority can change. This is explained in what follows:

1. Event *main_to_small* can be fired when the priority is on main road ($prt = 0$), when some cars are present on the small road ($car = 1$) and when the long delay has passed ($clk = 1$): this corresponds to **Rule 1** and **Rule 2** above.
2. Event *small_to_main_1* can be fired when the priority is on small road ($prt = 1$), and when no cars are present on the small road ($car = 0$): this corresponds to **Rule 3**.
3. Event *small_to_main_2* can be fired when the priority is on small road ($prt = 1$) and when the long delay has passed ($clk = 1$): this corresponds to **Rule 4**.

In all three cases, the priority changes ($chg := 1$) and *pri* is modified accordingly. Here are the events:



Another series of events corresponds to the circuit doing nothing except resetting the *chg* exit to 0 (no change). This occurs in three circumstances:

1. Event `do_nothing_1` can be fired when the priority is on main road ($prt = 0$) and when there are no cars on the small road ($car = 0$): this corresponds to **Rule 1** above.
2. Event `do_nothing_2` can be fired when the priority is on main road ($prt = 0$) and when the delay has not passed yet ($clk = 0$): this corresponds to **Rule 2**.
3. Event `do_nothing_3` can be fired when the priority is on small road ($prt = 1$), when there are cars present on the small road ($car = 1$), and when the delay has not passed yet ($clk = 0$): this corresponds to **Rule 3** and **Rule 4**.

Here are these events:

<pre>do_nothing_1 ≡ SELECT mode = cir ∧ prt = 0 ∧ car = 0 THEN mode := env chg := 0 END</pre>	<pre>do_nothing_2 ≡ SELECT mode = cir ∧ prt = 0 ∧ clk = 0 THEN mode := env chg := 0 END</pre>	<pre>do_nothing_3 ≡ SELECT mode = cir ∧ prt = 1 ∧ car = 1 ∧ clk = 0 THEN mode := env chg := 0 END</pre>
--	--	--

4.3.4 Non-determinacy and Deadlockfreeness

We notice that the priority change events and the “do-nothing” ones do not overlap, their guards being clearly disjoint (which is reassuring: the circuit “knows” what it has to do). On the other hand, there is a possible non-determinacy between `small_to_main_1` and `small_to_main_2`, and also between `do_nothing_1` and `do_nothing_2`. It does not matter however since these couples of events have the same “action” parts.

The circuit does not deadlock as the disjunction of the guards of the various events obviously holds. Hint: observe the guard of `main_to_small` versus those of `do_nothing_1` and `do_nothing_2`, and the guard of `do_nothing_3` versus those of `small_to_main_1` and `small_to_main_2`.

4.3.5 Designing the (simplified) Circuit

It is not difficult to reduce the various cases to the following ones (proof done automatically):

<pre>IF (prt = 0 ∧ car = 1 ∧ clk = 1) ∨ (prt = 1 ∧ car = 0) ∨ (prt = 1 ∧ clk = 1) THEN chg := 1 ELSE chg := 0 END</pre>	<pre>IF (prt = 1 ∧ car = 1 ∧ clk = 0) ∨ (prt = 0 ∧ car = 1 ∧ clk = 1) THEN prt := 1 ELSE prt := 0 END</pre>
---	---

These cases can be further unified as follows (proof done automatically):

```

IF
  (car = 1 ∧ clk = 1) ∨
  (car = 0 ∧ prt = 1)
THEN
  chg := 1
ELSE
  chg := 0
END

```

```

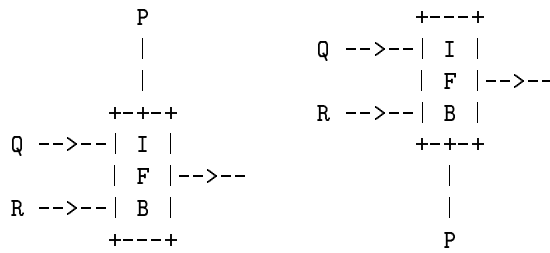
IF
  (prt = 1 ∧ ¬((car = 1 ∧ clk = 1) ∨ (car = 0 ∧ prt = 1))) ∨
  (prt = 0 ∧ ((car = 1 ∧ clk = 1) ∨ (car = 0 ∧ prt = 1)))
THEN
  prt := 1
ELSE
  prt := 0
END

```

In this last version we notice several occurrences of predicates of the form:

$$(P \wedge Q) \vee (\neg P \wedge R)$$

This will be economically represented by an IF Box (for short IFB), considered to be an “atomic” gate. Such a box is pictorially represented by either one of the following conventions:



Here is the final circuit we obtain:

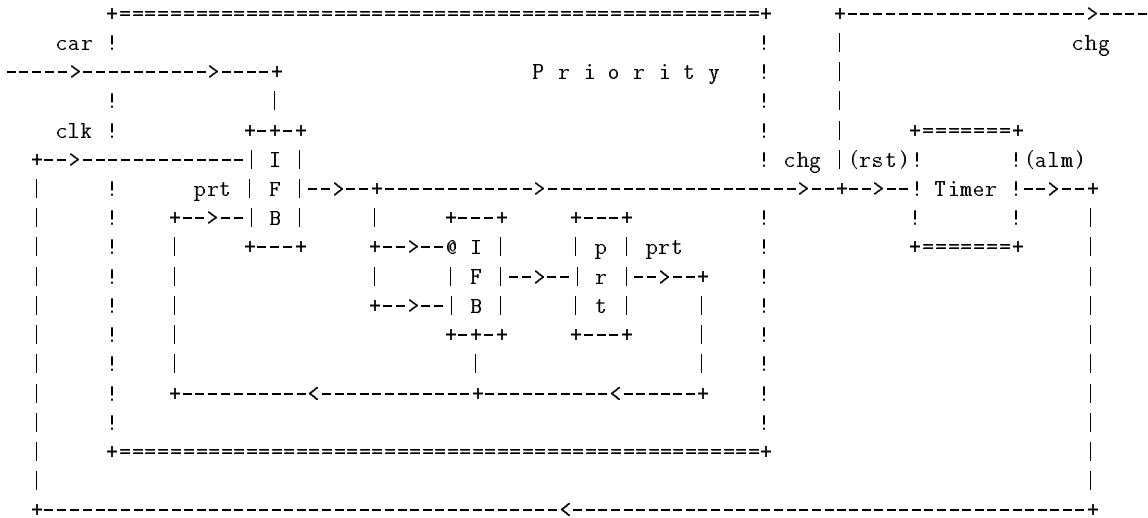


Fig. 9. The final simplified circuit

4.4 Inadequacy of the Previous (simplified) Circuit: a Discussion

The previous circuit delivers an output, chg , which, when equal to 1, indicates a necessary change of priority. Our intention is now to develop our **Light** circuit supposed to control the precise sequencing of the traffic lights. Our idea is to plug that circuit to the chg output of the previous one.

Of course, the immediate question that comes to mind is the following. Is the chg output of the **Priority** circuit an adequate input for our intended **Light** circuit? Clearly, the input we need for that circuit is one “telling” exactly when the **Light** circuit has to change the colors. Should these colors be just green and red, then the previous output would be sufficient. Unfortunately, as we know, a green light is not turning red right away, *it has first to turn orange for a little while*.

The design decision to be made at this point concerns the place where to realize this second timing: either in the **Priority** circuit or in the **Light** circuit. It seems that the latter is the most natural choice: after all, the matter only concerns the handling of the colors, which is the business of the second circuit. But there is clearly a synchronisation problem to solve between the two circuits: a new change of priority must not be signaled by the **Priority** circuit before the small “orange” delay has passed (since otherwise the **Light** circuit will get confused).

This problem seems to be “naturally” solved due to the small value of this second delay versus that of the first one. More precisely, when the priority is given to the main road, we know that it cannot be given back to the small road before the long delay has passed, hence the short delay is certainly over when this occurs. Likewise, when the priority is given to the small road, it has necessarily to be given back to the main road at the end of the long delay (same conclusion then as in the previous case). But there is another case where the priority has to be given back to the main road: this is when no cars are waiting anymore on the small road. Now the question is the following: is it possible that no cars are waiting on the small road while the light is still orange on the main road? This seems to be impossible as the very reason for the light on the main road to have turned orange to begin with was precisely because of the presence of some cars on the small road: such cars could not have disappeared by magic. Unfortunately, this might happen if drivers overpass the (still) red light on the small road (some drivers do so when there are no cars coming on the other side!) or change mind at the last moment and depart from the crossing by doing a U-turn. We must clearly preclude the priority to change in that case and wait until the light has turned red on the main road. Then and only then can the priority be turned back to the main road after an orange phase (for nothing probably) on the small road. In other words, the “orange” handling clearly retro-acts on the priority policy. The moral of the story is that the two concerns which were exhibited at the beginning of this example (priority and light) might then not be so “separate” as we pretended.

Does this conclusion ruin what we have done so far? Is our approach with two separate circuits wrong? Our opinion is to give two “no” answers to these questions. We think that the idea of having two circuits is a good one. Moreover, we think that the **Light** circuit (to be developed later) should have nothing to do with any priority policy at all: its rôle is just to manipulate the lights in a certain specific order. We have thus no choice but to reconsider the **Priority** circuit.

4.6 Reshaping the Priority Circuit

4.6.1 Revisiting the System Informal Specification

The **Priority** circuit we have developed was a bit too simplistic: its specification has to be adapted according to the discussion of the previous section. This can be done by means of the following extra informal rule:

Rule 7 The priority should stay unchanged on either roads for at least a (short) delay.

This delay corresponds to the time during which the light is orange on the side that has just lost the priority. Notice that this rule is only effective on the small road since **Rule 2** already stipulates that the priority on the main road should not change before a long delay.

4.6.2 Revisiting the Timer Circuit

In section 4.3.1 we described the specification of a Timer circuit with a simple interface made of an entry and an exit *alm*. The idea is to use that same timer, but with more capabilities. The new timer has two more exits: one called *alm2* and another one called *chg2*. The exit *alm* is now called *alm1*. Here is the new interface:

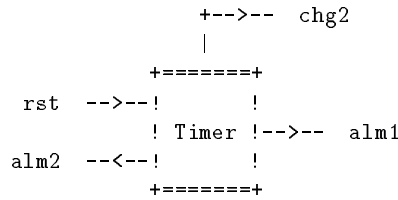


Fig. 10. The new Timer interface

The entry *rst* resets the timer counter *tck* to the constant positive value *tmp* as before. The exit *alm1* is set to 1 when the counter *tck* is null as was *alm* in the previous timer. The exit *alm2* is set to 1 when the counter *tck* is smaller than or equal to the positive constant *tmq* (supposed to be smaller than and distinct from *tmp*). When *alm2* is set to 1 this means that the short delay is over. Finally, the exit *chg2* is set to 1 when the counter *tck* is exactly equal to *tmq*. The exit *chg2* is thus a single pulse telling exactly when the short delay is over: this is clearly a desirable entry for our future circuit Light. This can be formalised as follows:

$ \begin{aligned} &tmp \in \mathbb{N} \\ &tmq \in \mathbb{N} \\ &tmq < tmp \\ &tmq > 0 \\ &rst \in \{0, 1\} \\ &alm1 \in \{0, 1\} \\ &alm2 \in \{0, 1\} \\ &chg2 \in \{0, 1\} \\ &tck \in 0 .. tmp \\ &alm1 = 1 \Leftrightarrow tck = 0 \\ &alm2 = 1 \Leftrightarrow tck \leq tmq \\ &chg2 = 1 \Leftrightarrow tck = tmq \end{aligned} $
--

The last three invariants state exactly when the exits are set depending on the value of the counter *tck*. The first event is modified as follows:

```

timer_1 ≐
SELECT
  rst = 0 ∧
  tck > 1
THEN
  IF tck ≤ tmq + 1 THEN alm2 := 1 ELSE alm2 := 0 END ||
  IF tck = tmq + 1 THEN chg2 := 1 ELSE chg2 := 0 END ||
  tck := tck - 1
END

```

As can be seen the exit *alm2* and *chg2* are set depending on appropriate values of the counter *tck*. The three other events are only slightly modified in a straightforward fashion.

```

timer_2 ≐
SELECT
  rst = 0 ∧
  tck = 1
THEN
  alm1 := 1 ||
  alm2 := 1 ||
  chg2 := 0 ||
  tck := 0
END

```

```

timer_3 ≐
SELECT
  rst = 0 ∧
  tck = 0
THEN
  alm1 := 1 ||
  alm2 := 1 ||
  chg2 := 0 ||
END

```

```

timer_4 ≐
SELECT
  rst = 1
THEN
  alm1 := 0 ||
  alm2 := 0 ||
  chg2 := 0 ||
  tck := tmp
END

```

4.6.3 Revisiting the Interface of the Priority Circuit

The new Priority circuit has the following interface:

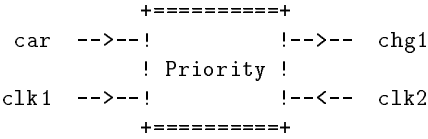


Fig. 11. The new Priority interface

As can be seen, we have changed the name of *clk* to *clk1* and added another entry *clk2* that corresponds to the timer sending an alarm as soon as the short delay is over. Connecting this new circuit with the previous extended Timer yields the combined circuits of figure 11. We now have two exits, namely *chg1* and *chg2*. They correspond to the two possible transitions. We join them with an “OR” gate as indicated, yielding the unique exit *sig* which is now the proper entry for the Light circuit.

4.6.4 Revisiting the Design of the Priority Circuit

The modification in the three priority changing events of the new Priority circuit are rather limited. We replace *clk* by *clk1* and *chg* by *chg1*, and we extend the guard of event *small_to_main_1* with the predicate *clk2 = 1*, thus only turning the priority from the small to the main road when there are no cars pending *and when the small delay is over* (this implements Rule 7).

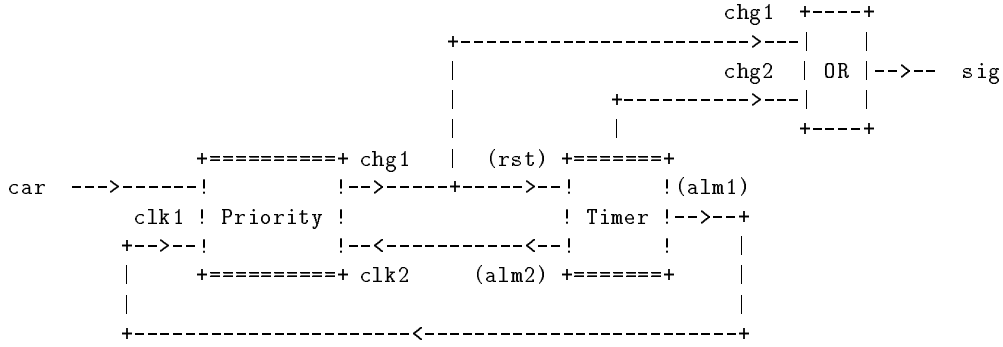
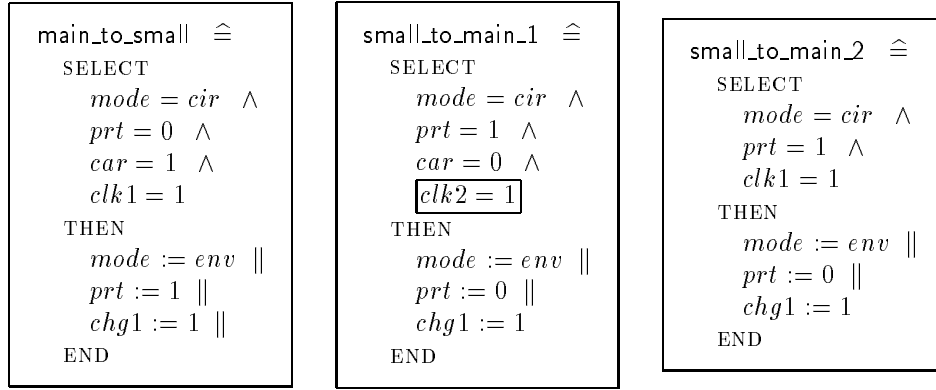
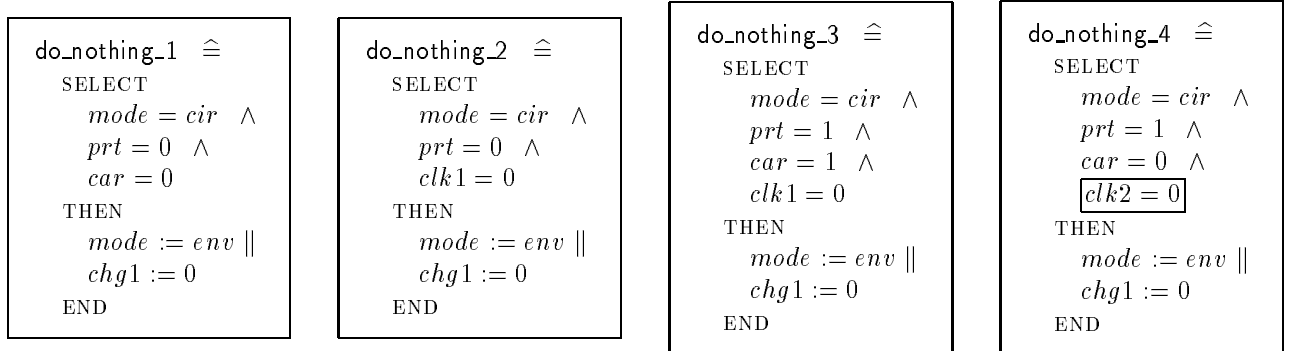


Fig. 12. Connecting the new circuits



The previous “do_nothing” events are not modified (except with *clk* replaced by *clk1*). However, a fourth event is added which is fired when the priority is on the small road, when there are no cars pending *and when the small delay has not passed yet* (this again implements Rule 7). The guard of this extra event is the “complement” of the guard of the new version of the event *small_to_main_1* (which thus proves deadlockfreeness of this new Priority circuit from that of the previous one). Here are the new events:



These cases can be unified as follows (notice the very slight change only with comparison to the simplified circuit):

```

IF
  (car = 1 ∧ clk1 = 1) ∨ (car = 0 ∧ prt = 1 ∧ clk2 = 1)
THEN
  chg := 1
ELSE
  chg := 0
END

```

```

IF
  (prt = 1 ∧ ¬((car = 1 ∧ clk1 = 1) ∨ (car = 0 ∧ prt = 1 ∧ clk2 = 1))) ∨
  (prt = 0 ∧ ((car = 1 ∧ clk1 = 1) ∨ (car = 0 ∧ prt = 1 ∧ clk2 = 1)))
THEN
  prt := 1
ELSE
  prt := 0
END

```

Notice that we must formally prove that the two outputs *chg1* and *chg2* obey the following properties: (1) they are never set to 1 simultaneously, and (2) their 1 states alternate systematically. The first property is easy to establish. The second one requires adding two (dummy) counters *ctr1* and *ctr2* that are incremented as the corresponding exit turn to 1. Then we have to prove the following extra invariant

$$ctr2 \leq ctr1 \leq ctr2 + 1$$

The corresponding proof is not difficult provided one adds the following technical invariants

$$\begin{aligned}
mode = cir \wedge chg2 = 1 &\Rightarrow ctr1 = ctr2 \\
mode = cir \wedge chg2 = 0 &\Rightarrow ctr1 = ctr2 + 1 \\
mode = env \wedge (chg1 = 0 \wedge chg2 = 1) &\Rightarrow ctr1 = ctr2 \\
mode = env \wedge (chg1 = 1 \vee chg2 = 0) &\Rightarrow ctr1 = ctr2 + 1
\end{aligned}$$

All this leads to the following final joined circuit `Priority_Timer` of figure 13

4.7 The Light Circuit

4.7.1 The Upper Circuit

We start with a simplified circuit whose rôle is to ensure the sequencing of a single traffic light. We shall further extend that circuit to handle two synchronized traffic lights. The circuit has a single boolean entry *sig* which, when valued to 1, indicates that a change in the light appearance should be performed. It has four boolean exits called *grn*, *org*, *rd1*, and *rd2*. The reason for decomposing the red color into two colors is one of symmetry. Exactly one of them at a time is valued to 1. This can be formalized as follows:


```

sig ∈ {0, 1}
grn, org ∈ {0, 1} × {0, 1}
rd1, rd2 ∈ {0, 1} × {0, 1}
grn = 1 ∨ org = 1 ∨ rd1 = 1 ∨ rd2 = 1
grn = 1 ⇒ org = 0 ∧ rd1 = 0 ∧ rd2 = 0
org = 1 ⇒ grn = 0 ∧ rd1 = 0 ∧ rd2 = 0
rd1 = 1 ⇒ org = 0 ∧ grn = 0 ∧ rd2 = 0
rd2 = 1 ⇒ org = 0 ∧ grn = 0 ∧ rd1 = 0

```

Inside the circuit, we have four boolean registers storing the value of the corresponding colors

```

GRN, ORG ∈ {0, 1} × {0, 1}
RD1, RD2 ∈ {0, 1} × {0, 1}
grn, org, rd1, rd2 = GRN, ORG, RD1, RD2

```

The events of the circuit are straightforward

```

grn_to_org ≐
SELECT
  mode = cir ∧
  sig = 1 ∧
  GRN = 1
THEN
  mode := env ||
  grn, org, rd1, rd2 := 0, 1, 0, 0 ||
  GRN, ORG, RD1, RD2 := 0, 1, 0, 0
END

```

```

org_to_rd1 ≐
SELECT
  mode = cir ∧
  sig = 1 ∧
  ORG = 1
THEN
  mode := env ||
  grn, org, rd1, rd2 := 0, 0, 1, 0 ||
  GRN, ORG, RD1, RD2 := 0, 0, 1, 0
END

```

```

rd1_to_rd2 ≐
SELECT
  mode = cir ∧
  sig = 1 ∧
  RD1 = 1
THEN
  mode := env ||
  grn, org, rd1, rd2 := 0, 0, 0, 1 ||
  GRN, ORG, RD1, RD2 := 0, 0, 0, 1
END

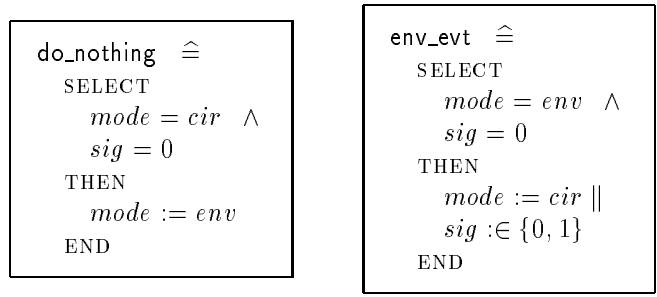
```

```

rd2_to_grn ≐
SELECT
  mode = cir ∧
  sig = 1 ∧
  RD2 = 1
THEN
  mode := env ||
  grn, org, rd1, rd2 := 1, 0, 0, 0 ||
  GRN, ORG, RD1, RD2 := 1, 0, 0, 0
END

```

We finally have a “do-nothing” event in the circuit (when *sig* is valued 0) and also an environment event assigning *sig* in a non-deterministic way. These are as follows:



The various circuit events can be unified (easy automatic proof) by the following parallel conditional assignments:

```

IF IFB(sig, RD2, GRN) THEN GRN := 1 ELSE GRN := 0 END ||
IF IFB(sig, GRN, ORG) THEN ORG := 1 ELSE ORG := 0 END ||
IF IFB(sig, ORG, RD1) THEN RD1 := 1 ELSE RD1 := 0 END ||
IF IFB(sig, RD1, RD2) THEN RD2 := 1 ELSE RD2 := 0 END
```

The output wires have similar unifications. All this yields the circuit drawn on figure 14.

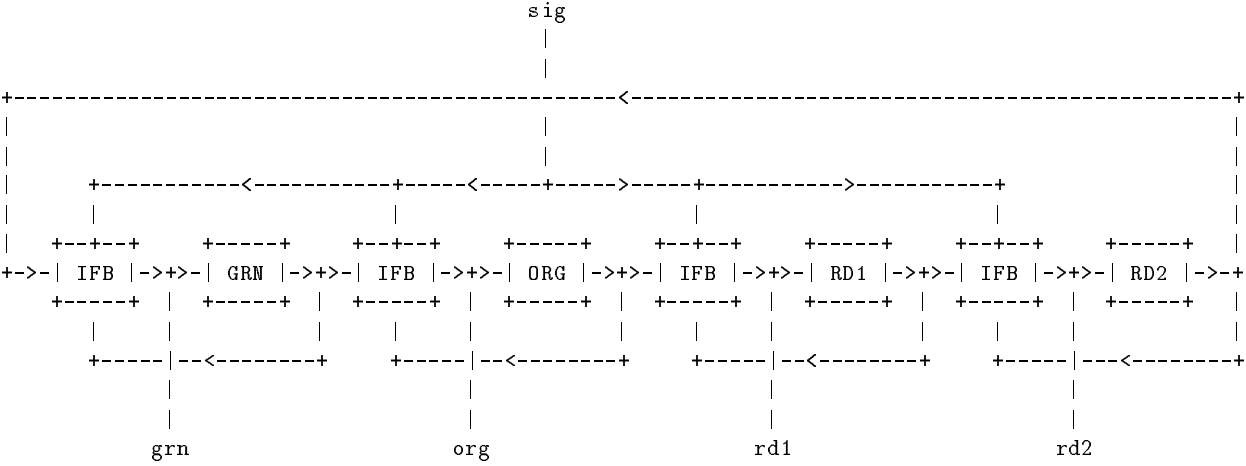


Fig. 15. The Upper circuit

4.7.1 The Lower Circuit

The Lower circuit transforms the previous outputs into proper traffic lights. The corresponding interface is on figure 15. It has the obvious meanings: the “MN” exits corresponds the lights of the main road whereas the “SM” ones are for the small road. The conditions that govern these transformation are straightforward:

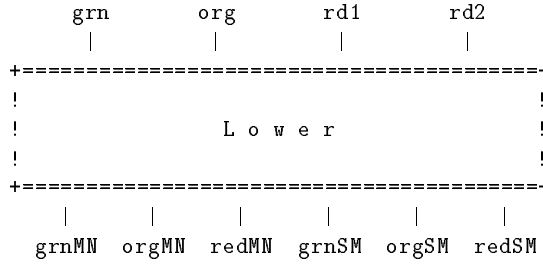


Fig. 16. The interface of the Lower circuit

$$\begin{aligned}
 grnMN &= grn \\
 orgMN &= org \\
 redMN &= 1 \Leftrightarrow (rd1 = 1 \vee rd2 = 1) \\
 grnSM &= rd1 \\
 orgSM &= rd2 \\
 redSM &= 1 \Leftrightarrow (grn = 1 \vee org = 1)
 \end{aligned}$$

From these conditions we can deduce the following fundamental safety conditions stating that when the light is orange or green on one road it is red on the other and vice-versa (this is **Rule 6**)

$$\begin{aligned}
 (grnMN = 1 \vee orgMN = 1) &\Leftrightarrow redSM = 1 \\
 (grnSM = 1 \vee orgSM = 1) &\Leftrightarrow redMN = 1
 \end{aligned}$$

We leave it to the reader to formally construct and prove that the following drawing is indeed a correct construction of this circuit: The complete circuit is drawn next page.

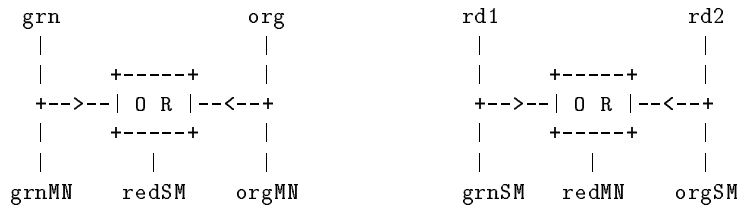


Fig. 17. The Lower circuit

5 Conclusion

In this presentation we have shown how the formal construction of little circuits can be realized and mechanically proved with the B Methods and tools. The technique for connecting circuits that we have considered in the last example is certainly one that needs more thoughts. But, already as it is, it is certainly the point of departure to envisage the construction of more complex examples.

Acknowledgements: I would like to thank J. Mermet for very interesting and illuminating discussions and suggestions.

References

1. J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press (1996).
2. J.R. Abrial. *Extending B Without Changing it (for Developing Distributed Systems)*. In H. Habrias, editor, *1st Conference on the B-Method*. November 1996.
3. J.R. Abrial and L. Mussat. *Introducing Dynamic Constraints in B*. In D. Bert editor, *B'98: Recent Advances in the Development and Uses of the B-Method*. LNCS vol 1393. 1998.
4. T. Kropf. *Formal Hardware Verification: Methods and Systems in Comparison*. LNCS State-of-the-art Survey Springer. 1991

