VIII. Development of Electronic Circuits (October 2008)

1 Introduction

In this chapter, a simple methodology supporting the *progressive proved development* of synchronous electronic circuits is presented. A typical circuit is shown on figure 1

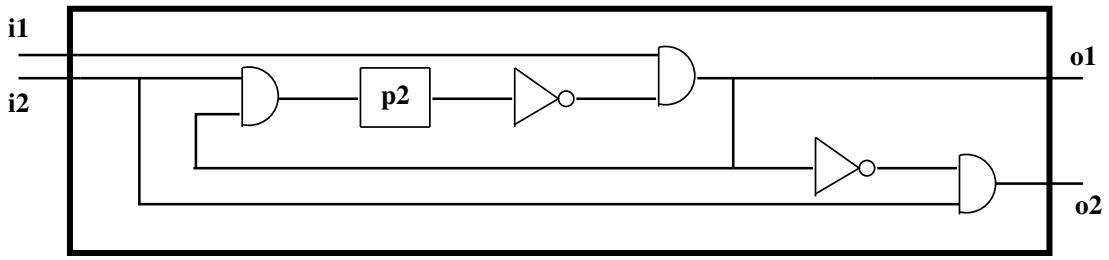

Fig. 1. A Typical Circuit

This circuit is made of the following components: two input wires $i1$ and $i2$ carrying boolean values, two output wires $o1$ and $o2$ carrying boolean values, various *gates* (here three and-gates and two not-gates), and a *register* $p2$ containing a boolean value. We would like to develop such circuits in a systematic fashion.

1.1 Synchronous Circuits

A synchronous circuit is viewed as a box which has a certain *state*, let us call this state cir_state. Some *input* lines are entering into the box, and some *output* lines are emerging out of it. Input and output lines are supposed to carry boolean values.All this is indicated on figure 2.

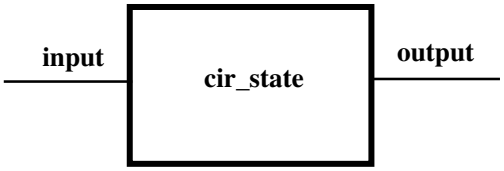

Fig. 2. A Circuit as a Box with some Input and Output Wires

As a sufficient abstraction, we can say that the circuit is *synchronized* by a clock, which pulses regularly between two alternative positions, low and $high$, as indicated on figure 3.

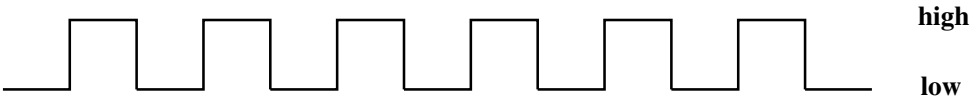

Fig. 3. A Clock

This abstraction of the clock is interpreted in the following way: (1) when the clock is low, cir_state and the $output$ line are supposed to be idle, only the $input$ line may change, and conversely (2) when the clock is $high$, the $input$ lines is supposed to stay idle whereas cir_state may be modified as well as the $output$ line. From now on, we consider that the circuit state cir_state and the output wire $output$ together form the *circuit*, whereas the input line constitutes its *environment*. Note that the environment may also comprise a state, which we call env_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 on figure 4.

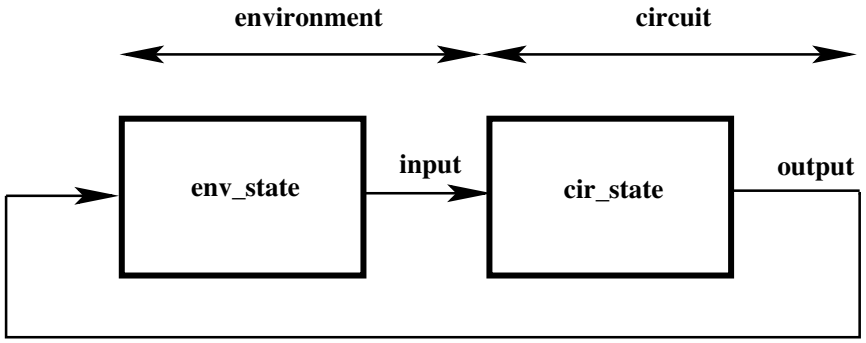

Fig. 4. A Circuit and its Environment

1.3 Dynamic View of the Coupling

Suppose cir_state is formalized by means of a number of boolean variables c. The various dynamic evolutions of the circuit can be formalized by means of a number of events defined as follows:

cir_event_i
when
$mode = cir$
$GC\_i(input, c)$
then
$mode := env$
$c, output :| PC\_i(input, c, c', output')$
end

Likewise, the environment is formalized by means of a number of variables e. The various dynamic evolutions of the environment can be formalized by means of a number of events defined as follows:

env_event_j
when
$mode = env$
$GE\_j(output, e)$
then
$mode := cir$
$e, input :| PE\_j(output, e, e', input')$
end

As can be seen, there is an important distinction to be made between the way the $input$ line and environment variables e are modified when the $mode$ is env, and the $output$ line and circuit variables c are modified when the $mode$ is cir. The modification of the environment may follow some specific rules but in no case is it influenced by the circuit variables (it may be influenced by the $output$ however). Conversely, the modification in the circuit may depend on the $input$ line and on the circuit variables c but not on the environment variables e however.

Also notice that, in an abstract view of our circuit and environment, the status of the $input$ and $output$ lines and of c and e are not necessarily represented by boolean values (which will probably be the case in a refined implementation). For instance, in an abstract specification, variables e and c 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 *independent* 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.

$$mode = env \Rightarrow C(e, input, c, output)$$
$$mode = cir \ \Rightarrow D(e, input, c, output)$$

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:

$$C(e, input, c, output)$$
$$GE\_j(output, e)$$
$$PE\_j(output, e, e', input')$$
$$\Rightarrow$$
$$D(e', input', c, output))$$

$$D(e, input, c, output)$$
$$GC\_i(input, c)$$
$$PC\_i(input, c, c', output')$$
$$\Rightarrow$$
$$C(e, input, c', output')$$

Informally, this means that when $mode$ is env and the static condition C holds, then D must hold after any accepted modifications e' 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 c' and $output'$ made by the circuit.

1.6 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 state 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 its state only, whereas the circuit modifies its state 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.

1.7 Final Construction of the Circuit

A final refinement situation is obtained when the following conditions hold:

1. the circuit variables must all be boolean,
2. the inputs must be boolean,
3. the outputs must be boolean,
4. the circuit must be deadlock free,
5. the circuit must be internally deterministic: this concerns circuit variables and outputs,
6. the circuit must be externally deterministic: circuit guards are mutually exclusive,
7. the environment does not access the circuit variables except the output
8. the circuit does not access the environment variables except the input.

Note that the environment might be still externally as well as internally non-deterministic. As a result, a circuit event has the following shape:

cir_event_i
when
$mode = cir$
$GC\_i(input, c)$
then
$mode := env$
$c := C\_i(input, c)$
$output := O\_i(input, c)$
end

We are going to prove now that each circuit event can be refined in such a way that they all have the *same action* on the circuit state and output. Here is one of these refinement:

cir_event_i
when
$mode = cir$
$GC\_i(input, c)$
then
$mode := env$
$c := \text{bool}\left(\begin{array}{l}\ldots \quad \vee \\ (\, GC\_i(input, c) \ \wedge \ C\_i(input, c) = \text{TRUE}\,) \quad \vee \\ \ldots\end{array}\right)$
$output := \text{bool}\left(\begin{array}{l}\ldots \quad \vee \\ (\, GC\_i(input, c) \ \wedge \ O\_i(input, c) = \text{TRUE}\,) \quad \vee \\ \ldots\end{array}\right)$
end

Notice our usage of the operator "bool" transforming a predicate into a boolean expression. It is defined by means of the following equivalence:

$$E = \text{bool}(P) \ \Leftrightarrow \ \left(\begin{array}{rcl} P & \Rightarrow & E = \text{TRUE} \\ \neg\, P & \Rightarrow & E = \text{FALSE}\end{array}\right)$$

The refinement proof is now straightforward. It amounts to proving the following concerning variable c (the proof to be done concerning $output$ is similar and thus not shown):

$$\begin{array}{l} GC\_i(input, c) \\ \vdash \\ C\_i(input, c) \ = \ \text{bool}\left(\begin{array}{l}\ldots \quad \vee \\ (\, GC\_i(input, c) \ \wedge \ C\_i(input, c) = \text{TRUE}\,) \quad \vee \\ \ldots\end{array}\right)\end{array}$$

According to the definition of operator bool this reduces to proving two statements. Here is the first of them:

$$\begin{array}{l} GC\_i(input, c) \\ \left(\begin{array}{l}\ldots \quad \vee \\ (\, GC\_i(input, c) \ \wedge \ C\_i(input, c) = \text{TRUE}\,) \quad \vee \\ \ldots\end{array}\right) \\ \vdash \\ C\_i(input, c) = \text{TRUE}\end{array}$$

Thanks to the mutual exclusion of the guards (that is $GC\_i(input, c) \Rightarrow \neg\, GC\_j(input, c)$ when $i \neq j$), this first statement reduces to the following which holds trivially:

$$\begin{array}{l} GC\_i(input, c) \\ C\_i(input, c) = \text{TRUE} \\ \vdash \\ C\_i(input, c) = \text{TRUE} \end{array}$$

Here is now the second statement:

$$\begin{array}{l} GC\_i(input, c) \\ \neg \left(\begin{array}{ll} \ldots & \vee \\ (\, GC\_i(input, c) \ \wedge \ C\_i(input, c) = \text{TRUE}\,) & \vee \\ \ldots & \end{array} \right) \\ \vdash \\ C\_i(input, c) = \text{FALSE} \end{array}$$

By applying de Morgan law to remove the external negation, this second statement is equivalent to the following:

$$\begin{array}{l} GC\_i(input, c) \\ \ldots \\ \neg\, GC\_i(input, c) \ \vee \ C\_i(input, c) = \text{FALSE} \\ \ldots \\ \vdash \\ C\_i(input, c) = \text{FALSE} \end{array}$$

that is the following which holds trivially:

$$\begin{array}{l} GC\_i(input, c) \\ \ldots \\ C\_i(input, c) = \text{FALSE} \\ \ldots \\ \vdash \\ C\_i(input, c) = \text{FALSE} \end{array}$$

Since the circuit events are deadlock free (disjunction of guards holds under condition $mode = cir$) and have identical actions, they can all be merged into a single event as follows:

cir_event
when
$mode = cir$
then
$mode := env$
$$c := \text{bool} \left(\begin{array}{ll} \ldots & \vee \\ GC\_i(input, c) \ \wedge \ C\_i(input, c) = \text{TRUE} & \vee \\ \ldots & \end{array} \right)$$
$$output := \text{bool} \left(\begin{array}{ll} \ldots & \vee \\ GC\_i(input, c) \ \wedge \ O\_i(input, c) = \text{TRUE} & \vee \\ \ldots & \end{array} \right)$$
end

Notice that when $C_i(input, c)$ is syntactically equal to TRUE then $C_i(input, c) = \text{TRUE}$ can be removed, and when $C_i(input, c)$ is syntactically equal to FALSE then $GC\_i(input, c) \wedge C_i(input, c) = \text{TRUE}$ can be removed. We have similar simplifications for $O\_i(input, c)$. This last event *is* our circuit. From this, the circuit can be drawn in a systematic fashion.

1.8 A Very Small Illustrating Example

Suppose we end up a development with the following circuit events:

env1
when
$mode = env$
$input\_1 = \text{TRUE}$
$input\_2 = \text{TRUE}$
then
$mode := cir$
$output := \text{TRUE}$
end

env2
when
$mode = env$
$input\_1 = \text{TRUE}$
$input\_2 = \text{FALSE}$
then
$mode := cir$
$output := \text{TRUE}$
end

env3
when
$mode = env$
$input\_1 = \text{FALSE}$
$input\_2 = \text{TRUE}$
then
$mode := cir$
$output := \text{TRUE}$
end

env4
when
$mode = env$
$input\_1 = \text{FALSE}$
$input\_2 = \text{FALSE}$
then
$mode := cir$
$output := \text{FALSE}$
end

Clearly, these events are internally as well as externally deterministic and also deadlockfree. By applying the merging rule presented in the previous section, we obtain the following for the assignment of variable $output$:

$$\text{bool}\left(\begin{array}{l}(\,input\_1 = \text{TRUE} \;\wedge\; input\_2 = \text{TRUE} \;\wedge\; \text{TRUE} = \text{TRUE}\,) \;\vee \\ (\,input\_1 = \text{TRUE} \;\wedge\; input\_2 = \text{FALSE} \;\wedge\; \text{TRUE} = \text{TRUE}\,) \;\vee \\ (\,input\_1 = \text{FALSE} \;\wedge\; input\_2 = \text{TRUE} \;\wedge\; \text{TRUE} = \text{TRUE}\,) \;\vee \\ (\,input\_1 = \text{FALSE} \;\wedge\; input\_2 = \text{FALSE} \;\wedge\; \text{FALSE} = \text{TRUE}\,)\end{array}\right)$$

reducing as expected to:

$$\text{bool}(\,input\_1 = \text{TRUE} \;\vee\; input\_2 = \text{TRUE}\,)$$

As a result, these events can be merged into the following unique event:

or_gate
when
$mode = env$
then
$mode := cir$
$output := \text{bool}\,(\,input\_1 = \text{TRUE} \;\vee\; input\_2 = \text{TRUE}\,)$
end

2 A First 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 Informal Specification

The circuit we propose to study is a well-known benchmark that has been analyzed in different contexts: it is called the *Single Pulser* (Pulser for short). Here is a first informal specification taken from [1]:

> *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 [1], 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).*

A 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.

A schematic representation of this closed system is shown on figure 5.

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, figure 6 shows two wave diagrams: the first one represents a succession of depressions of the button followed by subsequent releases, while the second shows various corresponding flashes of the lamp:

As can be seen, the flash can be situated just after a button depression, or in between a depression and a subsequent release, or else just after a release.

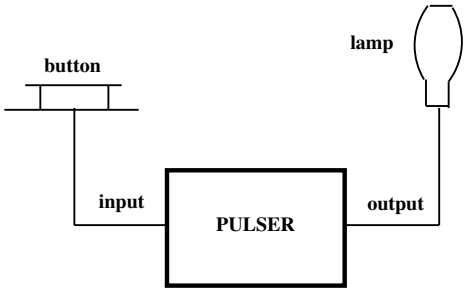

Fig. 5. A Pulser and its Environment

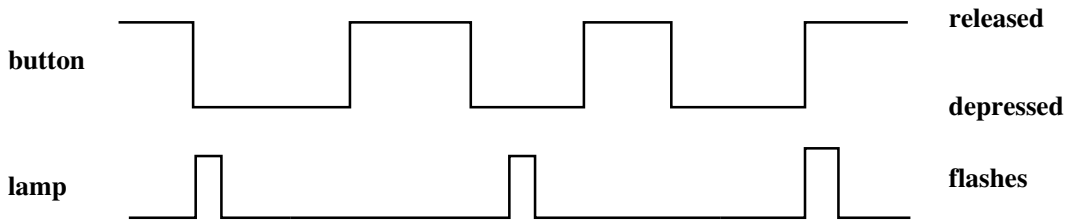

Fig. 6. Relationship Between the Button Depression and the Lamp Flash

2.2 Initial Model

The State. Before defining the state, we must formalize the set $MODE$ and its two values env and cir:

sets: $MODE$

constants: env, cir

axm0_1: $MODE = \{env, cir\}$
axm0_2: $env \neq cir$

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 invariants, stating quite naturally that $push$ is at least as pop and at most one more than pop:

variables: $mode$
$push$
pop

inv0_1: $mode \in MODE$
inv0_2: $push \in \mathbb{N}$
inv0_3: $pop \in \mathbb{N}$
inv0_4: $pop \leq push$
inv0_5: $push \leq pop + 1$

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 $flash$. In other words, you push the button then the lamp later flashes (the lamp being turned down when the circuit is started):

variables: $\ldots, flash$

inv0_6: $flash \in \mathbb{N}$

inv0_7: $flash \leq push$

inv0_8: $push \leq flash + 1$

The Events. Besides the initialization event, 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 equal to $push$, and we can release it when $push$ is different from pop (it is then one more than pop according to invariants **inv0_4** and **inv0_5**), finally we can do nothing in all circumstances:

init
$mode := env$
$push := 0$
$pop := 0$
$flash := 0$

env1
when
$mode = env$
$pop = push$
then
$mode := cir$
$push := push + 1$
end

env2
when
$mode = env$
$push \neq pop$
then
$mode := cir$
$pop := pop + 1$
end

env3
when
$mode = env$
then
$mode := cir$
end

The dynamics of the abstract circuit is a little more complicated . There are two events corresponding to flashing the lamp (event cir1) or doing nothing (event cir2). We can flash the lamp when $push$ is different from $flash$:

cir1
when
$mode = cir$
$push \neq flash$
then
$mode := env$
$flash := flash + 1$
end

The circumstances in which the circuit does nothing need to be studied carefully. When the button is depressed, the flash of the lamp can be done either immediately (case 1) or later as indicated in figure 7 (case 2 and 3).The latest time for the flash occurrence is just after the user releases the button (case 3). As a consequence, the circuit can do nothing in three different circumstances denoted A, B and C in the figure 8.

Conditions A, B, and C can be formalized more rigorously as follows:

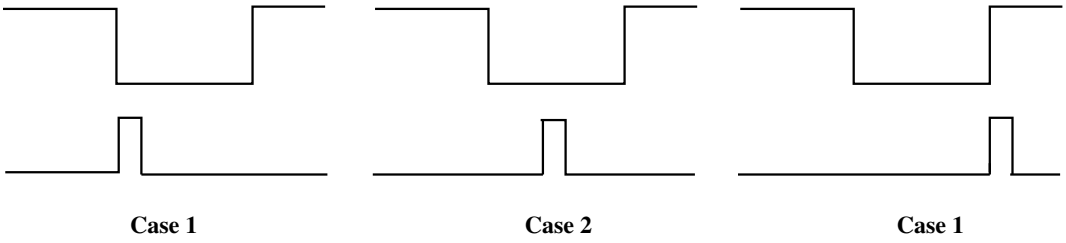

Fig. 7. The Various Cases Where the Circuit does Nothing

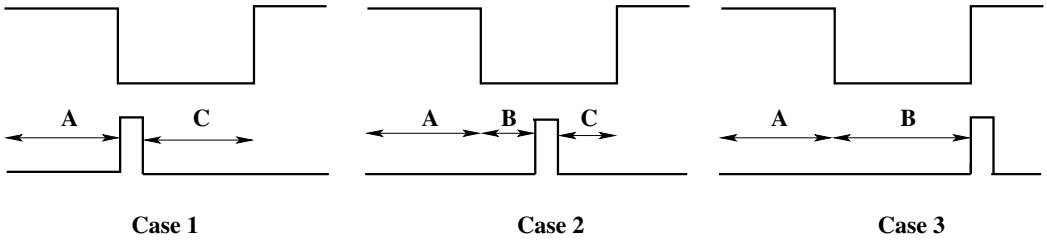

Fig. 8. The Various Conditions Where the Circuit does Nothing

Condition A: $push = pop \ \wedge \ push = flash$

Condition B: $push \neq pop \ \wedge \ push \neq flash$

Condition C: $push \neq pop \ \wedge \ push = flash$

The guard of the "do-nothing" event of the circuit corresponds to the disjunction of these conditions, namely:

$$A \ \vee \ B \ \vee \ C \ \Leftrightarrow \ push \neq pop \ \vee \ push = flash$$

cir2
when
$mode = cir$
$push \neq pop \ \vee \ push = flash$
then
$mode := env$
end

Proofs. The proof of consistency between the static properties and the events requires introducing the following additional invariant:

inv0_9: $mode = env \ \Rightarrow \ flash = push \ \vee \ flash = pop$

It may seem at first glance that the disjunction $flash = push \vee flash = pop$ is always true (even when $mode = cir$). In fact, it is almost always the case, except when the flash occurs at the latest as indicated in figure 9.

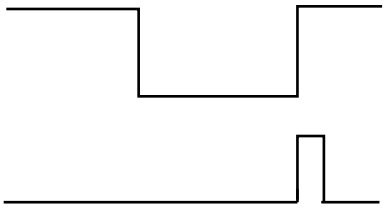

Fig. 9. The special case where $flash = push \ \vee \ flash = pop$ does not hold

Then just after the occurrence of event env2 (releasing the button) we have $mode = cir$ and $push = pop = flash + 1$, thus clearly $flash = push \ \vee \ flash = pop$ does not hold.

2.3 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 of its possible non-deterministic behaviors. Let us reconsider the two events of our circuit:

cir1
when
$mode = cir$
$push \neq flash$
then
$mode := env$
$flash := flash + 1$
end

cir2
when
$mode = cir$
$push \neq pop \ \vee \ push = flash$
then
$mode := env$
end

The guards, clearly, may overlap when $push \neq flash$ and $push \neq pop$ hold simultaneously. This occurs when we are in between a depression and a release ($push \neq pop$) and when the flash has not yet occurred ($push \neq flash$): in this situation, it is possible for the circuit to either flash the lamp or do nothing.

We can see that there are two different ways of making this system of events deterministic: (1) by replacing the guard of cir2 by $push = flash$, or (2) by adding the guard $push = pop$ to that of the event cir1. In both cases we are strengthening the guards. The net effect, in both cases, is to make each guard the negation of the other: the circuit has become deterministic indeed. The first solution, which we call PULSER1, corresponds to flashing the lamp as early as possible. It is illustrated on figure 10.

cir1_PULSER1
when
$mode = cir$
$push \neq flash$
then
$mode := env$
$flash := flash + 1$
end

cir2_PULSER1
when
$mode = cir$
$push = flash$
then
$mode := env$
end

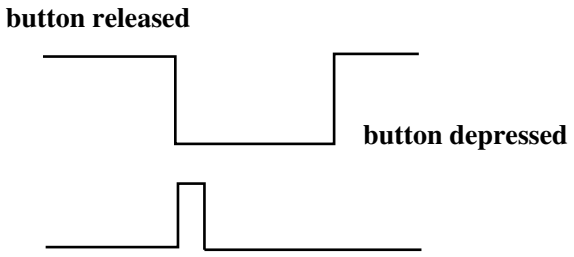

Fig. 10. The flash occurs as early as possible

In this case, the following invariant can be proved:

inv1_pulser1: $pop \neq push \ \wedge \ mode = env \ \Rightarrow \ flash \neq pop$

When the button is depressed ($pop \neq push$) and the mode is environment ($mode = env$) then the flash has occurred, thus the number of flashes is one more than the number of pops, or alternatively the flash number is equal to the push number ($flash = push$).

The second solution, which we call PULSER2, corresponds to flashing the lamp as late as possible. It is illustrated on figure 11.

cir1_PULSER2
when
$mode = cir$
$push = pop \ \wedge \ push \neq flash$
then
$mode := env$
$flash := flash + 1$
end

cir2_PULSER2
when
$mode = cir$
$push \neq pop \ \vee \ push = flash$
then
$mode := env$
end

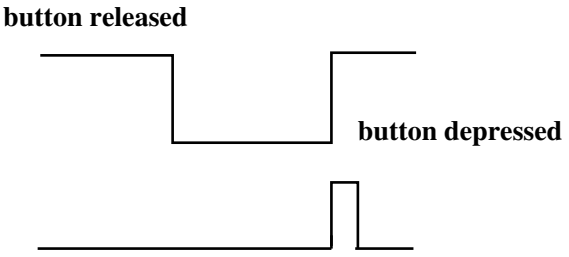

Fig. 11. The flash occurs as late as possible

In this case, the following invariant can be proved:

inv1_pulser2: $pop \neq push \Rightarrow flash \neq push$

When the button is depressed ($pop \neq push$) then the number of flashes is one less than the number of pushes ($flash \neq push$) or alternatively the flash number is equal to the pop number ($flash = pop$).

2.4 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 wires should be defined, and the abstract variables $push$, 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 $input$ and $output$, which corresponds to the input and output lines respectively. These variables are boolean.

variables: $mode$
$input$
$output$

inv2_1: $input \in \text{BOOL}$

inv2_2: $output \in \text{BOOL}$

The variable $input$ is an environment variables: it is modified by both events env1 and env2. The abstract variable $push$ is supposed to denote the number of times the variable $input$ moves from FALSE to TRUE. Likewise the abstract variable pop is supposed to denote the number of times the variable $input$ moves from TRUE to FALSE. This leads to the following new events env1 and env2:

env1
when
$mode = env$
$input = \text{FALSE}$
then
$mode := cir$
$input := \text{TRUE}$
end

env2
when
$mode = env$
$input = \text{TRUE}$
then
$mode := cir$
$input := \text{FALSE}$
end

env3
when
$mode = env$
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:

(abstract-)env1
when
$mode = env$
$pop = push$
then
$mode := cir$
$push := push + 1$
end

(abstract-)env2
when
$mode = env$
$pop \neq push$
then
$mode := cir$
$pop := pop + 1$
end

(abstract-)env3
when
$mode = env$
then
$mode := cir$
end

The correct refinement thus clearly involves proving the following relationship between the concrete environment space and the abstract one:

inv2_3: $input = \text{TRUE} \Leftrightarrow pop \neq push$

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

(abstract-)cir1_PULSER1
when
$mode = cir$
$push \neq flash$
then
$mode := env$
$flash := flash + 1$
end

(abstract-)cir2_PULSER1
when
$mode = cir$
$push = flash$
then
$mode := env$
end

The abstract circuit variable $flash$ has to disappear. It counts the number of time the concrete variable $output$ moves from FALSE to TRUE. 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 $input$ moves from FALSE to TRUE. Clearly, we can access the actual value of $input$, but certainly *not its previous value*. We have no choice then but to introduce a register, reg, internal to our circuit, and whose role is to store the previous value of $input$. We also have an equality between reg and $input$ when $mode = env$ holds (**invi2_5**):

variables: $mode, input, output, reg$

inv2_4: $reg \in \text{BOOL}$

invi2_5: $mode = env \Rightarrow reg = input$

This leads to the following implementation of the events cir1 and cir2 for PULSER1:

cir1_PULSER1
when
$mode = cir$
$input = \text{TRUE} \wedge reg = \text{FALSE}$
then
$mode := env$
$output := \text{TRUE}$
$reg := input$
end

cir2_PULSER1
when
$mode = cir$
$input = \text{FALSE} \vee reg = \text{TRUE}$
then
$mode := env$
$output := \text{FALSE}$
$reg := input$
end

The concrete guards must imply the abstract ones.. All this leads to the following properties to be maintained:

inv2_PULSER1_6: $mode = cir \Rightarrow \begin{pmatrix} input = \text{TRUE} \wedge reg = \text{FALSE} \\ \Leftrightarrow \\ push \neq flash \end{pmatrix}$

We have a similar implementation of the events cir1 and cir2 for PULSER2:

cir1_PULSER2
when
$mode = cir$
$input = \text{FALSE} \wedge reg = \text{TRUE}$
then
$mode := env$
$output := \text{TRUE}$
$reg := input$
end

cir2_PULSER2
when
$mode = cir$
$input = \text{TRUE} \vee reg = \text{FALSE}$
then
$mode := env$
$output := \text{FALSE}$
$reg := input$
end

And we have to ensure the following additional invariant:

inv2_PULSER2_6: $mode = cir \Rightarrow \begin{pmatrix} input = \text{FALSE} \wedge reg = \text{TRUE} \\ \Leftrightarrow \\ push \neq flash \wedge push = pop \end{pmatrix}$

2.5 Building the Final Circuits

Our next design step is to depart from the closed system and consider the circuit PULSER1 and PULSER2 in isolation. Here is a copy of the PULSER1 events:

cir1_PULSER1
when
$mode = cir$
$input = \text{TRUE} \wedge reg = \text{FALSE}$
then
$mode := env$
$output := \text{TRUE}$
$reg := input$
end

cir2_PULSER1
when
$mode = cir$
$input = \text{FALSE} \vee reg = \text{TRUE}$
then
$mode := env$
$output := \text{FALSE}$
$reg := input$
end

Applying the technique developed in section 1.7, we obtain:

PULSER1
when
$mode = cir$
then
$mode := env$
$output := \text{bool}((input = \text{TRUE} \ \wedge \ reg = \text{FALSE} \ \wedge \ \text{TRUE} = \text{TRUE}) \ \vee$
$(\ldots \ \wedge \ \text{FALSE} = \text{TRUE}))$
$reg := \text{bool}(input = \text{TRUE} \ \wedge \ (input = \text{TRUE} \ \wedge \ reg = \text{FALSE}) \ \vee$
$input = \text{TRUE} \ \wedge \ (input = \text{FALSE} \ \vee \ reg = \text{TRUE}))$
end

which reduces to the following:

PULSER1
when
$mode = cir$
then
$mode := env$
$output := \text{bool}(input = \text{TRUE} \ \wedge \ reg = \text{FALSE})$
$reg := \text{bool}(input = \text{TRUE})$
end

We have eventually constructed our little circuit PULSER1 as shown on figure 12.

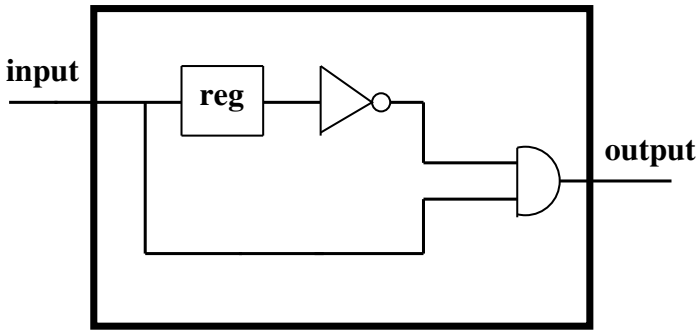

Fig. 12. The circuit PULSER1

We can construct the following circuit PULSER2 in a similar fashion:

cir1_PULSER2
when
$mode = cir$
$input = \text{FALSE} \ \wedge \ reg = \text{TRUE}$
then
$mode := env$
$output := \text{TRUE}$
$reg := input$
end

cir2_PULSER2
when
$mode = cir$
$input = \text{TRUE} \ \vee \ reg = \text{FALSE}$
then
$mode := env$
$output := \text{FALSE}$
$reg := input$
end

Applying the technique developed in section 1.7, we obtain:

PULSER2
when
$mode = cir$
then
$mode := env$
$output := \text{bool}((input = \text{FALSE} \ \wedge \ reg = \text{TRUE} \ \wedge \ \text{TRUE} = \text{TRUE}) \ \vee$
$(\ldots \ \wedge \ \text{FALSE} = \text{TRUE}))$
$reg := \text{bool}(input = \text{TRUE} \ \wedge \ (input = \text{FALSE} \ \wedge \ reg = \text{TRUE}) \ \vee$
$input = \text{TRUE} \ \wedge \ (input = \text{TRUE} \ \vee \ reg = \text{FALSE}))$
end

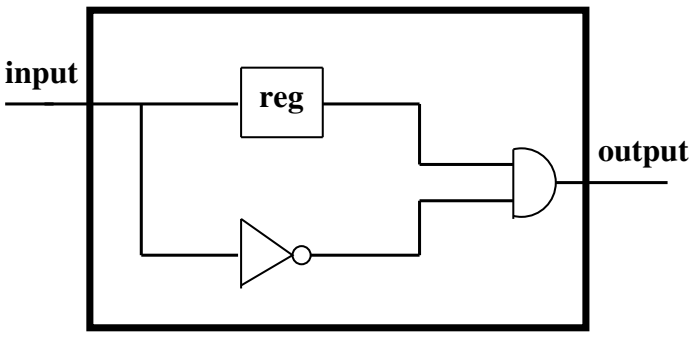

Fig. 13. The circuit PULSER2

which reduces to:

PULSER2
when
$mode = cir$
then
$mode := env$
$output := \text{bool}(input = \text{FALSE} \ \wedge \ reg = \text{TRUE})$
$reg \quad := \text{bool}(input = \text{TRUE})$
end

This lead to the circuit of figure 13.

3 Second Example: the Arbiter

3.1. Informal Specification

This simple circuit is called the (binary) $Arbiter$. It has two boolean input lines called i_1 and i_2 and two boolean output lines called o_1 and o_2. This is indicated on figure 14.

Fig. 14. The Arbiter

The circuit has two boolean inputs $i1$ and $i2$ and two boolean outputs $o1$ and $o2$	FUN-1

When input i_i is valued to TRUE, this means that a certain $user\_i$, associated by construction with the line i_i, has required (asked for) the usage of a certain shared resource (the specific resource in question as well as the nature of the users do not play any rôle in this system).

A TRUE input means a user (associated with that input) is asking for a certain resource	FUN-2

When the circuit, used with input i_i valued to TRUE, reacts with the output o_i valued to TRUE, this means that the circuit has indeed granted the resource to $user\_i$. Of course, an output o_i can only be valued to TRUE when the corresponding input i_i is TRUE.

The circuit reacts positively to a request by setting the corresponding output to TRUE	FUN-3

Conversely, the circuit should react as soon as it can. But this reaction is constrained by the fact that the circuit can only grant the resource to *at most one user* at a time.

The circuit can react positively to one request only at a time (mutual exclusion)	FUN-4

Notice that each winning user is supposed to immediately release the resource so that it can ask for it again immediately after getting it.

Each user frees the resource immediately	FUN-5

We have a number of additional constraints:

– No requiring user can be indefinitely denied the right to obtain the resource (this could be the case, should the other user always requires the resource again immediately after getting it). Notice that in this example, we shall make this constraint more precise by asking that a requiring user should not wait for more than one clock pulse before being served. In other words, a new requesting user, if not served at the next circuit reaction, must necessarily be served at the one that follows the next.

A requesting user cannot be postponed indefinitely	FUN-6

– We suppose that a requiring user shall not give up requiring the resource without being served (this is just a simplification that could have been relaxed).

A user asking for a resource comtinues to ask for it as long as it is not served	FUN-7

– Finally, we require that the circuit correctly reacts to the void case where no user is asking for the resource: in that case, the resource must not then be granted to any user.

The resource cannot be granted without a user asking for it	FUN-8

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.

3.1 Initial Model

The State. 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 requests 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 (**inv0_5** to **inv0_8**):

variables: $r1$, $r2$, $a1$, $a2$

inv0_1: $r1 \in \mathbb{N}$

inv0_2: $r2 \in \mathbb{N}$

inv0_3: $a1 \in \mathbb{N}$

inv0_4: $a2 \in \mathbb{N}$

inv0_5: $a1 \leq r1$

inv0_6: $r1 \leq a1 + 1$

inv0_7: $a2 \leq r2$

inv0_8: $r2 \leq a2 + 1$

We have not yet stated however that no user must wait indefinitely. For this, we introduce two boolean variables in the circuit: $p1$ and $p2$. When, say, pi is TRUE it means that $user\_i$ now waits for the resource. Clearly, $p1$ and $p2$ cannot be both equal to TRUE simultaneously (**inv011**) because that would mean that the circuit has not reacted immediately. In fact, when $mode$ is env, $pi = \text{FALSE}$ is equivalent to $ri = ai$: no request is pending for $user\_i$ (**inv0_12** and **inv0_13**):

variables: $p1$, $p2$

inv0_9: $p1 \in \text{BOOL}$

inv0_10: $p2 \in \text{BOOL}$

inv0_11: $p1 = \text{FALSE} \vee p2 = \text{FALSE}$

inv0_12: $mode = env \Rightarrow (r1 = a1 \Leftrightarrow p1 = \text{FALSE})$

inv0_13: $mode = env \Rightarrow (r2 = a2 \Leftrightarrow p2 = \text{FALSE})$

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

env1
when
$mode = env$
$r1 = a1$
then
$mode := cir$
$r1 := r1 + 1$
end

env2
when
$mode = env$
$r2 = a2$
then
$mode := cir$
$r2 := r2 + 1$
end

env3
when
$mode = env$
$r1 = a1$
$r2 = a2$
then
$mode := cir$
$r1 := r1 + 1$
$r2 := r2 + 1$
end

env0
when
$mode = env$
then
$mode := cir$
end

The events of the circuit are very simple. In case a request is pending (in events cir1 and cir2), the event increments the acknowledgement counter and set the corresponding variables, say $p1$ for event cir1, to FALSE. Notice that it can be the case already. It does so, however, provided the other user has not itself required the resource for more than one clock pulse: hence the guard $p2 = \text{FALSE}$. When no request is made (in event cir0), the event does nothing except setting $p1$ and $p2$ to FALSE:

cir1
when
$mode = cir$
$r1 \neq a1$
$p2 = \text{FALSE}$
then
$mode := env$
$a1 := a1 + 1$
$p1 := \text{FALSE}$
$p2 := \text{bool}(r2 \neq a2)$
end

cir2
when
$mode = cir$
$r2 \neq a2$
$p1 = \text{FALSE}$
then
$mode := env$
$a2 := a2 + 1$
$p2 := \text{FALSE}$
$p1 := \text{bool}(r1 \neq a1)$
end

cir0
when
$mode = cir$
$r1 = a1$
$r2 = a2$
then
$mode := env$
$p1 := \text{FALSE}$
$p2 := \text{FALSE}$
end

Proving Deadlock Freedom. Nothing guarantees, of course, that the circuit events are not stuck because their guards do not hold. We have thus to prove the following, stating that while in the cir mode, the disjunction of the guards of the circuit always holds:

thm0_1: $$mode = cir \Rightarrow \begin{pmatrix} r1 \neq a1 \ \wedge \ p2 = \text{FALSE} \ \vee \\ r2 \neq a2 \ \wedge \ p1 = \text{FALSE} \ \vee \\ r1 = a1 \ \wedge \ r2 = a2 \end{pmatrix}$$

For proving this, it is necessary to add the following invariants:

inv0_14: $mode = cir \Rightarrow (r1 = a1 \Rightarrow p1 = \text{FALSE})$

inv0_15: $mode = cir \Rightarrow (r2 = a2 \Rightarrow p2 = \text{FALSE})$

Note that the circuit is still non-deterministic: this is the case when both users are just requiring the resource simultaneously (thus $p1 = \text{FALSE}$ and $p2 = \text{FALSE}$ hold simultaneously). In this case, both events cir1 and cir2 can be fired.

3.2 First Refinement: Generating Binary 0outputs from the Circuit

The State. In the previous section, 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 an offset (that is a 0 or a 1), the proper incrementation itself being done by the environment on two slightly time-shifted counters, say $b1$ and $b2$. But we want the circuit to produce boolean values only. For this, we introduce a constant function b_2_01 transforming a boolean value into a numeric value.

constants: b_2_01

axm1_1: $b\_2\_01 \in \text{BOOL} \rightarrow \{0, 1\}$

axm1_2: $b\_2\_01(\text{TRUE}) = 1$

axm1_3: $b\_2\_01(\text{FALSE}) = 0$

This refinement introduces thus four variables typed as follows:

variables: $b1, o1, b2, o2$

inv1_1: $b1 \in \mathbb{N}$

inv1_2: $o1 \in \text{BOOL}$

inv1_3: $b2 \in \mathbb{N}$

inv1_4: $o2 \in \text{BOOL}$

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

inv1_5: $mode = cir \Rightarrow a1 = b1$

inv1_6: $mode = cir \Rightarrow a2 = b2$

inv1_7: $mode = env \Rightarrow a1 = b1 + b\_2\_01(o1)$

inv1_8: $mode = env \Rightarrow a2 = b2 + b\_2\_01(o2)$

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 Events. The environment events are all modified in a straightforward way:

env1
when
$mode = env$
$r1 = b1 + b\_2\_01(o1)$
then
$mode := cir$
$r1 := r1 + 1$
$b1 := b1 + b\_2\_01(o1)$
$b2 := b2 + b\_2\_01(o2)$
end

env2
when
$mode = env$
$r2 = b2 + b\_2\_01(o2)$
then
$mode := cir$
$r2 := r2 + 1$
$b1 := b1 + b\_2\_01(o1)$
$b2 := b2 + b\_2\_01(o2)$
end

env3
when
$mode = env$
$r1 = b1 + b\_2\_01(o1)$
$r2 = b2 + b\_2\_01(o2)$
then
$mode := cir$
$r1 := r1 + 1$
$r2 := r2 + 1$
$b1 := b1 + b\_2\_01(o1)$
$b2 := b2 + b\_2\_01(o2)$
end

The circuit events are modified accordingly:

cir1
when
$mode = cir$
$r1 \neq b1$
$p2 = \text{FALSE}$
then
$mode := env$
$o1 := \text{TRUE}$
$o2 := \text{FALSE}$
$p1 := \text{FALSE}$
$p2 := \text{bool}(r2 \neq b2)$
end

cir2
when
$mode = cir$
$r2 \neq b2$
$p1 = \text{FALSE}$
then
$mode := env$
$o1 := \text{FALSE}$
$o2 := \text{TRUE}$
$p1 := \text{bool}(r1 \neq b1)$
$p2 := \text{FALSE}$
end

cir0
when
$mode = cir$
$r1 = b1$
$r2 = b2$
then
$mode := env$
$o1 := \text{FALSE}$
$o2 := \text{FALSE}$
$p1 := \text{FALSE}$
$p2 := \text{FALSE}$
end

3.3 Second Refinement

The State. The environment event are now accessing environment variables only ($r1$, $r2$, $b1$, and $b2$) together with the outputs of the circuit ($o1$ and $o2$). But, the circuit events still access the environment variables ($r1$, $r2$, $b1$, and $b2$). In this refinement, we introduce proper inputs $i1$ and $i2$ to the circuit.

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 from invariants **inv0_5** to **inv0_8**. For this, we introduce two new binary variables i_1 and i_2:

variables: $i1, i2$

inv2_1: $i1 \in \text{BOOL}$

inv2_2: $i2 \in \text{BOOL}$

The invariants relating $i1$ and $i2$ to $r1$, $r2$, $b1$, and $b2$ are straightforward:

inv2_2: $mode = cir \Rightarrow (i1 = \text{FALSE} \Leftrightarrow r1 = b1)$

inv2_3: $mode = cir \Rightarrow (i2 = \text{FALSE} \Leftrightarrow r2 = b2)$

The modification of the environment events are very simple:

env1
when
$mode = env$
$r1 = b1 + b\_2\_01(o1)$
then
$mode := cir$
$r1 := r1 + 1$
$b1 := b1 + b\_2\_01(o1)$
$b2 := b2 + b\_2\_01(o2)$
$i1 := \text{TRUE}$
$i2 := \text{bool}(r2 \neq b2 + b\_2\_01(o2))$
end

env2
when
$mode = env$
$r2 = b2 + b\_2\_01(o2)$
then
$mode := cir$
$r2 := r2 + 1$
$b1 := b1 + b\_2\_01(o1)$
$b2 := b2 + b\_2\_01(o2)$
$i1 := \text{bool}(r1 \neq b1 + b\_1\_01(o1))$
$i2 := \text{TRUE}$
end

env3
when
$mode = env$
$r1 = b1 + b\_2\_01(o1)$
$r2 = b2 + b\_2\_01(o2)$
then
$mode := cir$
$r1 := r1 + 1$
$r2 := r2 + 1$
$b1 := b1 + b\_2\_01(o1)$
$b2 := b2 + b\_2\_01(o2)$
$i1 := \text{TRUE}$
$i2 := \text{TRUE}$
end

env0
when
$mode = env$
then
$mode := cir$
$b1 := b1 + b\_2\_01(o1)$
$b2 := b2 + b\_2\_01(o2)$
$i1 := \text{bool}(r1 \neq b1 + b\_1\_01(o1))$
$i2 := \text{bool}(r1 \neq b2 + b\_1\_01(o2))$
end

Here are the new circuit events:

cir1
when
$mode = cir$
$i1 = \text{TRUE}$
$p2 = \text{FALSE}$
then
$mode := env$
$o1 := \text{TRUE}$
$o2 := \text{FALSE}$
$p1 := \text{FALSE}$
$p2 := i2$
end

cir2
when
$mode = cir$
$i2 = \text{TRUE}$
$p1 = \text{FALSE}$
then
$mode := env$
$o1 := \text{FALSE}$
$o2 := \text{TRUE}$
$p1 := i1$
$p2 := \text{FALSE}$
end

cir0
when
$mode = cir$
$i1 = \text{FALSE}$
$i2 = \text{FALSE}$
then
$mode := env$
$o1 := \text{FALSE}$
$o2 := \text{FALSE}$
$p1 := \text{FALSE}$
$p2 := \text{FALSE}$
end

3.4 Third Refinement: Reducing Non-determinacy of the Circuit

The State. The circuit we have obtained in the previous section is now complete and simple, but *still non-deterministic*: when $i1$ and $i2$ are both equal to TRUE with $p1$ and $p2$ both equal to FALSE, the circuit can choose to set $o1$ or $o2$ to TRUE. In other words, both events cir1 and cir2 are enabled. In order to make the circuit completely deterministic, we decide that, in this case, $o1$ say, will be the winner. In fact, we remove variables $p1$.

The Events. The environment events remain the same whereas the circuit events are modified as follows:

cir1
when
$mode = cir$
$i1 = \text{TRUE}$
$p2 = \text{FALSE}$
then
$mode := env$
$o1 := \text{TRUE}$
$o2 := \text{FALSE}$
$p2 := i2$
end

cir2
when
$mode = cir$
$i2 = \text{TRUE}$
$\neg(i1 = \text{TRUE} \ \wedge \ p2 = \text{FALSE})$
then
$mode := env$
$o1 := \text{FALSE}$
$o2 := \text{TRUE}$
$p2 := \text{FALSE}$
end

cir0
when
$mode = cir$
$i1 = \text{FALSE}$
$i2 = \text{FALSE}$
then
$mode := env$
$o1 := \text{FALSE}$
$o2 := \text{FALSE}$
$p2 := \text{FALSE}$
end

The circuit events are now clearly internally as well as externally deterministic.

Revisiting Deadlock Freedom. The interesting and fundamental last statement to prove is that the events of the circuit are deadlock free. 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:

thm3_1: $$mode = cir \Rightarrow \begin{pmatrix} i1 = \text{TRUE} \ \wedge \ p2 = \text{FALSE}) \ \vee \\ i2 = \text{TRUE} \ \wedge \ \neg (i1 = \text{TRUE} \ \wedge \ p2 = \text{FALSE}) \ \vee \\ i1 = \text{FALSE} \ \wedge \ i2 = \text{FALSE} \end{pmatrix}$$

3.5 Fourth Refinement: Building the Final Circuit

The circuit and environment now fulfill all the final conditions stated in section 1.7. As a result, we can construct our final circuit in a systematic fashion:

arbiter
when
$mode = cir$
then
$mode := env$
$o1 := \text{bool}\,(\,(i1 = \text{TRUE} \ \wedge \ p2 = \text{FALSE} \ \wedge \ \text{TRUE} = \text{TRUE}) \ \vee$
$(\ldots \ \wedge \ \text{FALSE} = \text{TRUE}) \ \vee$
$(\ldots \ \wedge \ \text{FALSE} = \text{TRUE})\,)$
$o2 := \text{bool}\,(\,(\ldots \ \wedge \ \text{FALSE} = \text{TRUE}) \ \vee$
$(i2 = \text{TRUE} \ \wedge \ \neg (i1 = \text{TRUE} \ \wedge \ p2 = \text{FALSE}) \ \wedge \ \text{TRUE} = \text{TRUE}) \ \vee$
$(\ldots \ \wedge \ \text{FALSE} = \text{TRUE})\,)$
$p2 := \text{bool}\,(\,(i1 = \text{TRUE} \ \wedge \ p2 = \text{FALSE} \ \wedge \ i2 = \text{TRUE}\,) \ \vee$
$(\ldots \ \wedge \ \text{FALSE} = \text{TRUE}) \ \vee$
$(\ldots \ \wedge \ \text{FALSE} = \text{TRUE})\,)$
end

This can be simplified as follows:

arbiter
when
$mode = cir$
then
$mode := env$
$o1 := \text{bool}\,(\,i1 = \text{TRUE} \ \wedge \ p2 = \text{FALSE}\,)$
$o2 := \text{bool}\,(\,i2 = \text{TRUE} \ \wedge \ \neg (i1 = \text{TRUE} \ \wedge \ p2 = \text{FALSE})\,)$
$p2 := \text{bool}\,(\,i1 = \text{TRUE} \ \wedge \ p2 = \text{FALSE} \ \wedge \ i2 = \text{TRUE}\,)$
end

This leads to the circuit of figure 15.

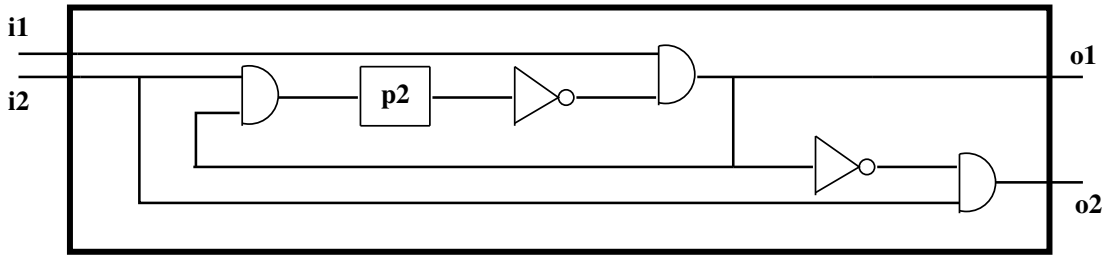

Fig. 15. The Arbiter

4 Third Example: A Special Road Traffic Light

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

4.1 Informal 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 is 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 is 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 behavior 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 Priority Circuit: Initial Model

The State. The simplest Priority circuit we can think of is one with two boolean inputs, car and clk: car corresponds to the information elaborated by the car sensors disposed on the small road and clk is an alarm coming from an external "timer" saying that the long delay is over. The priority circuit has two boolean outputs, chg and prt. chg yields the information concerning a *change* in the priority, whereas prt yields the priority in use. All this is indicated on figure 16.

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 in the circuit, (2) the presence of cars on the small road, and (3) the state of the alarm coming from the timer. The Priority circuit has an internal register, prt, holding the actual priority. The output chg is used externally to reset the external timer. The overall picture is indicated on figure 17.

The variables of the priority circuit are declared as follows:

Fig. 16. The priority Circuit

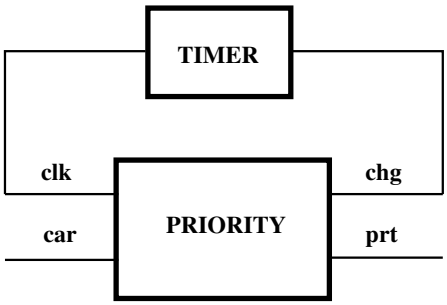

Fig. 17. Connection of the Priority Circuit with the Timer Circuit

variables: car, clk, chg, prt

inv0_1: $car \in \text{BOOL}$

inv0_2: $clk \in \text{BOOL}$

inv0_3: $chg \in \text{BOOL}$

inv0_4: $prt \in \text{BOOL}$

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

Events. The events of the Priority circuit elaborate priority changes. We have two such events, called main_to_small and small_to_main. 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 ($prt = \text{FALSE}$), when some car are present on the small road ($car = \text{TRUE}$) and when the long delay has passed ($clk = \text{TRUE}$): this corresponds to **Rule 1** and **Rule 2** above.

2. Event small_to_main_ can be fired when the priority is on small road ($prt = \text{TRUE}$), and when no cars are present on the small road ($car = \text{FALSE}$) or when the long delay has passed ($clk = \text{TRUE}$): this corresponds to **Rule 3** and **Rule 4**.

In both cases, the priority changes ($chg := \text{TRUE}$) and variable prt is modified accordingly. Here are the events:

main_to_small
when
$mode = cir$
$prt = \text{FALSE}$
$car = \text{TRUE}$
$clk = \text{TRUE}$
then
$mode := env$
$prt := \text{TRUE}$
$chg := \text{TRUE}$
end

small_to_main
when
$mode = cir$
$prt = \text{TRUE}$
$car = \text{FALSE} \lor clk = \text{TRUE}$
then
$mode := env$
$prt := \text{FALSE}$
$chg := \text{TRUE}$
end

Another series of events corresponds to the circuit doing nothing except resetting the chg output to FALSE (no change). This occurs in two circumstances:

1. Event do_nothing_1 can be fired when the priority is on main road ($prt = \text{FALSE}$) and when there are no cars on the small road ($car = \text{FALSE}$) or the delay has not passed yet ($clk = \text{FALSE}$): this corresponds to **Rule 1** and **Rule 2** above.

2. Event do_nothing_2 can be fired when the priority is on small road ($prt = \text{TRUE}$), when there are cars present on the small road ($car = \text{TRUE}$), and when the delay has not passed yet ($clk = \text{FALSE}$): this corresponds to **Rule 3** and **Rule 4**.

Here are these events:

do_nothing_1
when
$mode = cir$
$prt = \text{FALSE}$
$car = \text{FALSE} \lor clk = \text{FALSE}$
then
$mode := env$
$chg := \text{FALSE}$
end

do_nothing_2
when
$mode = cir$
$prt = \text{TRUE}$
$car = \text{TRUE}$
$clk = \text{FALSE}$
then
$mode := env$
$chg := \text{FALSE}$
end

The unique environment event is the following:

env1
when
$mode = env$
then
$mode := cir$
$car :\in \text{BOOL}$
$clk :\in \text{BOOL}$
end

Notice that this event is not very realistic as car may come and then disappear in a rather random way. In section ??, we shall make this event more realistic by splitting it.

Deadlock Freedom The Priority circuit is deadlock free as stated in this theorem:

$$\textbf{thm0\_1:} \quad mode = cir \;\Rightarrow\; \begin{pmatrix} prt = \text{FALSE} \;\wedge\; car = \text{TRUE} \;\wedge\; clk = \text{TRUE} \\ prt = \text{TRUE} \;\wedge\; (car = \text{FALSE} \;\vee\; clk = \text{TRUE}) \\ prt = \text{FALSE} \;\wedge\; (car = \text{FALSE} \;\vee\; clk = \text{FALSE}) \\ prt = \text{TRUE} \;\wedge\; car = \text{TRUE} \;\wedge\; clk = \text{FALSE} \end{pmatrix}$$

4.4 The final Priority Circuit

The priority circuit fulfills the condition of section 1.7. Notice that events do_nothing_1 and do_nothing_2 do not mention variable prt: in fact, we could consider that they both have the action $prt := prt$. With this in mind, the circuit generation goes as follows:

priority
when
$mode = cir$
then
$mode := env$

$$prt := \text{bool} \begin{pmatrix} (prt = \text{FALSE} \;\wedge\; car = \text{TRUE} \;\wedge\; clk = \text{TRUE} \;\wedge \text{TRUE} = \text{TRUE}) \;\vee \\ (prt = \text{FALSE} \;\wedge\; (car = \text{FALSE} \vee clk = \text{FALSE}) \;\wedge\; prt = \text{TRUE}) \;\vee \\ (prt = \text{TRUE} \;\wedge\; car = \text{TRUE} \;\wedge\; clk = \text{FALSE} \;\wedge\; prt = \text{TRUE}) \end{pmatrix}$$

$$chg := \text{bool} \begin{pmatrix} (prt = \text{FALSE} \;\wedge\; car = \text{TRUE} \;\wedge\; clk = \text{TRUE} \;\wedge\; \text{TRUE} = \text{TRUE}) \;\vee \\ (prt = \text{TRUE} \;\wedge\; (car = \text{FALSE} \;\vee\; clk = \text{TRUE}) \;\wedge\; \text{TRUE} = \text{TRUE}) \;\vee \\ (\ldots \;\wedge\; \text{FALSE} = \text{TRUE}) \;\vee \\ (\ldots \;\wedge\; \text{FALSE} = \text{TRUE})) \end{pmatrix}$$

end

This reduces trivially to the following:

priority
when
$mode = cir$
then
$mode := env$

$$prt := \text{bool} \begin{pmatrix} (prt = \text{FALSE} \;\wedge\; car = \text{TRUE} \;\wedge\; clk = \text{TRUE}) \;\vee \\ (prt = \text{TRUE} \;\wedge\; car = \text{TRUE} \;\wedge\; clk = \text{FALSE}) \end{pmatrix}$$

$$chg := \text{bool} \begin{pmatrix} (prt = \text{FALSE} \;\wedge\; car = \text{TRUE} \;\wedge\; clk = \text{TRUE}) \;\vee \\ (prt = \text{TRUE} \;\wedge\; (car = \text{FALSE} \;\vee\; clk = \text{TRUE})) \end{pmatrix}$$

end

This circuit can be further transformed in the following equivalent fashion:

priority
when
$mode = cir$
then
$mode := env$

$$prt := \text{bool}\left(\begin{array}{l}(prt = \text{TRUE} \ \wedge \ \neg \left(\begin{array}{l}(car = \text{TRUE} \ \wedge \ clk = \text{TRUE}) \ \vee \\ (car = \text{FALSE} \ \wedge \ prt = \text{TRUE})\end{array}\right) \ \vee \\ \\ (prt = \text{FALSE} \ \wedge \ \left(\begin{array}{l}(car = \text{TRUE} \ \wedge \ clk = \text{TRUE}) \ \vee \\ (car = \text{FALSE} \ \wedge \ prt = \text{TRUE})\end{array}\right)\end{array}\right)$$

$$chg := \text{bool}\left(\begin{array}{l}(car = \text{TRUE} \ \wedge \ clk = \text{TRUE}) \ \vee \\ (car = \text{FALSE} \ \wedge \ prt = \text{TRUE})\end{array}\right)$$

end

It is easy to figure out that these two events are equivalent. Hint: (1) do a proof by cases, ($prt = \text{TRUE}$, then $prt = \text{FALSE}$) to prove the equivalence concerning the assignment to prt, and (2) do a proof by cases ($car = \text{TRUE}$, then $car = \text{FALSE}$) to prove the equivalence concerning the assignment to chg. The last event is interesting because it contains three times the following fragment, which can thus be computed only once:

$$\begin{array}{l}car = \text{TRUE} \ \wedge \ clk = \text{TRUE} \ \vee \\ car = \text{FALSE} \ \wedge \ prt = \text{TRUE}\end{array}$$

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

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

This will be economically represented by an IF gate, considered to be an atomic one. Such a gate is pictorially represented on figure 18.

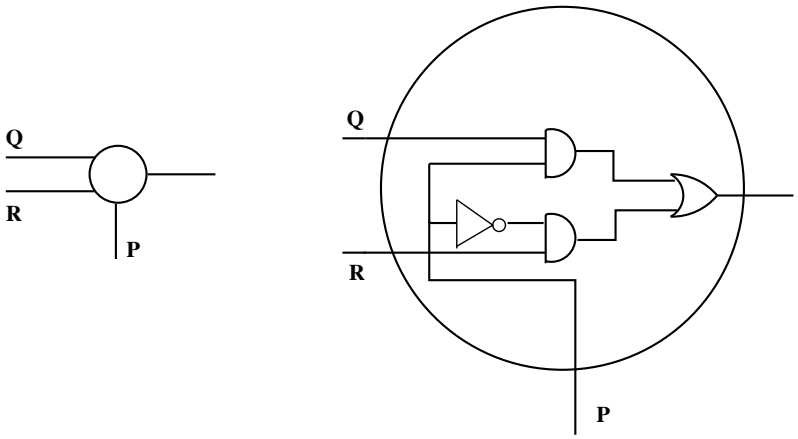

Fig. 18. An IF Gate

Equipped with such an IF gate, we can draw our Priority circuit as indicated in figure 19.

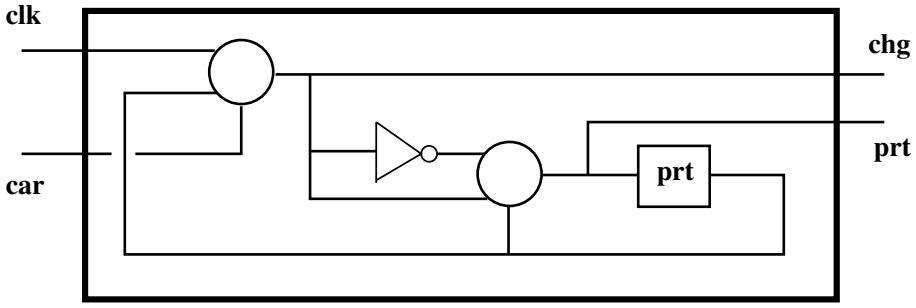

Fig. 19. The Priority Circuit

5 The Light Circuit

We now connect our Priority circuit to the Light circuit as is shown in the diagram below. The Light circuit delivers the various colors of both traffic lights. This is indicate on figure 20.

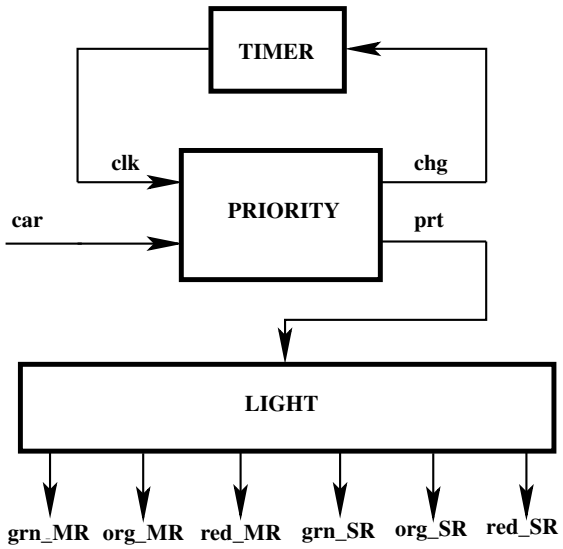

Fig. 20. The Priority Circuit Connected to the Light Circuit

5.1 An Abstraction: the Upper Circuit

We start with a simplified circuit whose rôle is to ensure the sequencing of a single traffic light, that of the *main road*.It is shown on figure 21.

We shall extend later that circuit to handle two synchronous traffic lights. The circuit has a single boolean entry prt which, when valued to TRUE, indicates that a the light appearance should give priority

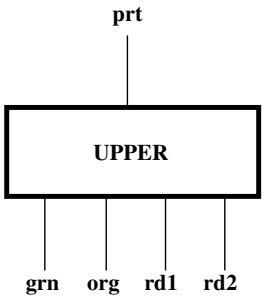

Fig. 21. The Upper Light Circuit

to the small road. It has four boolean outputs 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 TRUE. This can be formalized as follows:

variables: $prt, grn, org, rd1, rd2$

inv0_1: $prt \in \text{BOOL}$

inv0_2: $grn = \text{TRUE} \vee org = \text{TRUE} \vee rd1 = \text{TRUE} \vee rd2 = \text{TRUE}$

inv0_3: $grn = \text{TRUE} \Rightarrow org = \text{FALSE} \wedge rd1 = \text{FALSE} \wedge rd2 = \text{FALSE}$

inv0_4: $org = \text{TRUE} \Rightarrow rd1 = \text{FALSE} \wedge rd2 = \text{FALSE}$

inv0_5: $rd1 = \text{TRUE} \Rightarrow rd2 = \text{FALSE}$

inv0_6: $mode \in MODE$

The events of the circuit are straightforward

grn_to_org
when
$mode = cir$
$prt = \text{TRUE}$
$grn = \text{TRUE}$
then
$mode := env$
$grn := \text{FALSE}$
$org := \text{TRUE}$
end

org_to_rd1
when
$mode = cir$
$org = \text{TRUE}$
then
$mode := env$
$org := \text{FALSE}$
$rd1 := \text{TRUE}$
end

rd1_to_rd2
when
$mode = cir$
$prt = \text{FALSE}$
$rd1 = \text{TRUE}$
then
$mode := env$
$rd1 := \text{FALSE}$
$rd2 := \text{TRUE}$
end

rd2_to_grn
when
$mode = cir$
$rd2 = \text{TRUE}$
then
$mode := env$
$grn := \text{TRUE}$
$rd2 := \text{FALSE}$
end

We have two "do-nothing" events in the circuit and also an environment event assigning prt in a non-deterministic way. These are as follows:

grn_to_nth
when
$mode = cir$
$grn = \text{TRUE}$
$prt = \text{FALSE}$
then
$mode := env$
end

rd1_to_nth
when
$mode = cir$
$rd1 = \text{TRUE}$
$prt = \text{TRUE}$
then
$mode := env$
end

env_evt
when
$mode = env$
then
$mode := cir$
$prt :\in BOOL$
end

5.2 A Refinement: Adding the Lower Circuit

We refine the circuit by having now 6 outputs corresponding the the light appearance of both traffic lights. First those of the main road: grn_MR, org_MR, and red_MR. Then those of the small road: grn_SR, org_SR, and red_SR.

variables: $prt, grn, org, rd1, rd2$

grn_MR, org_MR, red_MR

grn_SR, org_SR, red_SR

The final colors are related to the variables of the initial model in a straightforward way:

inv1_1: $grn\_MR = grn$

inv1_2: $org\_MR = org$

inv1_3: $red\_MR = \text{TRUE} \Leftrightarrow (rd1 = \text{TRUE} \lor rd2 = \text{TRUE})$

inv1_4: $grn\_SR = rd1$

inv1_5: $org\_SR = rd2$

inv1_6: $red\_SR = \text{TRUE} \Leftrightarrow (grn = \text{TRUE} \lor org = \text{TRUE})$

We can prove the following safety theorems:

thm1_1: $red\_MR = \text{TRUE} \Leftrightarrow (grn\_SR = TRUE \vee org\_SR = \text{TRUE})$

thm1_2: $red\_SR = \text{TRUE} \Leftrightarrow (grn\_MR = TRUE \vee org\_MR = \text{TRUE})$

Next are the refinements of the events:

grn_to_org
when
$mode = cir$
$prt = \text{TRUE}$
$grn = \text{TRUE}$
then
$mode := env$
$grn := \text{FALSE}$
$org := \text{TRUE}$
$grn\_MR := \text{FALSE}$
$org\_MR := \text{TRUE}$
end

org_to_rd1
when
$mode = cir$
$org = \text{TRUE}$
then
$mode := env$
$org := \text{FALSE}$
$rd1 := \text{TRUE}$
$org\_MR := \text{FALSE}$
$red\_MR := \text{TRUE}$
$grn\_SR := \text{TRUE}$
$red\_SR := \text{FALSE}$
end

rd1_to_rd2
when
$mode = cir$
$prt = \text{FALSE}$
$rd1 = \text{TRUE}$
then
$mode := env$
$rd1 := \text{FALSE}$
$rd2 := \text{TRUE}$
$org\_SR := \text{TRUE}$
$grn\_SR := \text{FALSE}$
end

rd2_to_grn
when
$mode = cir$
$rd2 = \text{TRUE}$
then
$mode := env$
$grn := \text{TRUE}$
$rd2 := \text{FALSE}$
$grn\_MR := \text{TRUE}$
$red\_MR := \text{FALSE}$
$org\_SR := \text{FALSE}$
$red\_SR := \text{TRUE}$
end

The various circuit events can be unified as usual:

light
when
$mode = cir$
then
$mode := env$
$grn := \text{bool}(\, rd2 = \text{TRUE} \ \vee \ (prt = \text{FALSE} \ \wedge \ grn = \text{TRUE})\,)$
$org := \text{bool}(\, prt = \text{TRUE} \ \wedge \ grn = \text{TRUE}\,)$
$rd1 := \text{bool}(\, org = \text{TRUE} \ \vee \ (prt = \text{TRUE} \ \wedge \ rd1 = \text{TRUE})\,)$
$rd2 := \text{bool}(\, prt = \text{FALSE} \ \wedge \ rd1 = \text{TRUE}\,)$
$grn\_MR := \text{bool}(\, rd2 = \text{TRUE} \ \vee \ (prt = \text{FALSE} \ \wedge \ grn = \text{TRUE})\,)$
$org\_MR := \text{bool}(\, prt = \text{TRUE} \ \wedge \ grn = \text{TRUE}\,)$
$red\_MR := \text{bool}(\, org = \text{TRUE} \ \vee \ (prt = \text{TRUE} \ \wedge \ rd1 = \text{TRUE}) \vee$
$(prt = \text{FALSE} \ \wedge \ rd1 = \text{TRUE})\,)$
$grn\_SR := \text{bool}(\, org = \text{TRUE} \ \vee \ (prt = \text{TRUE} \ \wedge \ rd1 = \text{TRUE})\,)$
$org\_SR := \text{bool}(\, prt = \text{FALSE} \ \wedge \ rd1 = \text{TRUE}\,)$
$red\_SR := \text{bool}(\, rd2 = \text{TRUE} \ \vee \ (prt = \text{FALSE} \ \wedge \ grn = \text{TRUE}) \vee$
$(prt = \text{TRUE} \ \wedge \ grn = \text{TRUE})\,)$
end

The final Light circuit is shown on figure 22.

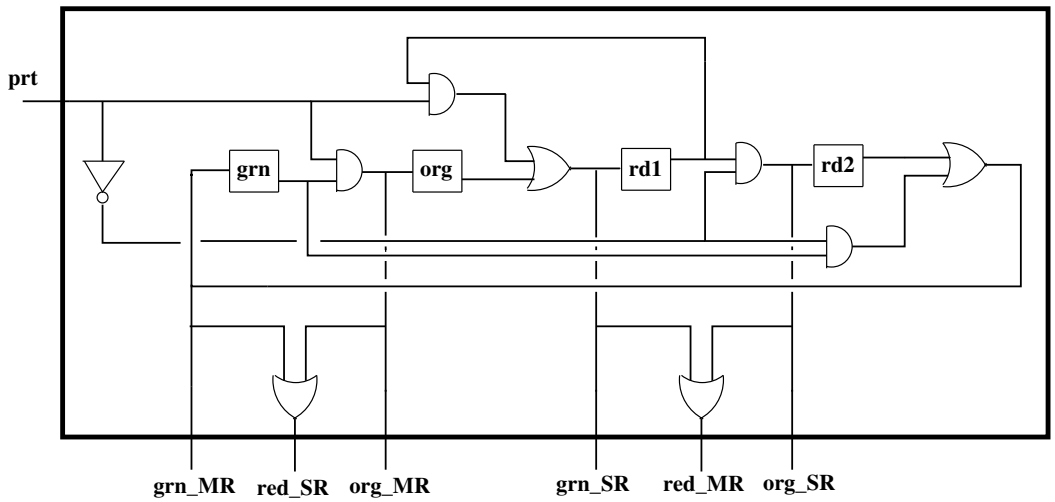

Fig. 22. The Light Circuit

References

1. T. Kropf. *Formal Hardware Verification: Methods and Systems in Comparison*. LNCS State-of-the-art Survey Springer. 1991