III. A Mechanical Press Controller (October 2008)

In this chapter, we develop the controller of another complete example: a mechanical press. The intention is to show how this can be done in a systematic fashion in order to obtain correct final code. In section 1, we present an informal description of this system. In section 2, we develop two general patterns that we shall subsequently use. The development of these patterns will be made by using the proofs as a mean of discovering the invariants and the guards of the events. In section 3, we defined the requirement document in a more precise fashion by using the terminology developed in the definition of the patterns. The main development of the mechanical press will take place in further sections where more design patterns will be presented.

1 Informal Description

1.1 Basic Equipments

A mechanical press is essentially made of the following pieces of equipment:

- a *vertical slide* which is either stopped or moving up and down very rapidly.
- an *electrical rotating motor* which can be stopped or working,
- a *connecting rod* which transmits the movement of the electrical motor to that of the slide,
- a *clutch* which allows to engage or disengage the motor on the connecting rod.

This is illustrated in Fig. 1.

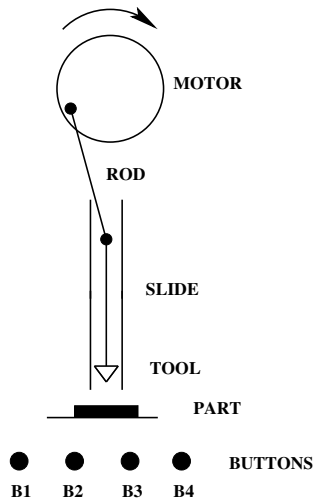


Fig. 1. Schematic View of the Press

1.2 Basic Commands and Buttons

The following *commands* can be performed by means of buttons named respectively B1, B2, B3, and B4.

- *Command* 1: start motor (this is performed by depressing button B1),
- *Command* 2: stop motor (this is performed by depressing button B2),
- *Command* 3: engage clutch (this is performed by depressing button B3),
- *Command* 4: disengage clutch (this is performed by depressing button B4).

1.3 Basic User Action

The following *actions* can be performed by the user (it is clearly better to do so when the vertical slide is stopped!).

- *Action* 1: change the tool at the lower extremity of the vertical slide,
- *Action* 2: put a part to be treated by the press at a specific place under the slide,
- *Action* 3: remove the part that has been treated by the press.

The very first schematic structure of the system could be thought of as being the one shown on Fig. 2.

Fig. 2. First Schematic View of the System

1.4 User Session

A typical *user session* is the following (we suppose that, *initially*, the motor is stopped and the clutch is disengaged):

- 1: start motor (*Command* 1),
- 2: change tool (*Action* 1),
- 3: put a part (*Action* 2),
- 4: engage the clutch (*Command* 3): the press now works,
- 5: disengage the clutch (*Command* 4): the press is stopped,
- 6: remove the part (*Action* 3),
- 7: repeat zero or more times items 3 to 6,
- 8: repeat zero or more times items 2 to 7,
- 9: stop motor (*Command* 2).

As can be seen, the philosophy of this mechanical press is that it can work without stopping the motor.

1.5 Danger: Necessity of a Controller

Clearly, *Action* 1 (change the tool), *Action* 2 (put a part), and *Action* 3 (remove a part) are dangerous because the user has to manipulate objects (tools, parts) in places which are just situated below the vertical slide. *Normally*, this slide should not move while doing such actions because the clutch must have been disengaged. However, the user could have forgotten to do so or a malfunction could have caused it not to happen.

As a consequence, a *controller* is placed between the commands and the equipment in order to make sure that things are working properly. In order to prevent malfunctions, the equipment is also reporting its own status to the controller. All this results in the second, more precise, system structure shown on Fig. 3.

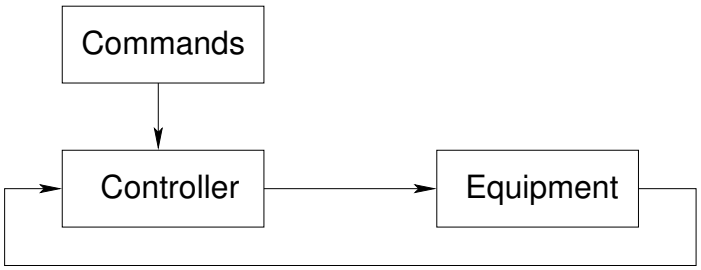


Fig. 3. Second Schematic View of the System

1.6 The Door

Placing a controller between the commands and the equipment is certainly not sufficient: one has also to make these commands more sophisticated *in order to protect the user*. In fact, the key is clearly the two commands for engaging and disengaging the clutch. For this, a *door* is put in front of the press.This is illustrated on Fig. 4.

Initially, the door is open. When the user depresses button B3 to engage the clutch, then the door is first closed *before* engaging the clutch, and when the user depresses button B4 to disengage the clutch, then the door is opened *after* disengaging the clutch.

2 Design Patterns

In this example, there are many cases where a user can depress a button, which is eventually followed by a certain reaction of the system. For example buttons B1 and B2 have an eventual action on the motor. This is not a direct action however. In other words, there is no direct connection between these buttons and the motor. Direct actions on the motor are initiated by the controller which sends commands after receiving some information coming from buttons B1 or B2.

For example, when the motor does not work the effect of depressing button B1 is to eventually have the motor working. Likewise, when the motor is working, the effect of depressing button B2 is that the motor will eventually stop. Note that when the user depresses such a button, say button B1, and releases it very quickly, it might be the case that nothing happen simply because the controller has not got enough time to figure out that this button was depressed.

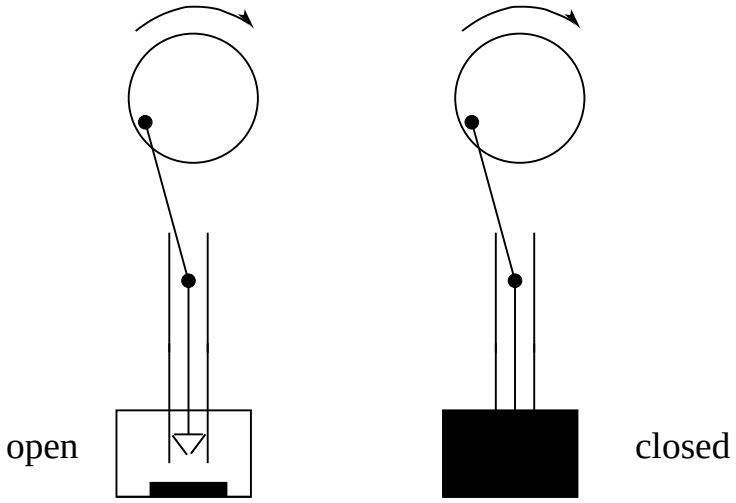


Fig. 4. The door

Another interesting case is the one where the user depresses button B1 and keep on depressing it by not removing his finger. Once the motor starts working, the user depresses button B2 with another finger. This results in having the motor being eventually stopped. But the fact that now button B1 is still depressed must not have any effect, the motor must not restart: this is due to the fact that any button must be first released in order to be taken into account once again.

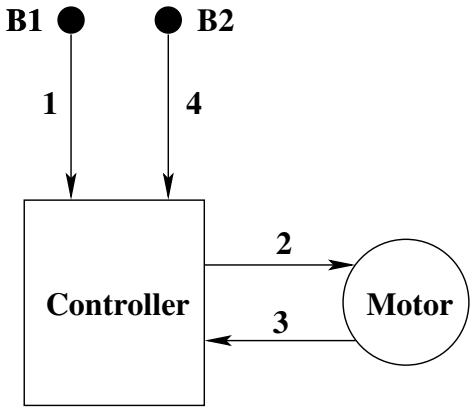


Fig. 5. Race Conditions Between 3 and 4

A more complicated case corresponds to the following sequence of actions as indicated in figure 5:

- (1) the user depresses button B1 (starting motor) and, not too quickly releases it,
- (2) the controller treats this depressing of button B1 by sending the start command to the motor,
- (3) the motor sends back to the controller an information telling that it has started working,
- (4) the user depresses button B2 (stopping motor) and, not too quickly, releases it.

The difficulty is that action (3) and (4) are done in parallel by the motor and by the user. Both these actions have to be taken into account by the controller. If action (3) (feedback from the motor) wins, then action (4) (depressing the stop button) is followed by a controller reaction whose purpose is to send to the motor the stop command. But if action (4) wins then the reaction of the controller cannot be performed as the controller does not know yet whether the motor is working since it has not received the corresponding information from the motor. In that case, the depressing on button B2 is not taken into account.

What we would like to do in this section is to have a *formal general study* of such cases. This will allow us to have a very systematic approach to the construction of our mechanical press reactive system in further sections.

2.1 Action and Reaction

The general paradigm in what we mentioned in the previous section is that of *actions* and *reactions*. Action and reactions can be illustrated by a diagram as shown in Fig. 6. We have an action, named a and represented by the plain line, followed by a reaction, named r and represented by the dashed line. Action and reaction can take two values: 0 or 1. We note that r, the reaction, always takes place *after* a, the action. In other words, r goes up (1) after a has gone up (1). Likewise, r goes down (0) after a has gone down (0).

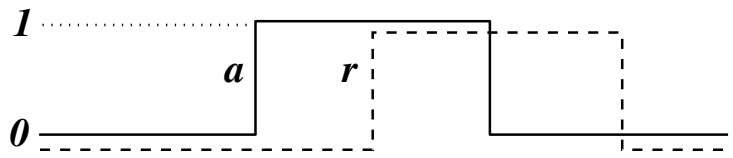


Fig. 6. Action and Reaction

2.2 First case: a Simple Action and Reaction Pattern Without Retro-action

Introduction. This first case corresponds to two possible scenarios. In the first one, it is possible that a goes up and down several times while r is not able to react so quickly: it stays down all the time. This is indicated in the diagram of Fig. 7.

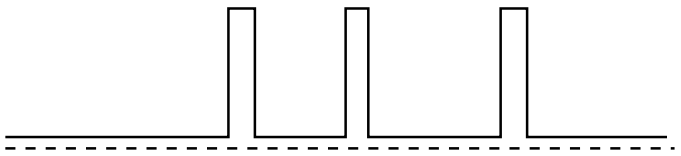


Fig. 7. Action and Weak Reaction (case 1)

As a second similar scenario, it is possible that once r has gone up then a goes down and then up again very quickly, so that r comes only down after a has done this several times. This is indicated in the diagram of Fig. 8.

When the behavior of an action-reaction system corresponds to what we have just described, it is said that we have a *weak synchronization* between the action and the reaction.

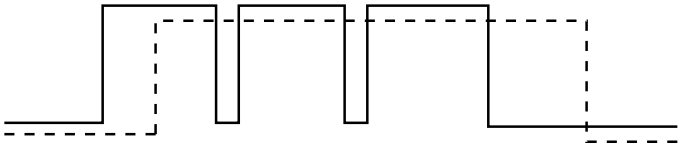


Fig. 8. Action and Weak Reaction (case 2)

Modeling. These two cases will be handled by the same model. Besides variables a and r denoting the state of the action and reaction (invariant **pat0_1** and **pat0_2** below), we introduce two counters: the first one is named ca and is associated with a and the second one is named cr and is associated with r (invariant **pat0_3** and **pat0_4** below). These counters denote the number of times each action and reaction respectively has gone up. The role of these counters is precisely to formalize the concept of a weak reaction: this is done in the main invariant, **pat0_5**, which says that cr is never greater than ca.

Note that these counters will not be present in our final definition of the patterns: they are there just *to make precise the constraint of the pattern*. For that reason, variables ca and cr will not be allowed in the guards of events, they will be present in event actions only.

variables: a, r, ca, cr

pat0_1: $a \in \{0, 1\}$

pat0_2: $r \in \{0, 1\}$

pat0_3: $ca \in \mathbb{N}$

pat0_4: $cr \in \mathbb{N}$

pat0_5: $cr \leq ca$

Initially, no action and reaction have taken place (event init below). Events a_on and a_off correspond to the action a. As can be seen, these events are not constrained by the reaction.

init
$a := 0$
$r := 0$
$ca := 0$
$cr := 0$

a_on
when
$a = 0$
then
$a := 1$
$ca := ca + 1$
end

a_off
when
$a = 1$
then
$a := 0$
end

This is not the case for r_on and r_off corresponding to the reaction r. These events are synchronized with some occurrences of events a_on and a_off. This is due to the presence of the guards $a = 1$ and $a = 0$ in the guards of events r_on and r_off.

r_on
when
$r = 0$
$a = 1$
then
$r := 1$
$cr := cr + 1$
end

r_off
when
$r = 1$
$a = 0$
then
$r := 0$
end

The weak synchronization of action and reaction is illustrated in the diagram of Fig 9. In this figure, the arrows simply express that the occurrence of an event relies on the previous occurrences of some others. For example, the occurrence of event r_on depends on that of event a_on and on that of event r_off. Note that these arrows have to be understood informally only.

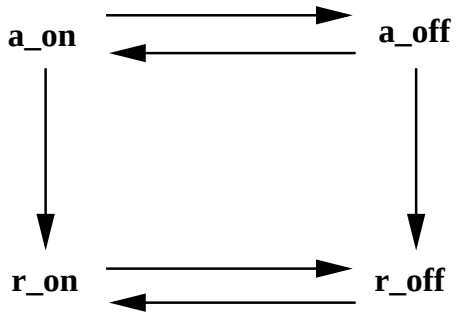


Fig. 9. Weak Synschronisation of the Events

Proofs. The proofs of invariant preservation are straightforward. Unfortunately, one of them fails. This is the proof of the preservation of invariant **pat0_5** by event r_on, that is r_on/**pat0_5**/INV.

r_on
when
$r = 0$
$a = 1$
then
$r := 1$
$cr := cr + 1$
end

One has to prove the following (after some simplifications):

Invariant **pat0_5**	$cr \leq ca$
Guards of	$r = 0$
event r_on	$a = 1$
$\vdash$	$\vdash$
Modified invariant **pat0_5**	$cr + 1 \leq ca$

We could solve difficulty by adding the predicate $cr < ca$ in the guard of event r_on: this is certainly the most economical solution as it does not affect the rest of the model. But, as was pointed out earlier, we do not want to incorporate counter variables in event guards. This suggests the following implicative invariant:

$$a = 1 \ \Rightarrow \ cr < ca$$

which is clearly preserved by event a_on which simultaneously sets a to 1 and increments ca, also trivially by events a_off (setting a to 0 and keeping cr and ca untouched) and r_off (keeping a, cr, and ca untouched). But unfortunately, this invariant is not preserved, again by event r_on. In this case, we have to prove the following:

New proposed invariant	$a = 1 \ \Rightarrow \ cr < ca$
Guards of	$r = 0$
event r_on	$a = 1$
$\vdash$	$\vdash$
Modified proposed invariant	$a = 1 \ \Rightarrow \ cr + 1 < ca$

This can be simplified to the following:

$$\begin{array}{l} cr < ca \\ r = 0 \\ a = 1 \\ \vdash \\ cr + 1 < ca \end{array}$$

This shows that our first proposed invariant, $a = 1 \ \Rightarrow \ cr < ca$ was not strong enough. The reader could also convince himself that the invariant $r = 0 \ \Rightarrow \ cr < ca$, would not be sufficient either. Thus, we have to also suppose that $r = 0$ holds. This leads to the following new invariant:

pat0_6:	$a = 1 \ \wedge \ r = 0 \ \Rightarrow \ cr < ca$

The preservation of this invariant by event r_on leads to the following

Invariant **pat0_6**	$a = 1 \ \wedge \ r = 0 \ \Rightarrow \ cr < ca$
Guards of	$r = 0$
event r_on	$a = 1$
$\vdash$	$\vdash$
Modified invariant **pat0_6**	$a = 1 \ \wedge \ 1 = 0 \ \Rightarrow \ cr + 1 < ca$

This simplifies to the following which holds trivially since there is a false assumption, namely $1 = 0$:

$$\begin{array}{l} cr < ca \\ r = 0 \\ a = 1 \\ 1 = 0 \\ \vdash \\ cr + 1 < ca \end{array}$$

2.3 Second Case: a Simple Action Pattern with a Retro-acting Reaction

Introduction. In this section, we refine the previous model by imposing now that the diagrams shown in figures Fig. 8 and Fig. 7 are not possible. We now have a *strong synchronization* between the action and the reaction. The only well synchronized possibilities are those indicated in Fig. 10.

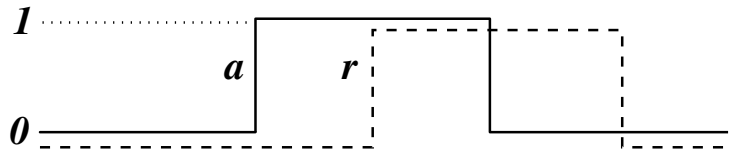


Fig. 10. Action and Strong Reaction

Modeling. We have exactly the same variables as in previous case, with an additional invariant stipulating that ca cannot exceed cr by more than one. In other words, either ca and cr are equal or ca is equal to $cr + 1$. This yields the following:

pat1_1: $ca \leq cr + 1$

Proofs. To begin with, since we do not know how to modify the events, we do not modify them at all. The idea again is that the failure of some proofs will give us some clues on how to improve the situation. In fact, all proofs succeed except one. Event a_on cannot maintain the new invariant **pat1_1**.

a_on
when
$a = 0$
then
$a := 1$
$ca := ca + 1$
end

After some simplifications, we have to prove:

Invariant **pat0_5**	$cr \leq ca$
Invariant **pat1_1**	$ca \leq cr + 1$
Guard of a_on	$a = 0$
$\vdash$	$\vdash$
Modified invariant **pat1_1**	$ca + 1 \leq cr + 1$

That is:

$$
\begin{array}{l}
cr \leq ca \\
ca \leq cr + 1 \\
a = 0 \\
\vdash \\
ca \leq cr
\end{array}
$$

The impossibility to prove this statement suggests the following invariant since ca cannot be strictly smaller than cr because of invariant **pat0_5** ($cr \leq ca$):

pat1_2:	$a = 0 \;\Rightarrow\; ca = cr$

Unfortunately, this time event a_off cannot preserve this invariant.

a_off
when
$a = 1$
then
$a := 0$
end

After some simplification, we are left to prove the following:

Guards of a_off	$a = 1$
$\vdash$	$\vdash$
Modified invariant **pat1_2**	$0 = 0 \;\Rightarrow\; ca = cr$

Note that we already have the following (this is **pat0_6**):

$$a = 1 \;\wedge\; r = 0 \;\Rightarrow\; cr < ca$$

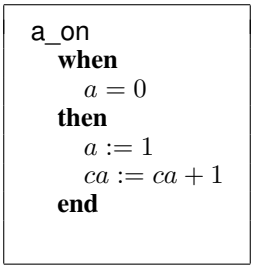
This suggests trying the following invariant

pat1_3:	$a = 1 \;\wedge\; r = 1 \;\Rightarrow\; ca = cr$

But unfortunately we have no guarantee that r is equal to 1 when we are using event a_off, *unless, of course, we add* $r = 1$ *as a new guard for event* a_off . We thus try to refine a_off by strengthening its guard as follows:

a_off
when
$a = 1$
$r = 1$
then
$a := 0$
end

Unfortunately, this time we have a problem with a_on.



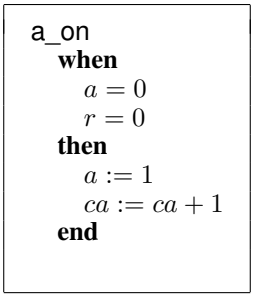
The preservation of the proposed invariant **pat1_3** leads to the following to prove:

Invariant **pat1_2**	$a = 0 \Rightarrow ca = cr$
Guards of a_on	$a = 0$
$\vdash$	$\vdash$
Modified invariant **pat1_3**	$1 = 1 \ \wedge \ r = 1 \ \Rightarrow \ ca + 1 = cr$

This can be simplified to the following:

$$\begin{array}{l} ca = cr \\ a = 0 \\ r = 1 \\ \vdash \\ ca + 1 = cr \end{array}$$

The only possibility to prove this is to have an additional guard in a_on in order to obtain a contradiction. The one that comes naturally is thus $r = 0$ (it will contradict $r = 1$). We thus refine a_on by strengthening its guard as follows:



And now we discover that *all invariant preservation proofs succeed*. Notice that we can put the two invariants **pat1_2** and **pat1_3** together:

pat1_2: $a = 0 \ \Rightarrow \ ca = cr$

pat1_3: $a = 1 \ \wedge \ r = 1 \ \Rightarrow \ ca = cr$

This leads to the following invariant which can thus replace the two previous ones:

pat1_4: $a = 0 \ \vee \ r = 1 \ \Rightarrow \ ca = cr$

It is very instructive to put invariant **pat0_6** next to this one:

pat0_6: $a = 1 \ \wedge \ r = 0 \ \Rightarrow \ cr < ca$

As can be seen, the antecedent of **pat0_6** is the negation of that of **pat1_4**. And now we can see in the diagram of Fig. 11 the places where these invariants hold.

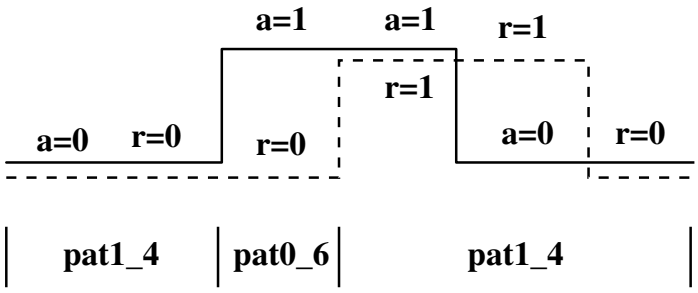


Fig. 11. Showing where the Invariants Hold

To summarize, here are the events for this strong synchronization case. We have removed the counters which were present just to formalize the relationship between the events:

a_on
when
$a = 0$
$r = 0$
then
$a := 1$
end

a_off
when
$a = 1$
$r = 1$
then
$a := 0$
end

r_on
when
$r = 0$
$a = 1$
then
$r := 1$
end

r_off
when
$r = 1$
$a = 0$
then
$r := 0$
end

The strong synchronization is illustrated on the diagram of Fig. 12.

3 Requirements of the Mechanical Press

In view of what we have seen in previous section, we now can clearly present the requirements of our Mechanical Press. We first have three requirements defining what the equipment are:

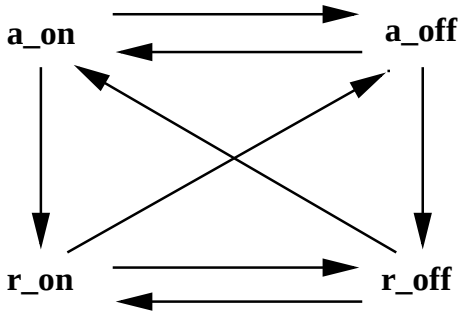


Fig. 12. Strong Synchronization

The system has got the following pieces of equipment: a Motor, a Clutch, and a Door	EQP_1

Four Buttons are used to start and stop the motor, and engage and disengage the clutch	EQP_2

A Controller is supposed to manage these equipment	EQP_3

Then we present the ways these equipment are connected to the controller:

Buttons and Controller are weakly synchronized	FUN_1

Controller and Equipment are strongly synchronized	FUN_2

Next are the two main safety requirements of the system:

When the clutch is engaged, the motor must work	SAF_1

When the clutch is engaged, the door must be closed	SAF_2

Finally, more constraints are put in place between the clutch and the door:

When the clutch is disengaged, the door cannot be closed several times, ONLY ONCE	FUN_3

When the door is closed, the clutch cannot be disengaged several times, ONLY ONCE	FUN_4

Opening and closing the door is not independent. It must be synchronized with disengaging and engaging the clutch	FUN_5

The overall structure of the system is presented in the diagram of Fig. 13

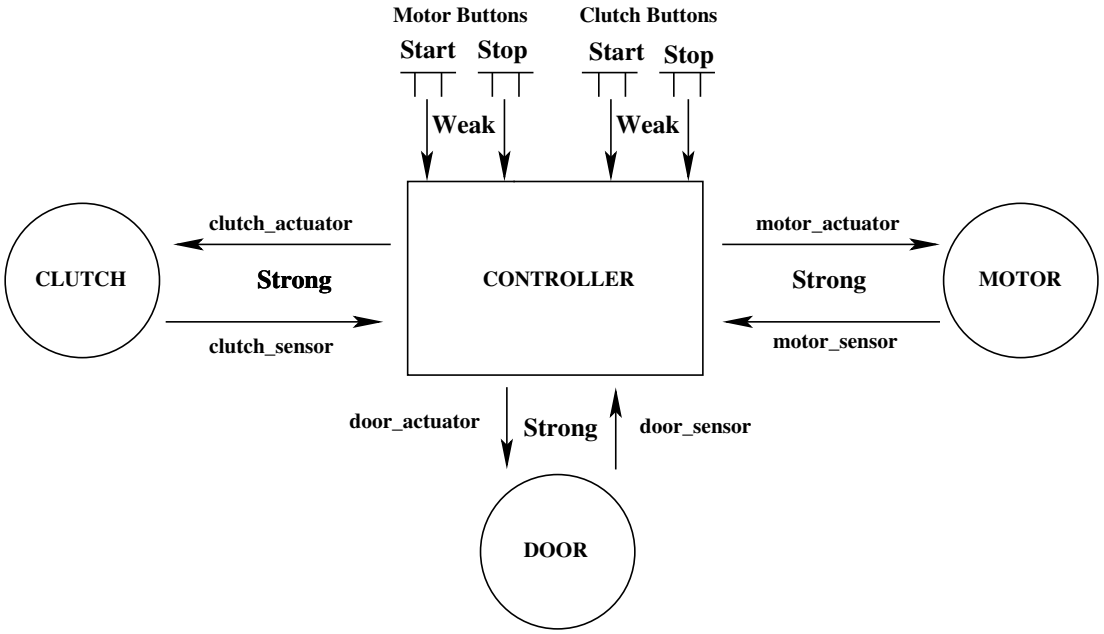


Fig. 13. The Press Controller

4 Refinement Strategy

In the following sections we are going to develop the design of the mechanical press according to the following strategy:

– Initial model: Connecting the controller to the motor,
– 1st refinement: Connecting the motor button to the controller,
– 2nd refinement: Connecting the controller to the clutch,
– 3rd refinement: Constraining the clutch and the motor,
– 4th refinement: Connecting the controller to the door,
– 5th refinement: Constraining the clutch and the door,
– 6th refinement: More constraints between the clutch and the door,
– 7th refinement: Connecting the clutch button to the controller.

In each case, we are going to do so by instantiating some design patterns.

5 Initial Model: Connecting the Controller to the Motor

5.1 Introduction.

This initial model formalizes the connection of the controller to the motor as illustrated in Fig. 14

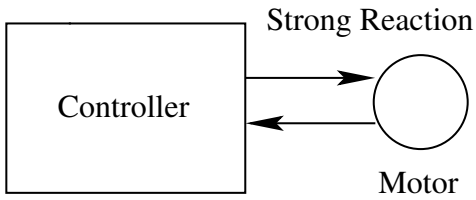


Fig. 14. Connecting the Controller to the Motor

We take partially into account requirement FUN_2:

Controller are Equipment are strongly synchronized	FUN_2

5.2 Modeling.

We first define a context with the set $STATUS$ defining the two different status of the motor: $stopped$ or $working$:

set: $STATUS$	**constants:** $stopped$ $working$	**axm0_1:** $STATUS = \{stopped, working\}$ **axm0_2:** $stopped \neq working$

Then we define two variables corresponding to the connection of the motor to the controller: $motor\_actuator$ and $motor\_sensor$. Variable $motor\_actuator$ formalizes the connection of the controller to the motor. It corresponds to the command sent by the controller, either to start or to stop the motor. Variable $motor\_sensor$ formalizes the connection of the motor to the controller. It corresponds to the feedback sent by the motor concerning its *physical status*.

variables:	$motor\_actuator$ $motor\_sensor$

inv0_1:	$motor\_sensor \in STATUS$
inv0_2:	$motor\_actuator \in STATUS$

In this connection, the controller acts as an *action* whereas the motor acts as a *reaction*. As we know, the reaction of the motor is strongly synchronized to the action of the controller. The idea then is to use the corresponding pattern (section 2.3) by *instantiating* it to the problem at hand. More precisely, we are going to instantiate the strong pattern as follows:

a	$\rightsquigarrow$	$motor\_actuator$
r	$\rightsquigarrow$	$motor\_sensor$
0	$\rightsquigarrow$	$stopped$
1	$\rightsquigarrow$	$working$
a_on	$\rightsquigarrow$	treat_start_motor
a_off	$\rightsquigarrow$	treat_stop_motor
r_on	$\rightsquigarrow$	Motor_start
r_off	$\rightsquigarrow$	Motor_stop

This leads first to the following events, which are supposed to represent the action of the controller:

a_on
when
$a = 0$
$r = 0$
then
$a := 1$
end

treat_start_motor
when
$motor\_actuator = stopped$
$motor\_sensor = stopped$
then
$motor\_actuator := working$
end

a_off
when
$a = 1$
$r = 1$
then
$a := 0$
end

treat_stop_motor
when
$motor\_actuator = working$
$motor\_sensor = working$
then
$motor\_actuator := stopped$
end

In this section and in the rest of this chapter, we shall follow the convention that the names of the events pertaining the controller all start with the prefix "treat-". A contrario, events whose names do not start with the prefix "treat-" are physical events occurring in the environment.

The following events are supposed to represent the physical reaction of the motor:

r_on
when
$r = 0$
$a = 1$
then
$r := 1$
end

Motor_start
when
$motor\_sensor = stopped$
$motor\_actuator = working$
then
$motor\_sensor := working$
end

r_off
when
$r = 1$
$a = 0$
then
$r := 0$
end

Motor_stop
when
$motor\_sensor = working$
$motor\_actuator = stopped$
then
$motor\_sensor := stopped$
end

5.3 Summary of the Events

- Environment
  - motor_start
  - motor_stop

- Controller
  - treat_start_motor
  - treat_stop_motor

6 First Refinement: Connecting the Motor Buttons to the Controller

6.1 Introduction.

We extend now the connection introduced in the previous section by connecting the motor buttons B1 (start motor) and B2 (stop motor) to the controller. This corresponds to the diagram of Fig. 15

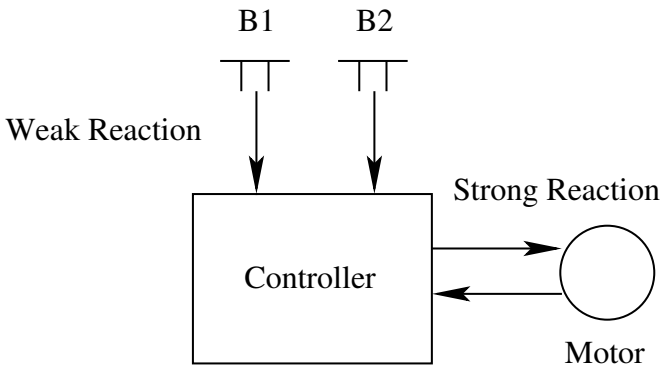


Fig. 15. Connecting the Motor Buttons to the Controller

We take partially into account requirement FUN_1:

Buttons and Controller are weakly synchronized	FUN_1

6.2 Modeling

We define two boolean variables corresponding to the connection of the motor buttons B1 and B2 to the controller: $start\_motor\_button$ and $stop\_motor\_button$. These physical variables denote the status of buttons B1 and B2 respectively: when equal to TRUE, it means that the corresponding button is physically depressed, when equal to FALSE, it means that it is physically released.

We define two more boolean variables, this time controller variables : $start\_motor\_impulse$ and $stop\_motor\_impulse$. These variables denotes the *knowledge* by the controller of the physical status of the buttons. They are clearly distinct from the two previous variables as the change of the physical status of a button occurs *before* the controller can be aware of it.

variables: ...
$start\_motor\_button$
$stop\_motor\_button$
$start\_motor\_impulse$
$stop\_motor\_impulse$

inv1_1: $stop\_motor\_button \in \text{BOOL}$
inv1_2: $start\_motor\_button \in \text{BOOL}$
inv1_3: $stop\_motor\_impulse \in \text{BOOL}$
inv1_4: $start\_motor\_impulse \in \text{BOOL}$

As we know, the controller weakly reacts to the buttons: it means that the buttons can be sometimes quickly depressed and released without the controller reacting to it: the behavior is clearly an instantiation of the weak reaction pattern we studied in section 2.2. Thus, we are going to instantiate the weak pattern as follows:

a_on	$\leadsto$	push_start_motor_button
a_off	$\leadsto$	release_start_motor_button
r_on	$\leadsto$	treat_start_motor
r_off	$\leadsto$	treat_release_start_motor_button
a	$\leadsto$	$start\_motor\_button$
r	$\leadsto$	$start\_motor\_impulse$
0	$\leadsto$	FALSE
1	$\leadsto$	TRUE

Here are the first two events:

a_on
when
$a = 0$
then
$a := 1$
end

push_start_motor_button
when
$start\_motor\_button = \text{FALSE}$
then
$start\_motor\_button := \text{TRUE}$
end

a_off
when
$a = 1$
then
$a := 0$
end

release_start_motor_button
when
$start\_motor\_button = \text{TRUE}$
then
$start\_motor\_button := \text{FALSE}$
end

Here are the two other events. As can be seen, the event treat_start_motor, which used to be the instantiation of an *action* in the initial model, is now the instantiation of a *reaction*. It is renamed below treat_push_start_motor_button:

r_on

when
$r = 0$
$a = 1$

then
$r := 1$

end

treat_push_start_motor_button
refines
treat_start_motor
when
$start\_motor\_impulse = \text{FALSE}$
$start\_motor\_button = \text{TRUE}$
$motor\_actuator = stopped$
$motor\_sensor = stopped$
then
$start\_motor\_impulse := \text{TRUE}$
$motor\_actuator := working$
end

r_off
when
$r = 1$
$a = 0$
then
$r := 0$
end

treat_release_start_motor_button
when
$start\_motor\_impulse = \text{TRUE}$
$start\_motor\_button = \text{FALSE}$
then
$start\_motor\_impulse := \text{FALSE}$
end

In order to understand what is happening here, let us show again the abstract event treat_start_motor.

treat_start_motor
when
$motor\_actuator = stopped$
$motor\_sensor = stopped$
then
$motor\_actuator := working$
end

We can see how the new pattern is *superposed* to the previous one:

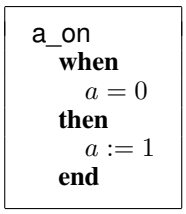
treat_push_start_motor_button
refines
treat_start_motor
when
$start\_motor\_impulse = \text{FALSE}$
$start\_motor\_button = \text{TRUE}$
$motor\_actuator = stopped$
$motor\_sensor = stopped$
then
$start\_motor\_impulse := \text{TRUE}$
$motor\_actuator := working$
end

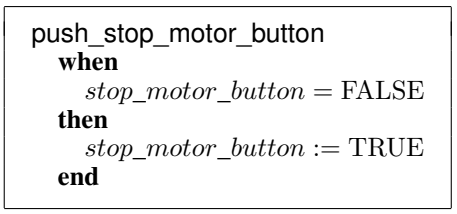
The guard of the concrete version of event treat_push_start_motor_button is made stronger and the action is enlarged: the new version of this event is indeed a refinement of the previous one. But, at the same time, the new version of this event is also *a refinement of the pattern* (up to renaming).

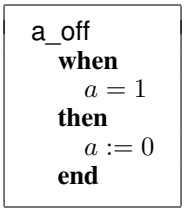
We now instantiate the weak pattern as follows:

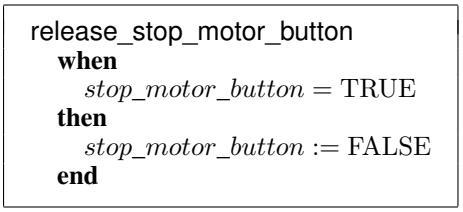
a_on	$\leadsto$	push_stop_motor_button
a_off	$\leadsto$	release_stop_motor_button
r_on	$\leadsto$	treat_stop_motor
r_off	$\leadsto$	treat_release_stop_motor_button
a	$\leadsto$	$stop\_motor\_button$
r	$\leadsto$	$stop\_motor\_impulse$
0	$\leadsto$	FALSE
1	$\leadsto$	TRUE

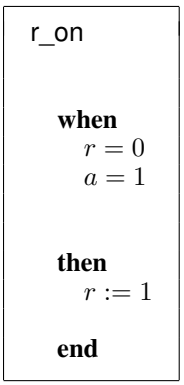
Once again, we can see that the event treat_stop_motor which used to be the instantiation of an action in the initial model is now the instantiation of a reaction. It is renamed treat_push_stop_motor_button.

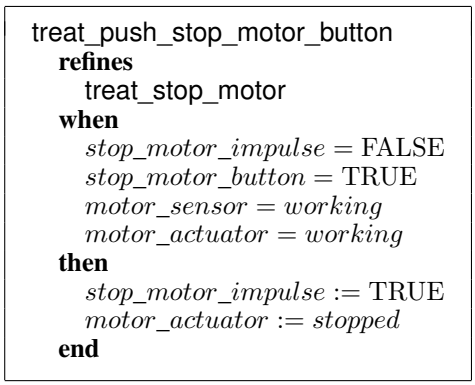












r_off
when
$r = 1$
$a = 0$
then
$r := 0$
end

treat_release_stop_motor_button
when
$stop\_motor\_impulse = \text{TRUE}$
$stop\_motor\_button = \text{FALSE}$
then
$stop\_motor\_impulse := \text{FALSE}$
end

In the diagram of Fig. 16, you can see a combined synchronization of the various events.

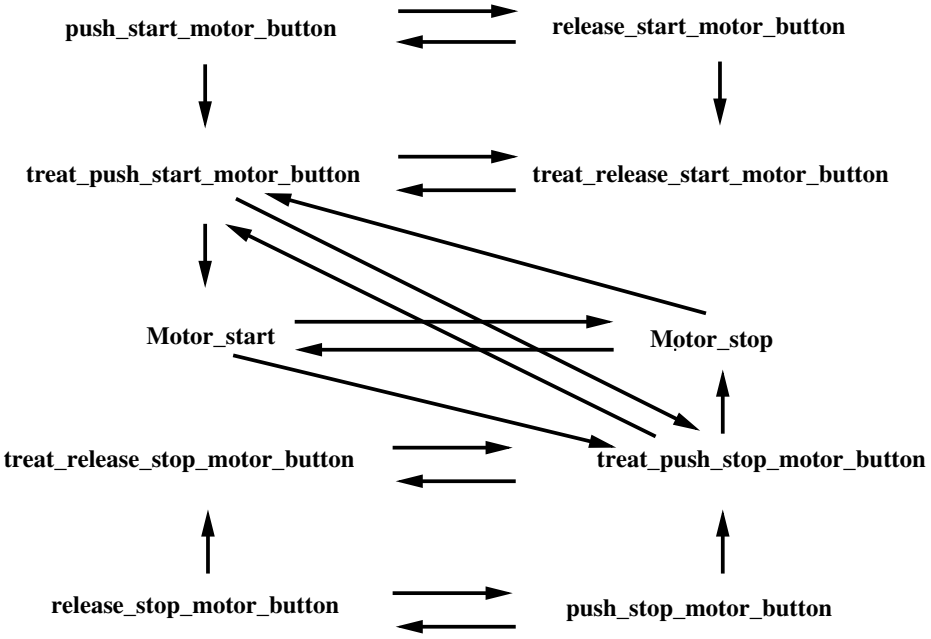


Fig. 16. Combined Synchronizations

6.3 Adding "false" events

The problem we tackle in this section has to do with the superposition of a pattern on an existing event. A typical example is the following event:

treat_push_start_motor_button
refines
treat_start_motor
when
$start\_motor\_impulse = \text{FALSE}$
$start\_motor\_button = \text{TRUE}$
$motor\_actuator = stopped$
$motor\_sensor = stopped$
then
$start\_motor\_impulse := \text{TRUE}$
$motor\_actuator := working$
end

In case the following condition is false

$$motor\_actuator = stopped \ \wedge \ motor\_sensor = stopped$$

while the following condition is true:

$$start\_motor\_impulse = \text{FALSE} \ \wedge \ start\_motor\_button = \text{TRUE}$$

then the event cannot be "executed" but nevertheless the button has been depressed so that the assignment

$$start\_motor\_impulse := \text{TRUE}$$

must be "executed". As a consequence, it is necessary to define the following additional event:

treat_push_start_motor_button_false
when
$start\_motor\_impulse = \text{FALSE}$
$start\_motor\_button = \text{TRUE}$
$\neg \ (motor\_actuator = stopped \ \wedge$
$motor\_sensor = stopped)$
then
$start\_motor\_impulse := \text{TRUE}$
end

In the sequel, we shall encounter similar cases for all buttons.

6.4 Summary of the Events

- Environment
  - motor_start
  - motor_stop
  - push_start_motor_button
  - release_start_motor_button
  - push_stop_motor_button

- release_stop_motor_button

- Controller
  - treat_push_start_motor_button
  - treat_push_start_motor_button_false
  - treat_push_stop_motor_button
  - treat_push_stop_motor_button_false
  - treat_release_start_motor_button
  - treat_release_stop_motor_button

7 Second Refinement: Connecting the Controller to the Clutch

We now connect the controller to the clutch. As it follows exactly the same approach as the one we have already used for the connection of the controller to the motor in section 6, we simply copy (after renaming "motor" to "clutch") what has been done in the initial model.

7.1 Summary of the Events

- Environment
  - motor_start
  - motor_stop
  - clutch_start
  - clutch_stop
  - push_start_motor_button
  - release_start_motor_button
  - push_stop_motor_button
  - release_stop_motor_button

- Controller
  - treat_push_start_motor_button
  - treat_push_start_motor_button_false
  - treat_push_stop_motor_button
  - treat_push_stop_motor_button_false
  - treat_release_start_motor_button
  - treat_release_stop_motor_button
  - treat_start_clutch
  - treat_stop_clutch

8 Another Design Pattern: Weak Synchronization of Two Strong Reactions

Our next step in designing the Mechanical Press is to take account of the following additional safety constraint:

When the clutch is engaged, the motor must work	SAF_1

It means that engaging the clutch is not independent of the starting of the motor as was the case in the previous refinement, where we had two completely independent strongly synchronized connections: that of the motor and that of the clutch. For studying this in general, we now consider another design pattern.

8.1 Introduction.

In this design pattern, we have two strongly synchronized patterns as indicated in Fig. 17, where in each case the arrows indicate the strong synchronization at work. Note that the first action and reaction are called a and r as before, whereas the second ones are called b and s.

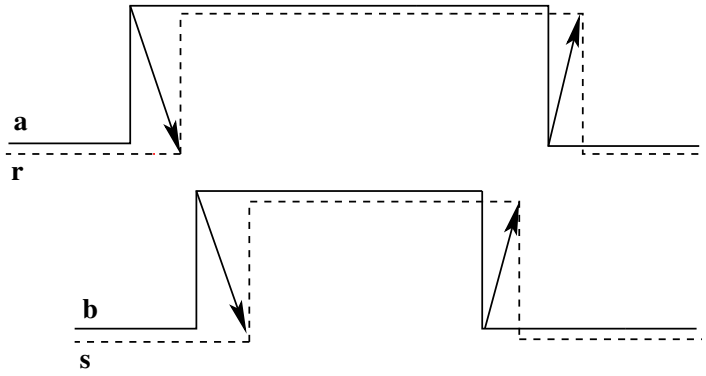


Fig. 17. Two Strongly Synchronized Action-reactions

We would like now to synchronize these actions and reactions so that the second reaction, s, only occurs when the first one, r, is enabled. In other words, we would like to ensure the following: $s = 1 \Rightarrow r = 1$.

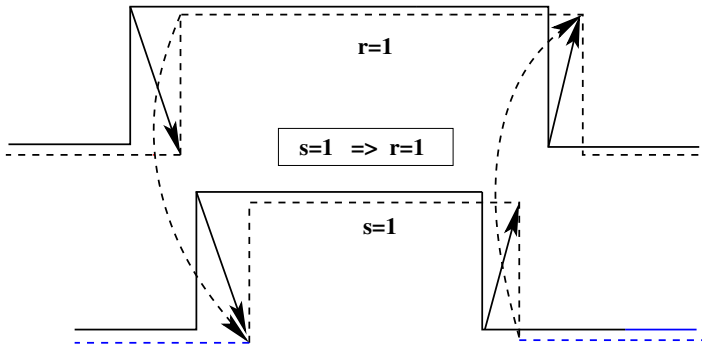


Fig. 18. Synchronizing two Strongly Synchronized Action-reactions

This is illustrated in Fig. 18, where the dashed arrows indicate this new synchronization. But this synchronization between the two is supposed to be weak only. For example, in our case, it is possible that the motor is started and stopped several time before the clutch is indeed engaged. Likewise, it is possible that the clutch is disengaged and re-engaged several times before the motor is stopped. All this is illustrated in Fig. 19.

In the diagrams of Fig. 19, the new relationship between the various events is illustrated by the dashed arrows. The reason why these arrows are dashed is that we have an additional constraint stating that *we do not want to modify the reacting events* s_on *and* r_off. This is illustrated in Fig. 20. More precisely,

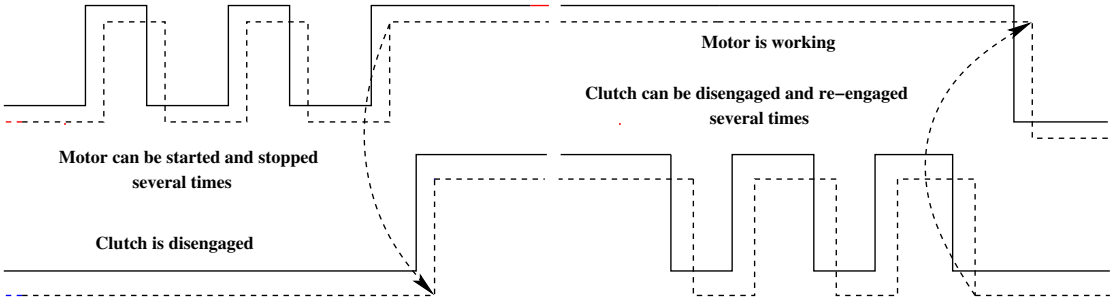


Fig. 19. Weak Synchronization of the Motor and the Clutch

we want to act at the level of the actions which have enabled these events. This is what we shall formalise in the next section.

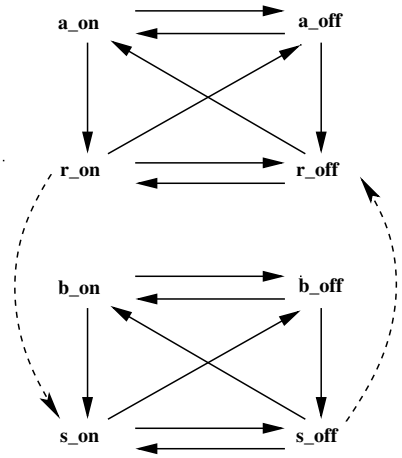


Fig. 20. Weak Synchronization of two Strongly Synchronized Action-reactions

8.2 Modeling.

Next is a blind copy of the two strongly synchronized patterns:

dbl0_1:	$a \in \{0,1\}$
dbl0_2:	$r \in \{0,1\}$
dbl0_3:	$ca \in \mathbb{N}$
dbl0_4:	$cr \in \mathbb{N}$
dbl0_5:	$a = 1 \wedge r = 0 \Rightarrow ca = cr + 1$
dbl0_6:	$a = 0 \vee r = 1 \Rightarrow ca = cr$

dbl0_7:	$b \in \{0,1\}$
dbl0_8:	$s \in \{0,1\}$
dbl0_9:	$cb \in \mathbb{N}$
dbl0_10:	$cs \in \mathbb{N}$
dbl0_11:	$b = 1 \wedge s = 0 \Rightarrow cb = cs + 1$
dbl0_12:	$b = 0 \vee s = 1 \Rightarrow cb = cs$

a_on
when
$a = 0$
$r = 0$
then
$a, ca := 1, ca + 1$
end

a_off
when
$a = 1$
$r = 1$
then
$a := 0$
end

r_on
when
$r = 0$
$a = 1$
then
$r, cr := 1, cr + 1$
end

r_off
when
$r = 1$
$a = 0$
then
$r := 0$
end

b_on
when
$b = 0$
$s = 0$
then
$b, cb := 1, cb + 1$
end

b_off
when
$b = 1$
$s = 1$
then
$b := 0$
end

s_on
when
$s = 0$
$b = 1$
then
$s, cs := 1, cs + 1$
end

s_off
when
$s = 1$
$b = 0$
then
$s := 0$
end

We now refine these patterns by introducing our new requirement

dbl1_1: $s = 1 \Rightarrow r = 1$

The only events which might cause any problem in proving this invariant are event s_on (setting s to 1) and r_off (setting r to 0). In order to solve this problem, it seems sufficient to add the guards $r = 1$ and $s = 0$ to events s_on and r_off respectively:

s_on
when
$s = 0$
$b = 1$
$\underline{r = 1}$
then
$s, cs := 1, cs + 1$
end

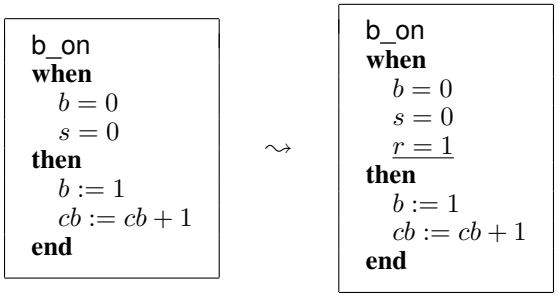
r_off
when
$r = 1$
$a = 0$
$\underline{s = 0}$
then
$r := 0$
end

But, as indicated above, *we do not want to touch these reacting events*. In order to obtain the same effect, it is sufficient to add the following invariants:

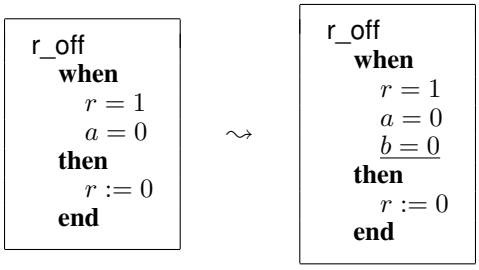
dbl1_2: $b = 1 \Rightarrow r = 1$

dbl1_3: $a = 0 \Rightarrow s = 0$

In order to maintain invariant **dbl1_2**, we have to modify event b_on by adding the guard $r = 1$ to it since it sets b to 1:



To maintain invariant **dbl1_2** we have also to add the guard $b = 0$ to event r_off since it sets r to 0:



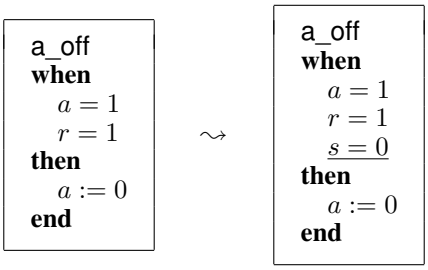
But, again, *we do not want to touch this reacting event* so that we introduce the following invariant:

dbl1_4: $a = 0 \Rightarrow b = 0$

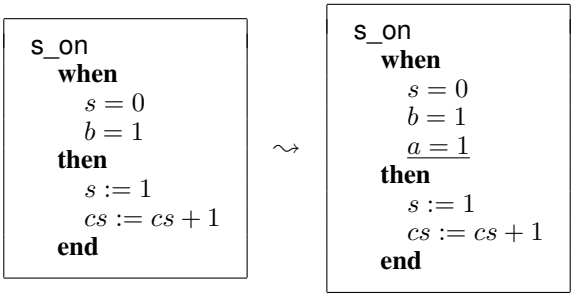
In order to maintain invariant **dbl1_3**, that is:

dbl1_3: $a = 0 \Rightarrow s = 0$

we have to refine event a_off as follows (guard strengthening):



We have also to refine event s_on as follows (guard strengthening)



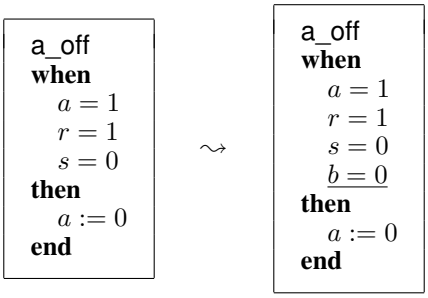
But, again, we do not want to touch this event, so that we have to introduce the following invariant:

$$b = 1 \;\Rightarrow\; a = 1$$

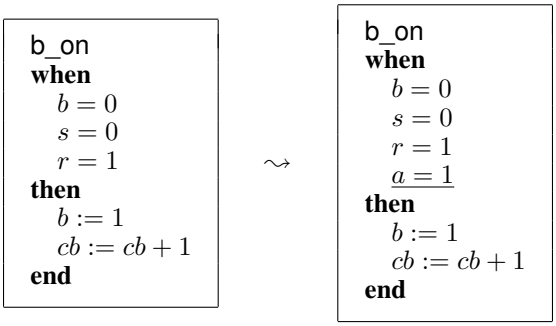
Fortunately, this is exactly **dbl1_4** contraposed

dbl1_4: $a = 0 \;\Rightarrow\; b = 0$

In order to maintain invariant **dbl1_4**, we have to refine a_off again



And also event b_on again:



Now we have obtained the desired effect, namely that of weakly synchronizing the reactions r and s by acting on their respective actions a and b. This is indicated in the diagram of Fig. 21.

Here is a summary of the introduced invariants:

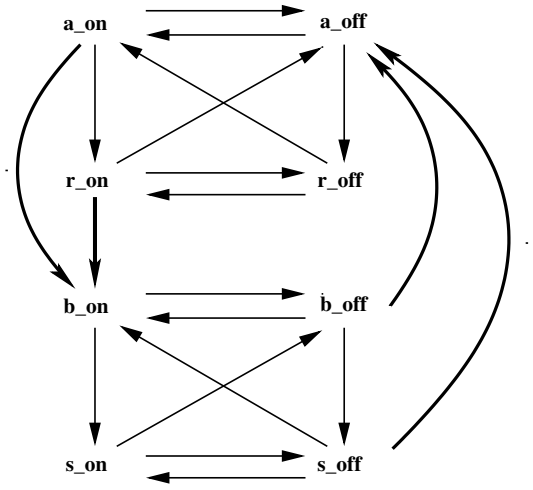


Fig. 21. Weak Synchronizing two Strongly Synchronized Action-reactions

dbl1_1: $s = 1 \;\Rightarrow\; r = 1$

dbl1_2: $b = 1 \;\Rightarrow\; r = 1$

dbl1_3: $a = 0 \;\Rightarrow\; s = 0$

dbl1_4: $a = 0 \;\Rightarrow\; b = 0$

Here is also a summary of the modified events a_off and b_on (where we have removed the incrementation of counter cb):

a_off
when
$a = 1$
$r = 1$
$\underline{s = 0}$
$\underline{b = 0}$
then
$a := 0$
end

b_on
when
$b = 0$
$s = 0$
$\underline{r = 1}$
$\underline{a = 1}$
then
$b := 1$
end

Note that the four previous invariants can be equivalently reduced to the following unique one, which can be "read" now on the diagram of Fig. 22.

dbl1_5: $b = 1 \;\vee\; s = 1 \;\Rightarrow\; a = 1 \;\wedge\; r = 1$

9 Third Refinement: Constraining the Clutch and the Motor

Coming back to our development, we incorporate now the following requirement:

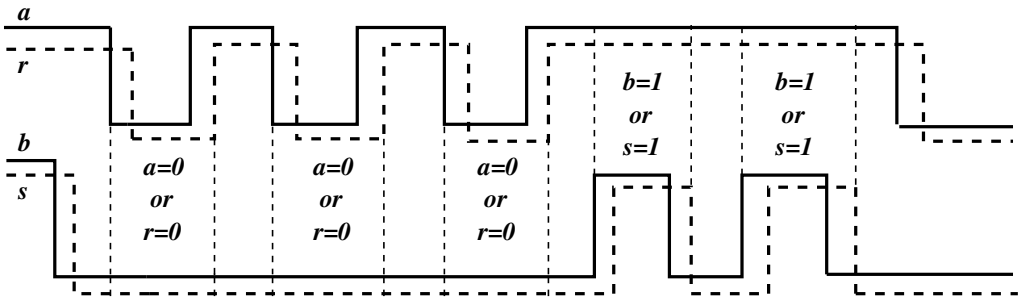


Fig. 22. $b = 1 \ \vee \ s = 1 \ \Rightarrow \ a = 1 \ \wedge \ r = 1$

When the clutch is engaged, the motor must work	SAF_1

This can be formalized by means of the following new invariant

inv3_1:	$clutch\_sensor = engaged \ \Rightarrow \ motor\_sensor = working$

This is an instance of the design pattern developed in section 8, which we instantiate as follows:

a	$\leadsto$	$motor\_actuator$	a_on	$\leadsto$	treat_push_start_motor_button
r	$\leadsto$	$motor\_sensor$	a_off	$\leadsto$	treat_push_stop_motor_button
0	$\leadsto$	$stopped$	r_on	$\leadsto$	Motor_start
1	$\leadsto$	$working$	r_off	$\leadsto$	Motor_stop
b	$\leadsto$	$clutch\_actuator$	b_on	$\leadsto$	treat_start_clutch
s	$\leadsto$	$clutch\_sensor$	b_off	$\leadsto$	treat_stop_clutch
0	$\leadsto$	$disengaged$	s_on	$\leadsto$	Clutch_start
1	$\leadsto$	$engaged$	s_off	$\leadsto$	Clutch_stop

The invariant are as follows:

dbl1_1: $s = 1 \Rightarrow r = 1$	**inv3_1:** $clutch\_sensor = engaged \Rightarrow motor\_sensor = working$
dbl1_2: $b = 1 \Rightarrow r = 1$	**inv3_2:** $clutch\_actuator = engaged \Rightarrow motor\_sensor = working$
dbl1_3: $a = 0 \Rightarrow s = 0$	**inv3_3:** $motor\_actuator = stopped \Rightarrow clutch\_sensor = disengaged$
dbl1_4: $a = 0 \Rightarrow b = 0$	**inv3_4:** $motor\_actuator = stopped \Rightarrow clutch\_actuator = disengaged$

The two modified events are as follows:

b_on
when
$b = 0$
$s = 0$
$r = 1$
$a = 1$
then
$b := 1$
end

treat_start_clutch
when
$clutch\_actuator = disengaged$
$clutch\_sensor = disengaged$
$motor\_sensor = working$
$motor\_actuator = working$
then
$clutch\_actuator := engaged$
end

a_off
when
$a = 1$
$r = 1$
$s = 0$
$b = 0$
then
$a := 0$
end

treat_stop_motor
when
$stop\_motor\_impulse = \text{FALSE}$
$stop\_motor\_button = \text{TRUE}$
$motor\_actuator = working$
$motor\_sensor = working$
$clutch\_sensor = disengaged$
$clutch\_actuator = disengaged$
then
$motor\_actuator := stopped$
$stop\_motor\_impulse := \text{TRUE}$
end

10 Fourth Refinement: Connecting the Controller to the Door

10.1 Copying

We copy (after renaming "motor" to "door") what has been done in the initial model (section 6)

10.2 Summary of the Events

- Environment
    - motor_start
    - motor_stop
    - clutch_start
    - clutch_stop
    - door_close
    - door_open
    - push_start_motor_button
    - release_start_motor_button
    - push_stop_motor_button
    - release_stop_motor_button

- Controller
    - treat_push_start_motor_button
    - treat_push_start_motor_button_false
    - treat_push_stop_motor_button
    - treat_push_stop_motor_button_false
    - treat_release_start_motor_button
    - treat_release_stop_motor_button
    - treat_start_clutch
    - treat_stop_clutch
    - treat_close_door
    - treat_open_door

11 Fifth Refinement: Constraining the Clutch and the Door

We now incorporate the following additional safety constraint:

When the clutch is engaged, the door must be closed	SAF_2

This is done by copying (after renaming "motor" to "door") what has been done in the third model (section 9). At this point, we figure out that we have forgotten something concerning the door: clearly it must be open when the motor is stopped so that the user can replace the part or change the tool. This can be stated by adding the following requirement:

When the motor is stopped, the door must be open	SAF_3

It is interesting to present this requirement under its equivalent contraposed form SAF_3':

When the door is closed, the motor must work	SAF_3'

We can take care of this requirement by copying (after renaming "clutch" to "door") what has been done in the third model (section 9). It is interesting to put now the two previous requirements SAF_1 and SAF_2 next to SAF_3':

When the clutch is engaged, the motor must work	SAF_1

When the clutch is engaged, the door must be closed	SAF_2

This shows that SAF_1 *is redundant* as it can be obtained by combining SAF_2 and SAF_3'! The moral of the story is that the third refinement (section 9) can be removed completely, and thus our refinement strategy (section 4) could have been simplified as follows:

- Initial model: Connecting the controller to the motor,
- 1st refinement: Connecting the motor button to the controller,
- 2nd refinement: Connecting the controller to the clutch,
- 3rd (4th) refinement: Connecting the controller to the door,
- 4th (5th) refinement: Constraining the clutch and the door and the motor and the door,
- 5th (6th) refinement: More constraints between the clutch and the door,
- 6th (7th) refinement: Connecting the clutch button to the controller.

12 Another Design Pattern: Strong Synchronization of Two Strong Reactions

12.1 Introduction.

We consider now the following requirements FUN_3 and FUN_4 concerning the relationship between the clutch and the door:

When the clutch is disengaged, the door cannot be closed several times

When the door is closed, the clutch cannot be disengaged several times

This is also a case of synchronization between two strong reactions. This time however the weak synchronization is not sufficient any more: we need a strong synchronization. This is indicated in the diagram of Fig. 23.The full picture is indicated on Fig. 24.

12.2 Modeling

The modeling of this new constraints will be presented as a refinement of the "weak-strong" model of section 8. In order to formalize this new kinds of synchronization, we have to consider again the counters ca, cr, cb, and cs as indicated in Fig. 25.

What we want to achieve is expressed in the following properties:

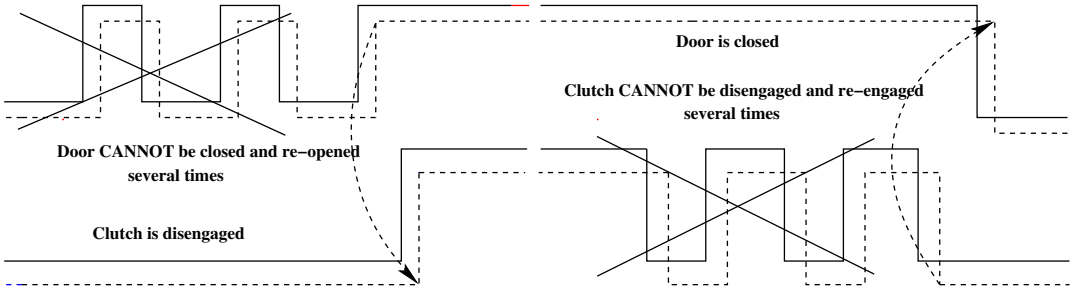


Fig. 23. Strong Synchronization between the Clutch and the Door

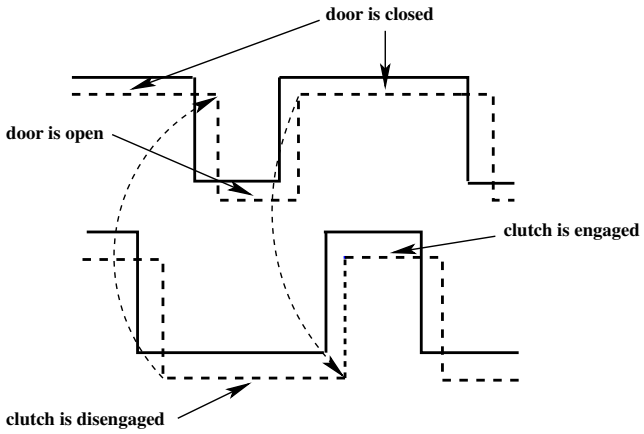


Fig. 24. The Full Picture of Strong Synchronization

$$ca = cb \ \vee \ ca = cb + 1$$
$$cr = cs \ \vee \ cr = cs + 1$$

Let us first treat the case of counters ca and cb as illustrated in Fig 26. It seems that the condition $ca = cb + 1$ is implied by the condition $a = 1 \wedge b = 0$ as indicated in fig 27. However, this guess is wrong as illustrated on Fig 28. The solution consists in introducing a new variable m as indicated in Fig. 29.

variables: $\ldots$, m

dbl2_1: $m \in \{0, 1\}$

dbl2_2: $m = 1 \ \Rightarrow \ ca = cb + 1$

dbl2_3: $m = 0 \ \Rightarrow \ ca = cb$

Let us now treat the case of counters cr and cs as indicated on Fig. 30. It seems that the condition $cr = cs + 1$ is implied by the condition $r = 1 \wedge s = 0$ as indicated in fig 31. But again this guess is wrong

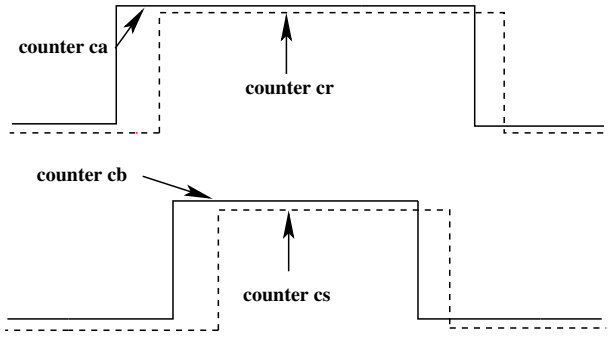


Fig. 25. The Counters

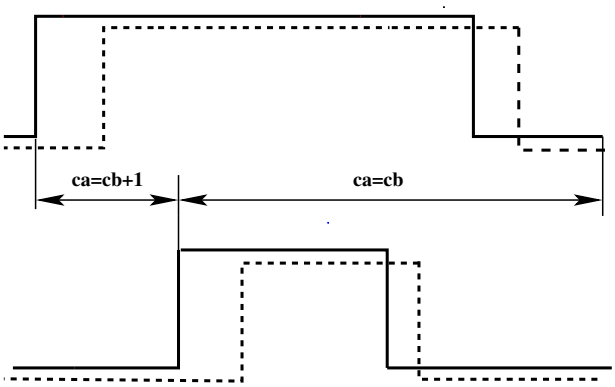


Fig. 26. Counters ca and cb

as illustrated on Fig 32. The solution is shown on Fig. 33. This lead to the following additional invariants **dbl2_4** and **dbl2_5**:

dbl2_1: $m \in \{0, 1\}$

dbl2_2: $m = 1 \Rightarrow ca = cb + 1$

dbl2_3: $m = 0 \Rightarrow ca = cb$

dbl2_4: $r = 1 \wedge s = 0 \wedge (m = 1 \vee b = 1) \Rightarrow cr = cs + 1$

dbl2_5: $r = 0 \vee s = 1 \vee (m = 0 \wedge b = 0) \Rightarrow cr = cs$

Let us now turn our attention to the modified events. This is indicated on Fig. 34. As can be seen the concerned events are a_on, b_on, and a_off. Here are the proposals for these events:

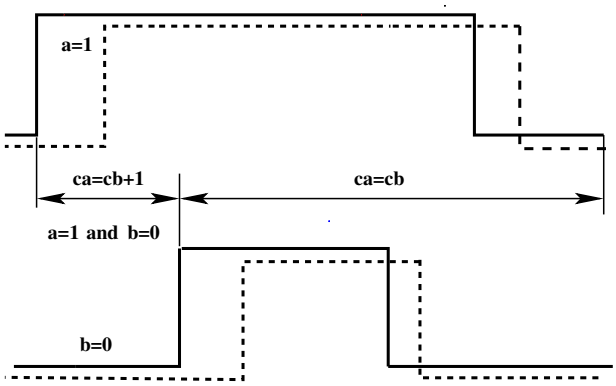


Fig. 27. A Guess

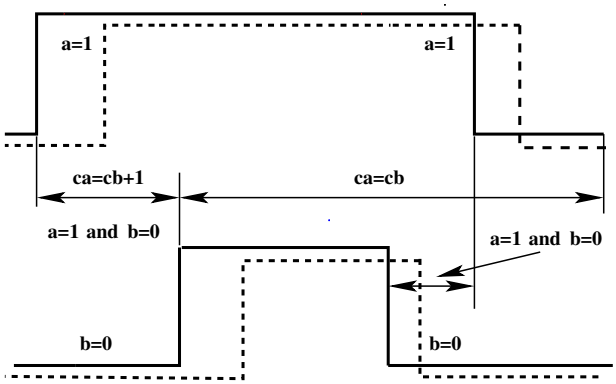


Fig. 28. The Guess is Wrong

a_on
when
$a = 0$
$r = 0$
then
$a := 1$
$ca := ca + 1$
$m := 1$
end

b_on
when
$r = 1$
$a = 1$
$b = 0$
$s = 0$
$m = 1$
then
$b := 1$
$cb := cb + 1$
$m := 0$
end

a_off
when
$a = 1$
$r = 1$
$b = 0$
$s = 0$
$m = 0$
then
$a := 0$
end

It remains now for us to do the proofs. Similar techniques as the ones used in sections 2 and 8 lead us to define the following additional invariants **dbl2_6** and **dbl2_7**:

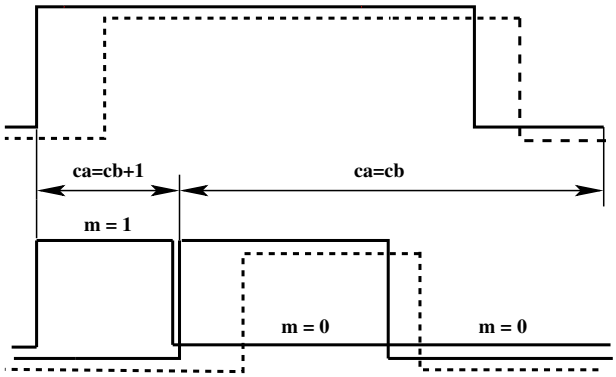


Fig. 29. Introducing a New Variable m

dbl2_1: $m \in \{0, 1\}$

dbl2_2: $m = 1 \Rightarrow ca = cb + 1$

dbl2_3: $m = 0 \Rightarrow ca = cb$

dbl2_4: $r = 1 \wedge s = 0 \wedge (m = 1 \vee b = 1) \Rightarrow cr = cs + 1$

dbl2_5: $r = 0 \vee s = 1 \vee (m = 0 \wedge b = 0) \Rightarrow cr = cs$

dbl2_6: $m = 0 \Rightarrow a = 0 \vee r = 1$

dbl2_7: $m = 1 \Rightarrow b = 0 \wedge s = 0 \wedge a = 1$

After this last invariant extension, the proofs are done easily.

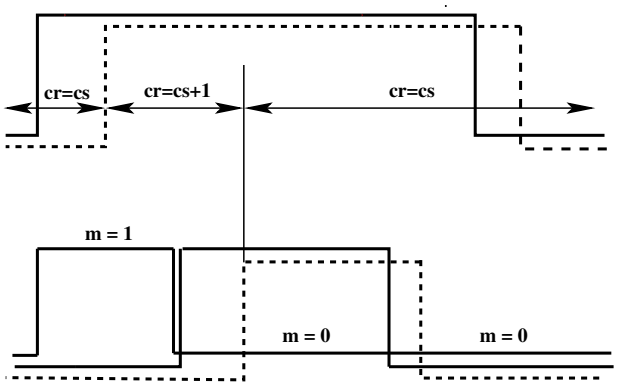


Fig. 30. Counters cr and cs

.

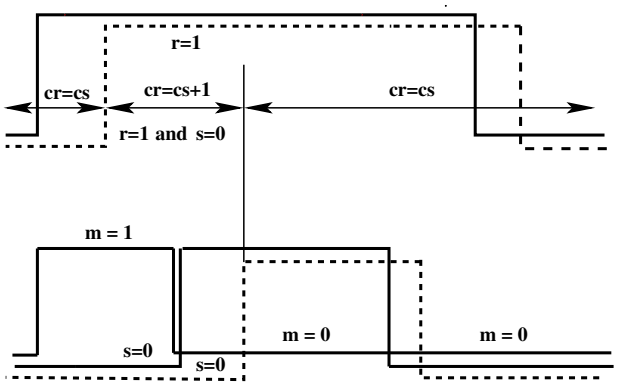


Fig. 31. A Guess

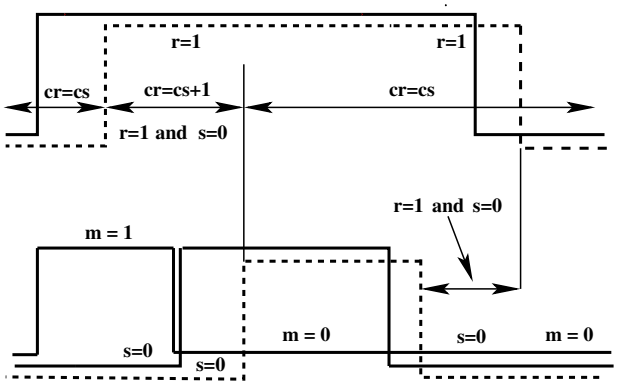


Fig. 32. The Guess is Wrong

13 Sixth Refinement: More Constraints between Clutch and Door

It remains now for us to instantiate the "strong-strong" pattern of previous section. It is done as follows:

a	$\rightsquigarrow$	$door\_actuator$	b	$\rightsquigarrow$	$clutch\_actuator$	a_on	$\rightsquigarrow$	treat_close_door
r	$\rightsquigarrow$	$door\_sensor$	s	$\rightsquigarrow$	$clutch\_sensor$	a_off	$\rightsquigarrow$	treat_open_door
0	$\rightsquigarrow$	$open$	0	$\rightsquigarrow$	$disengaged$	b_on	$\rightsquigarrow$	treat_start_clutch
1	$\rightsquigarrow$	$closed$	1	$\rightsquigarrow$	$engaged$			

This leads to the following event instantiations:

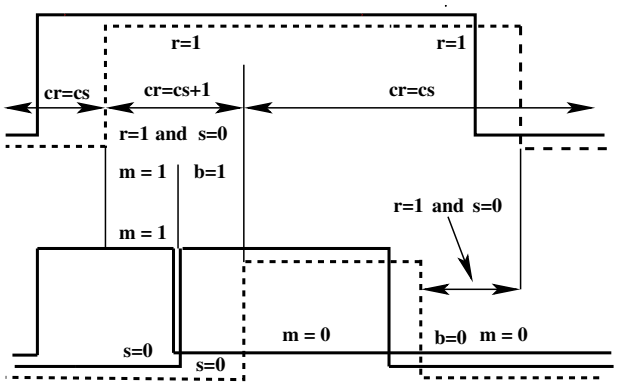


Fig. 33. The Solution

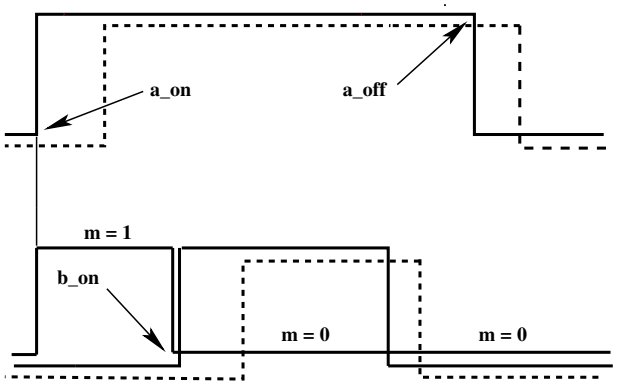


Fig. 34. The Events

a_on
when
$a = 0$
$r = 0$

then
$a := 1$
$m := 1$
end

treat_close_door
when
$door\_actuator = open$
$door\_sensor = open$
$motor\_actuator = working$
$motor\_sensor = working$
then
$door\_actuator := closed$
$m := 1$
end

b_on
when
$b = 0$
$s = 0$
$r = 1$
$a = 1$
$m = 1$
then
$b := 1$
$m := 0$
end

treat_start_clutch
when
$motor\_actuator = working$
$motor\_sensor = working$
$clutch\_actuator = disengaged$
$clutch\_sensor = disengaged$
$door\_sensor = closed$
$door\_actuator = closed$
$m = 1$
then
$clutch\_actuator := engaged$
$m := 0$
end

a_off
when
$a = 1$
$r = 1$
$s = 0$
$b = 0$
$m = 0$
then
$a := 0$
end

treat_open_door
when
$door\_actuator = closed$
$door\_sensor = closed$
$clutch\_sensor = disengaged$
$clutch\_actuator = disengaged$
$m = 0$
then
$door\_actuator := open$
end

The final synchronization of the door and the clutch is shown on Fig. 35. In this figure, the underlined events are environment events.

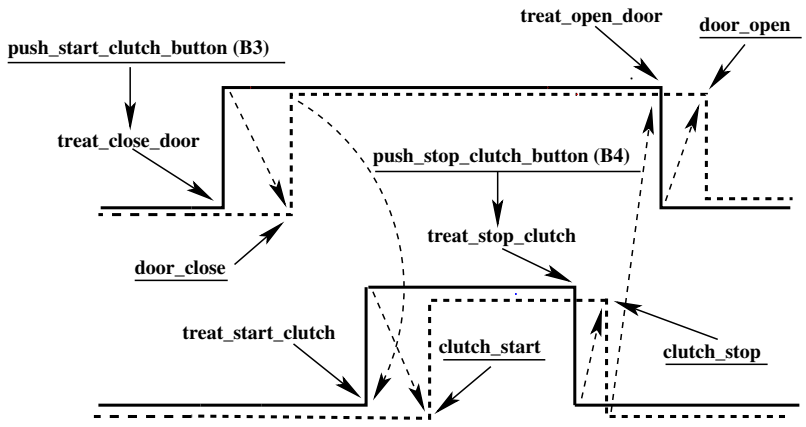


Fig. 35. The Final Synchronization of the Door and the Clutch

.

14 Seventh Refinement: Connecting the Controller to the Clutch Buttons

14.1 Copying

We simply connect button B3 to the event treat_close_door and button B4 to the events treat_stop_clutch.

14.2 Summary of Events

- Environment
    - motor_start
    - motor_stop
    - clutch_start
    - clutch_stop
    - door_close
    - door_open
    - push_start_motor_button
    - release_start_motor_button
    - push_stop_motor_button
    - release_stop_motor_button
    - push_start_clutch_button
    - release_start_clutch_button
    - push_stop_clutch_button
    - release_stop_clutch_button

- Controller
    - treat_push_start_motor_button
    - treat_push_start_motor_button_false
    - treat_push_stop_motor_button
    - treat_push_stop_motor_button_false
    - treat_release_start_motor_button
    - treat_release_stop_motor_button
    - treat_start_clutch
    - treat_stop_clutch
    - treat_close_door
    - treat_open_door
    - treat_close_door_false