

ICTAC 2008

Using **Design Patterns** in Formal Methods:
an **Event-B** Approach

Jean-Raymond Abrial (ETHZ)

Thai Son Hoang (ETHZ)

September 1st 2008

-
1. **Design patterns** in formal developments
 2. A quick introduction to **Event-B**
 3. An **example**
 4. A **demo** (time permitting)

1. Design Patterns in Formal Developments

- VDM, Z, and B are all formal methods with "proofs and refinement"
- They introduce sophisticated mathematical notations
- They allow to develop complex industrial (embedded) systems

- But they are rather **poor** from a **methodological point of view**
- **Component** based approaches have **not proved to be that useful**
- People are looking for more elaborate **methodological** approaches

- We want to build systems which are **correct by construction**
- We want to have **systematic methods** for doing so
- We wonder whether we could use **design patterns**
- "**Design Patterns: Elements of Reusable Object-Oriented Software**"
published in 1994 (by E. Gamma et al)

- This is an **engineering** concept
- It can be used **outside OO**
- The goal of each DP is **to solve a certain category of problems**
- But the design pattern has to be **adapted** to the problem at hand
- **Is it compatible with formal developments?**

2. A Quick Introduction to **Event-B**

- Event-B is not a programming language (even very abstract)
- Event-B is a notation used for developing mathematical models
- Mathematical models of discrete transition systems
- <http://www.event-b.org>

- Such **models**, once finished, can be used to **eventually construct**:
 - **sequential** programs,
 - **distributed** programs,
 - **concurrent** programs,
 - **electronic circuits**,
 - **large systems** involving a possibly **fragile environment**,
 - ...
- The underlined statement is an **important** case.

Action Systems developed by the Finnish school (Turku):

R.J.R. Back and R. Kurki-Suonio

Decentralization of Process Nets with Centralized Control.

2nd ACM SIGACT-SIGOPS Symposium

Principles of Distributed Computing (1983)

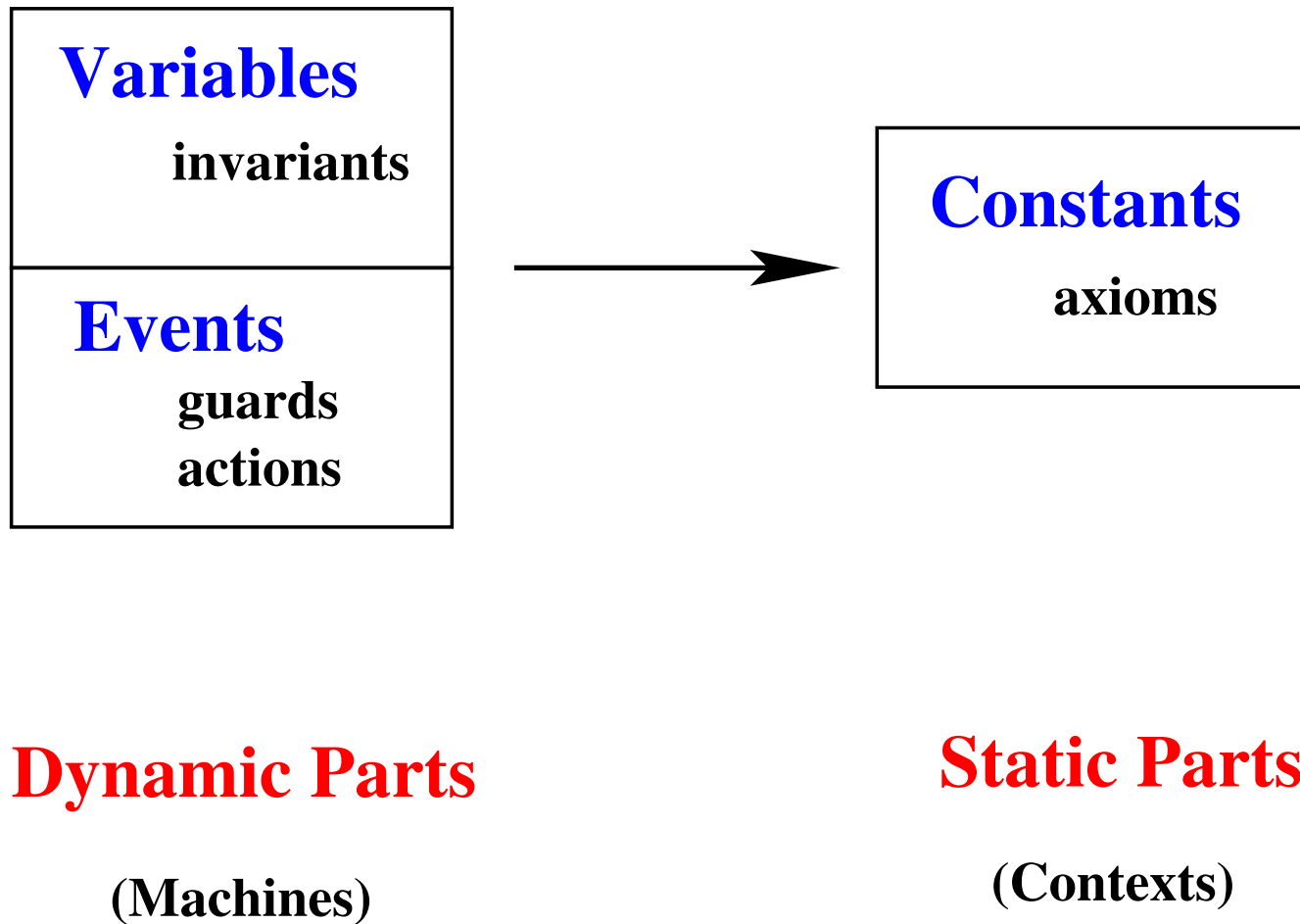
M.J. Butler

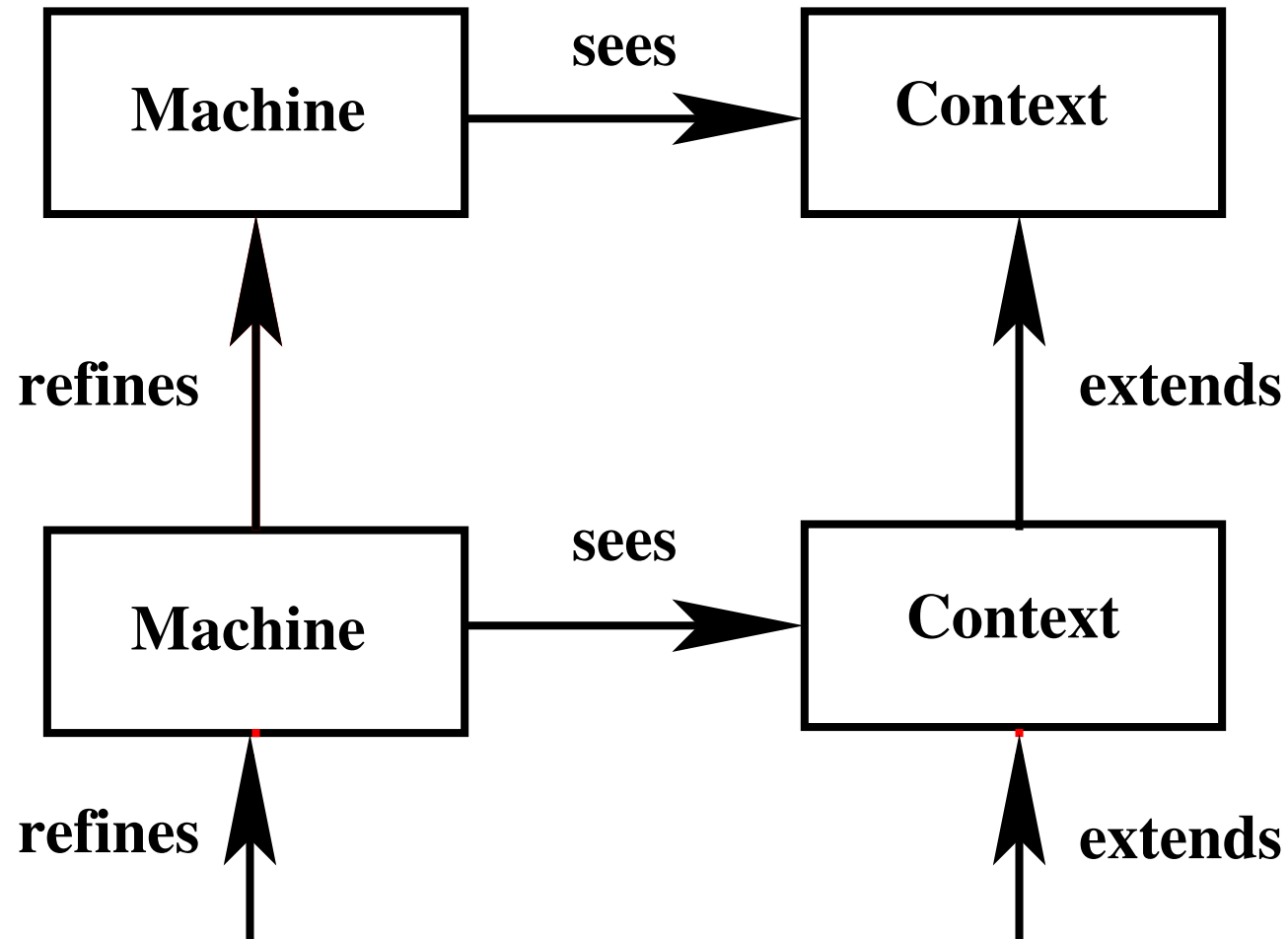
Stepwise Refinement of Communicating Systems.

Science of Computer Programming (1996)

- A discrete model is made of a **state** with **constants** and **variables**
- Constants and variables are linked by some **axioms** and **invariants**
- Axioms and invariants are written using **set-theoretic expressions**

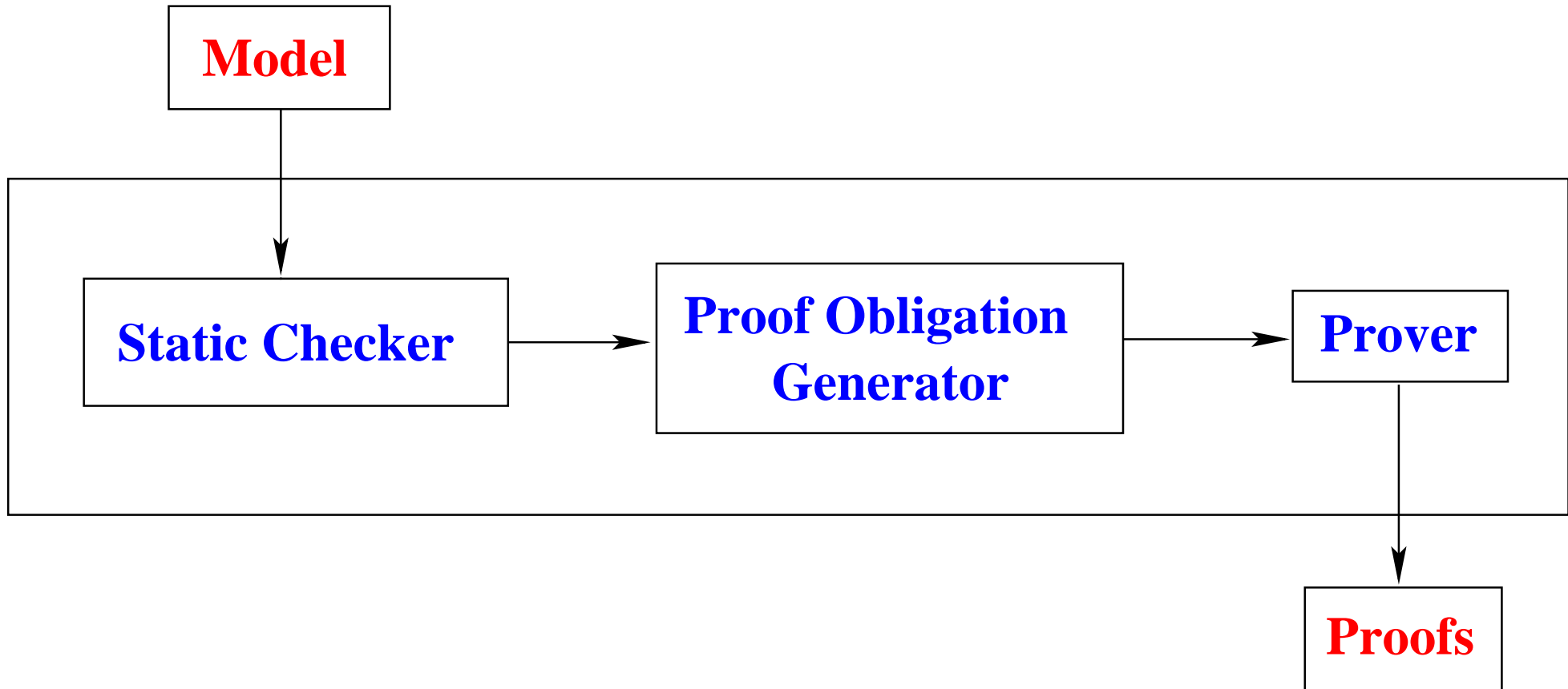
-
- A discrete model is also made of **events** with **guards** and **actions**
 - The **guard** denotes the **enabling condition** of the event
 - The **action** denotes the way the **state is modified** by the event
 - Guards and actions are written using **set-theoretic expressions**





-
- An Event-B mathematical model is subjected to **various proofs**:
 - **Invariant preservation**
 - **Correct refinement**
 - ...
 - Such proof statements are **determined by a tool** (POG)
 - An industrial project may require **several thousands of proofs**.

- A tool has been constructed: the **Rodin Platform** (open source)
- Financed by an EU project: **Rodin** (2004-2007)
- Another EU Project is progressing now: **Deploy** (2008-2011)



3. An Example

3.1. Presentation of the example

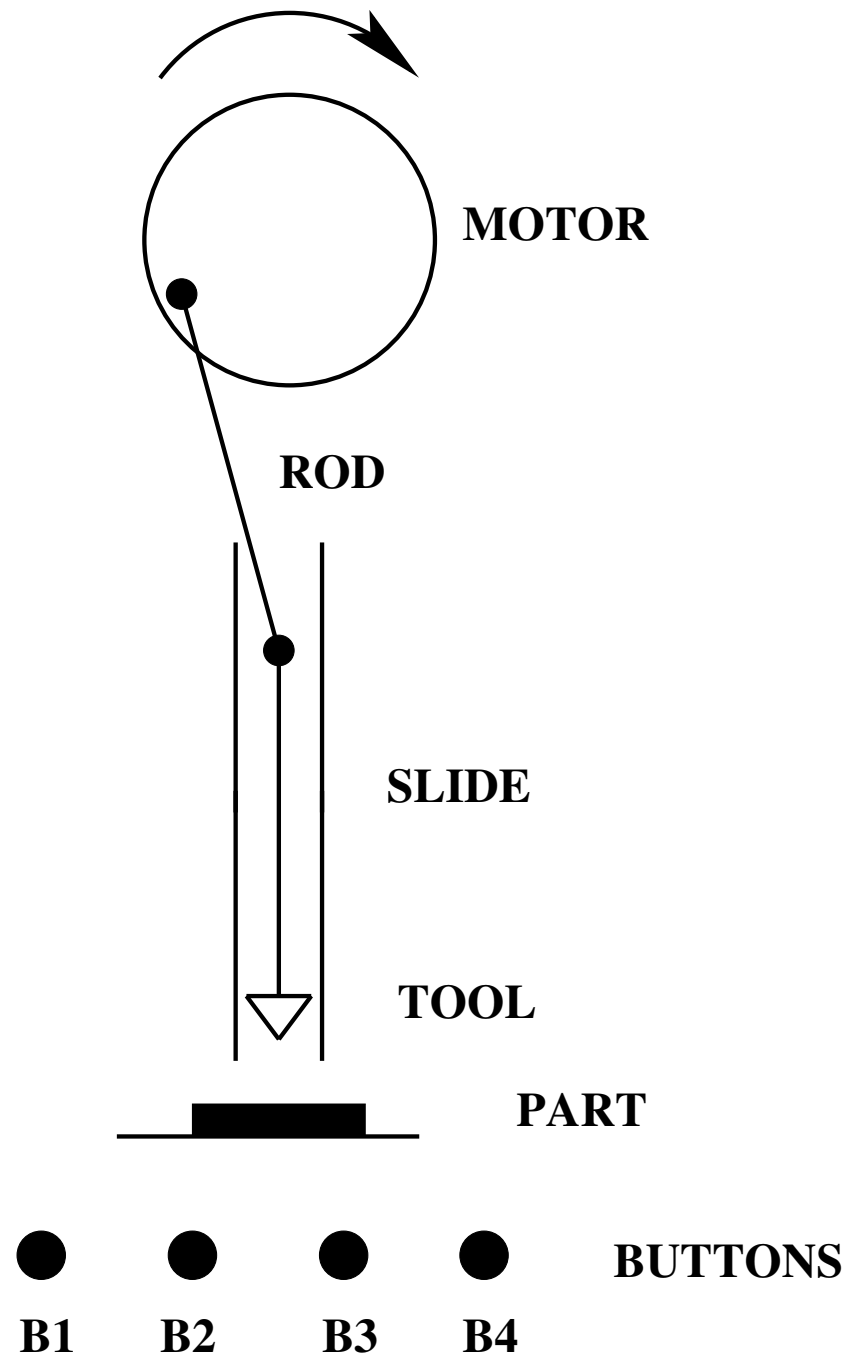
3.2. Why using design pattern in this example

3.3. Some design patterns

3.4. Insights on the formal development with design pattern

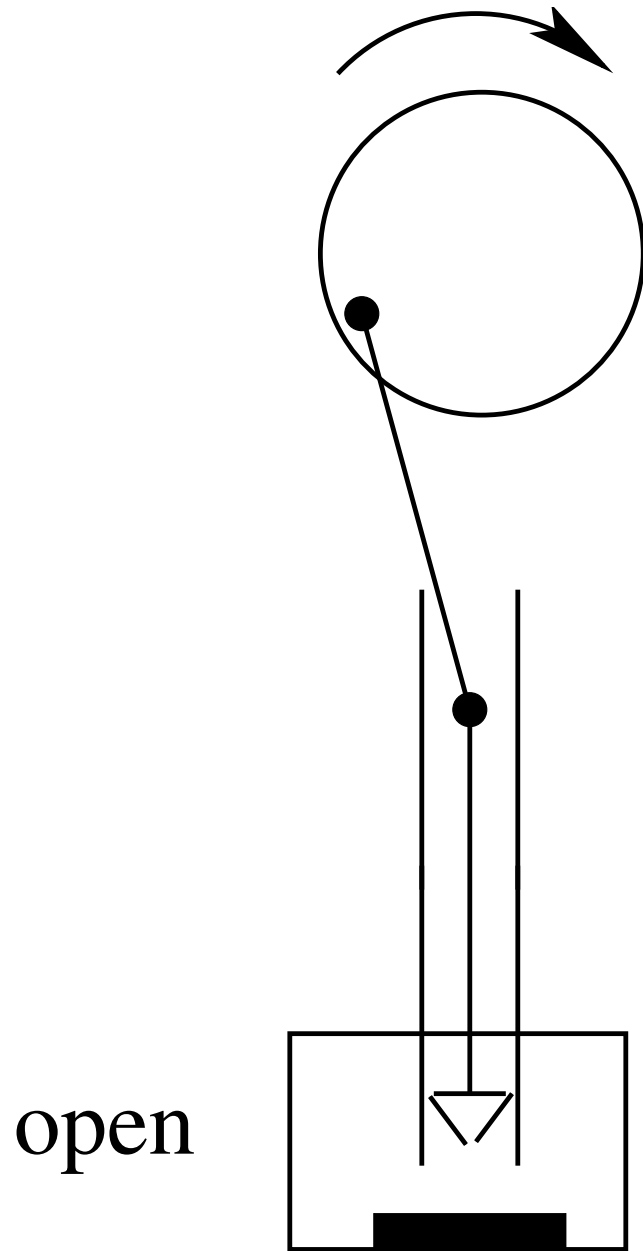
3.1. Presentation of the Example

- A mechanical **press controller**
- **Adapted** from a **real system**

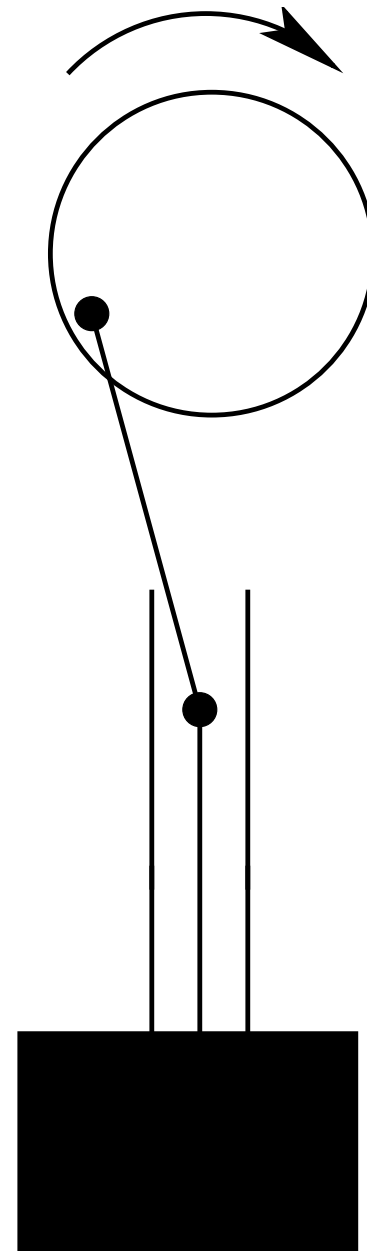


- Button B1: start motor
- Button B2: stop motor
- Button B3: engage clutch
- Button B4: disengage clutch

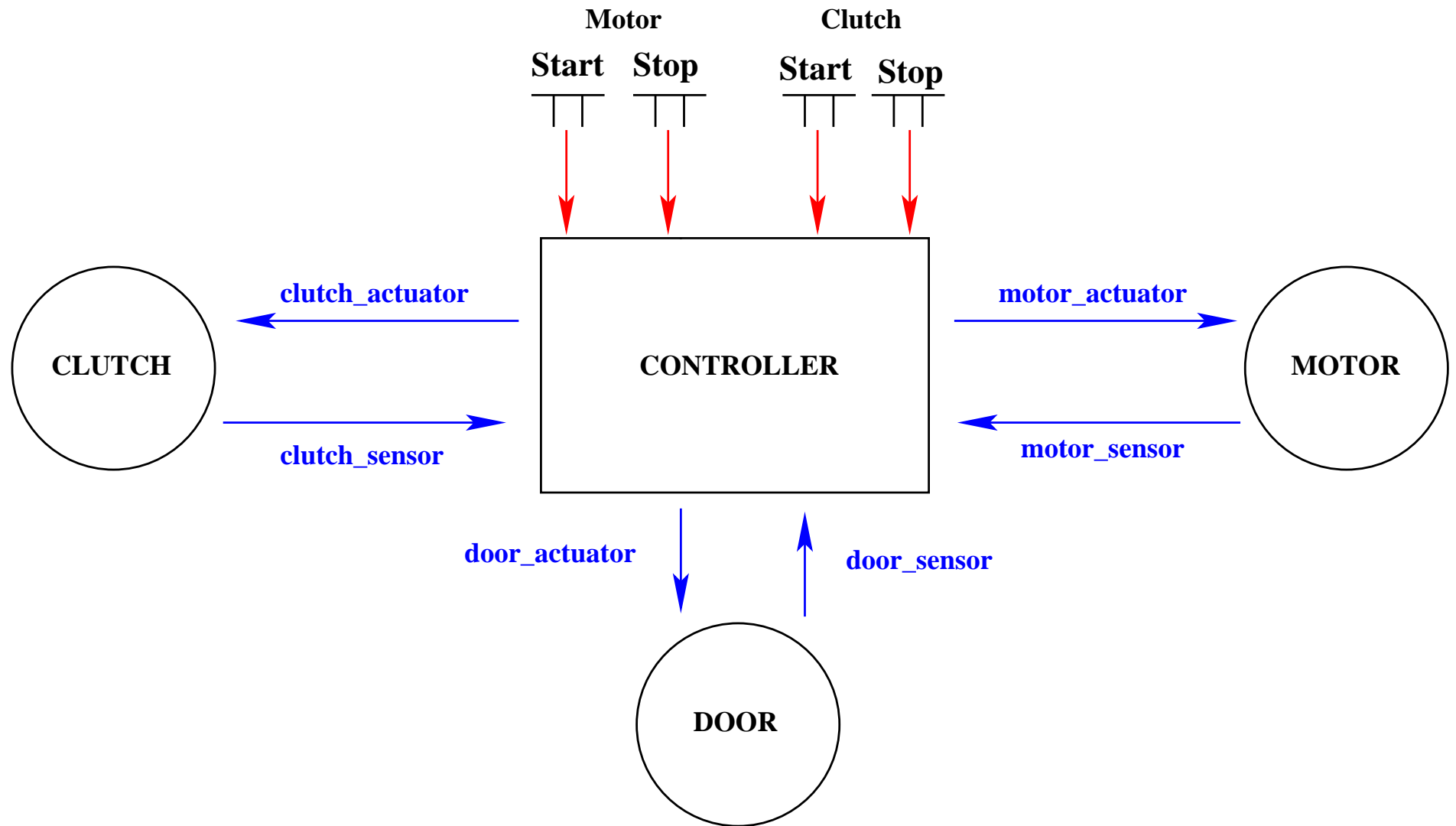
- Action 1: **Change the tool** at the lower extremity of the slide
- Action 2: **Put a part** to be treated under the slide
- Action 3: **Remove the part**
- These actions are all **dangerous**



open



closed



3.2. Why using design patterns on this example

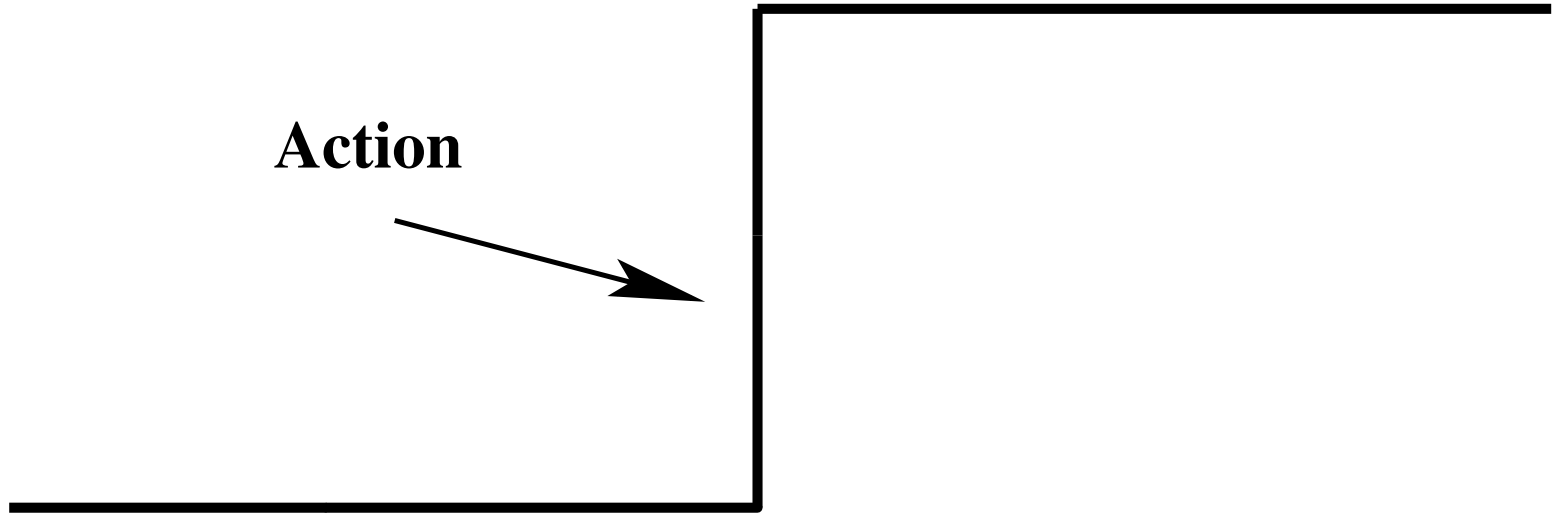
- A number of **similar behaviors**
- Some **complex situations** to handle

- A **specific action** results eventually in having a **specific reaction**:
 - Pushing **button B1** results eventually in **starting the motor**
 - Pushing **button B4** results eventually in **disengaging the clutch**
 - ...

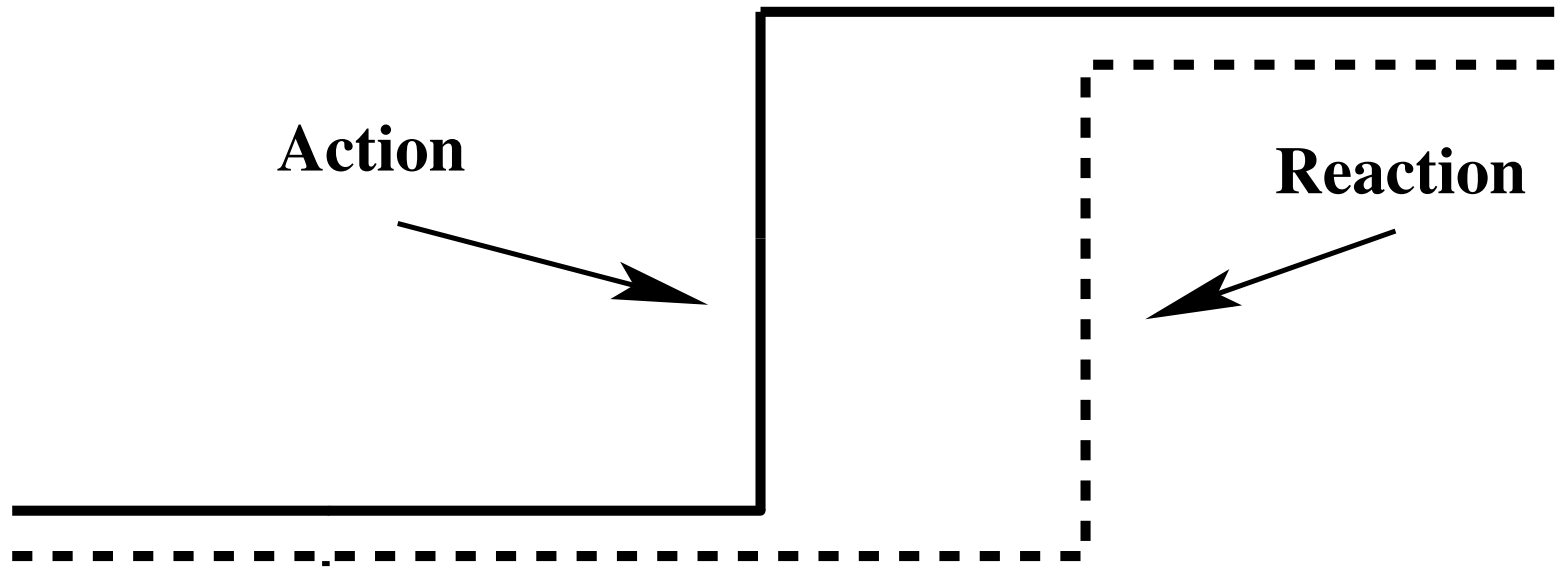
- Correlating **two pieces of equipment**:
 - When the **clutch is engaged** then the **motor must work**
 - When the **clutch is engaged** then the **door must be closed**

- Making an **action dependent** of another one:
 - **Engaging the clutch** implies **closing the door first**
 - **Disengaging the clutch** means **opening the door afterwards**

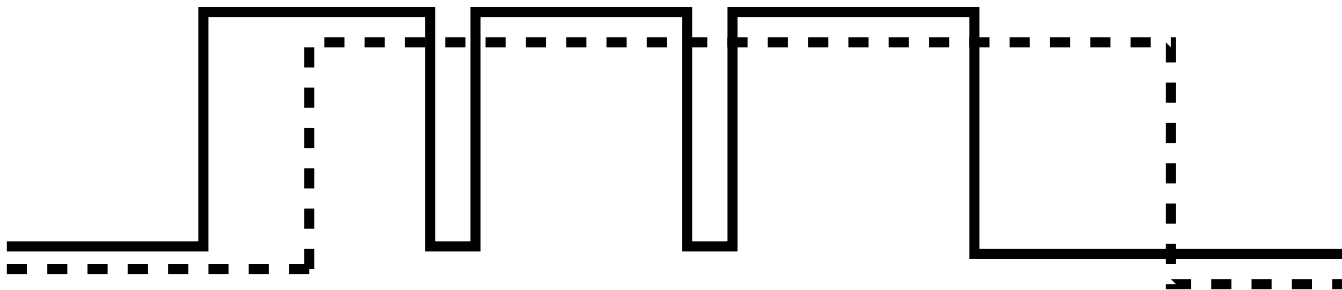
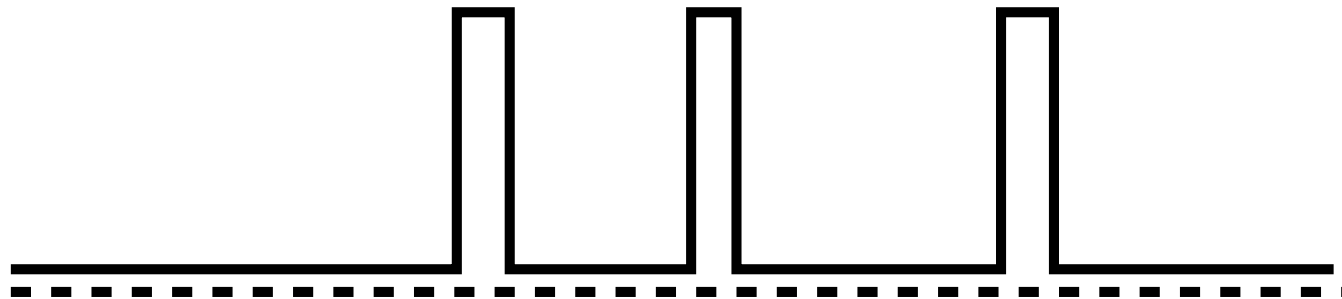
3.3. Some design patterns



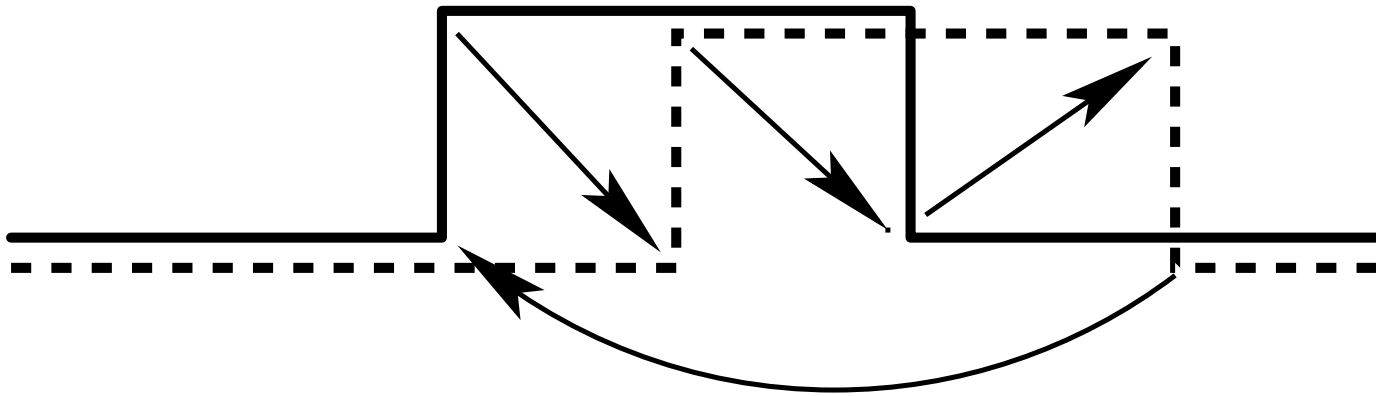
Action



- Sometimes, the reaction has **not enough time** to react
- Because the action moves **too quickly**



- Sometimes, the reaction **always follows** the action
- They are both **synchronized**



- We built first a **model of a weak reaction**
- The **strong reaction will be a refinement** of the weak one

variables: a
 r

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

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

- a denotes the **action**
- r denotes the **reaction**

variables: a
 r
 ca
 cr

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

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

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

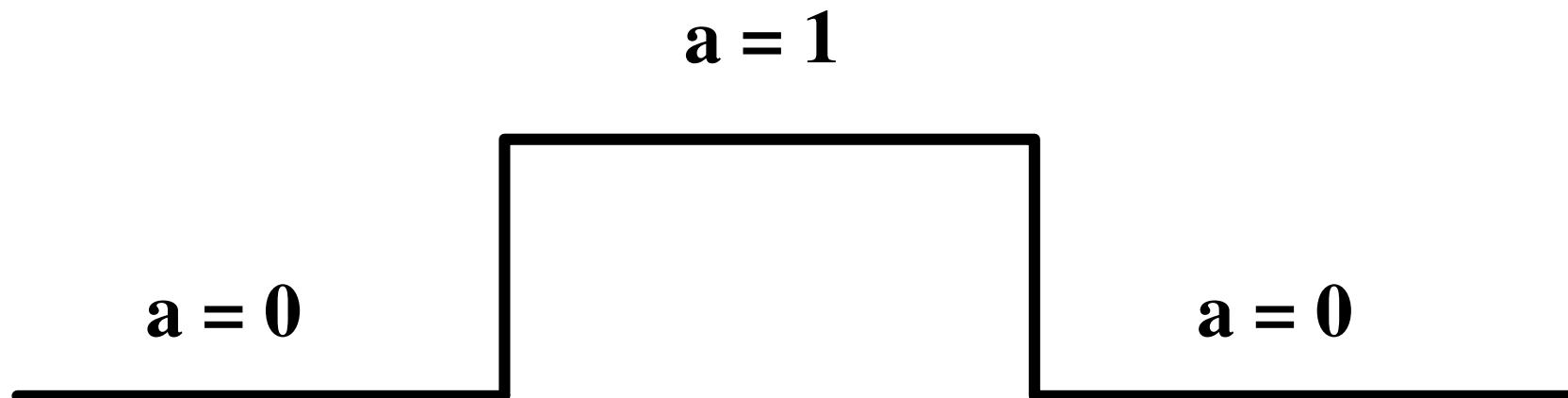
pat0_4: $cr \in \mathbb{N}$

pat0_5: $cr \leq ca$

- ca and cr denote how many times a and r are set to 1
- **pat0_5** formalizes the weak reaction

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

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

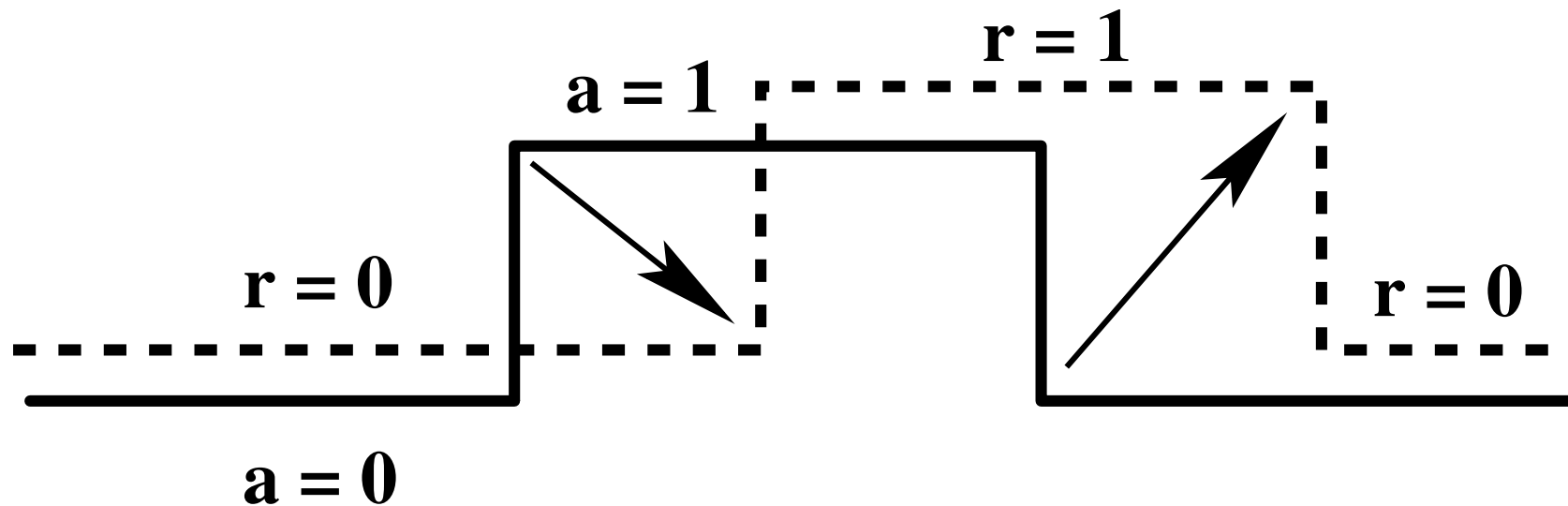


```

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 counters have
been removed

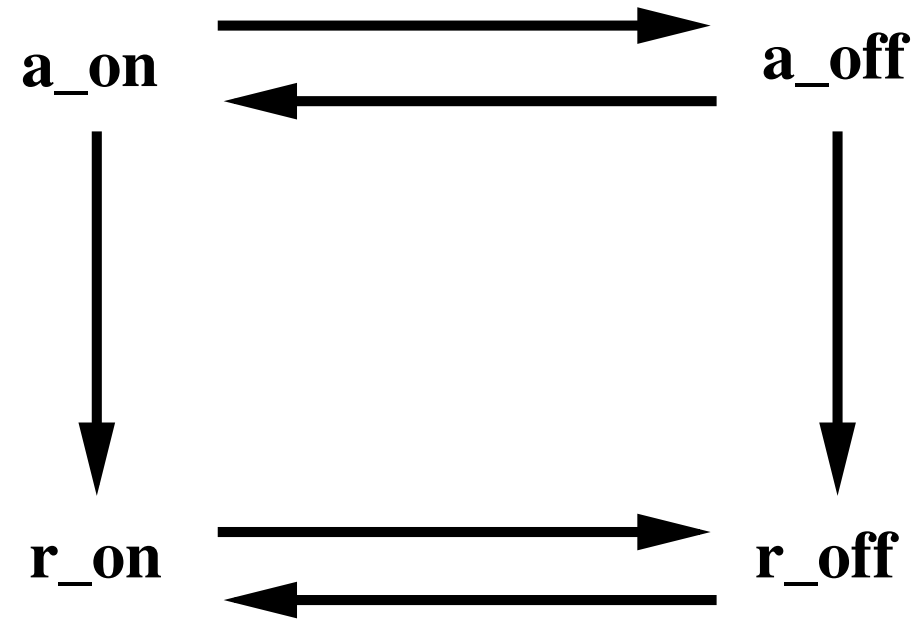
```
init
   $a := 0$ 
   $r := 0$ 
```

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

```
a_off
  when
     $a = 1$ 
  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 counters have
been removed

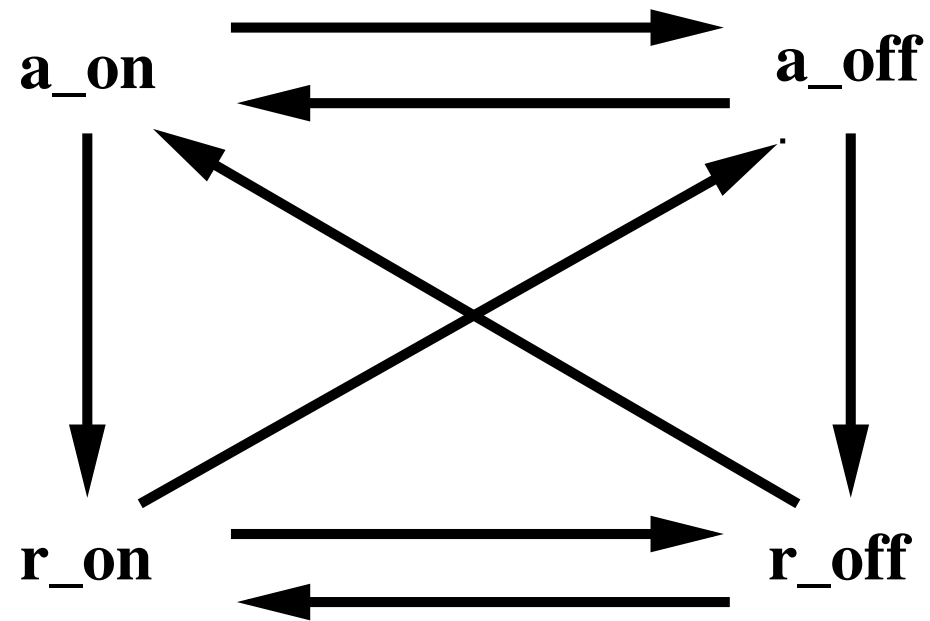
```
init
   $a := 0$ 
   $r := 0$ 
```

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

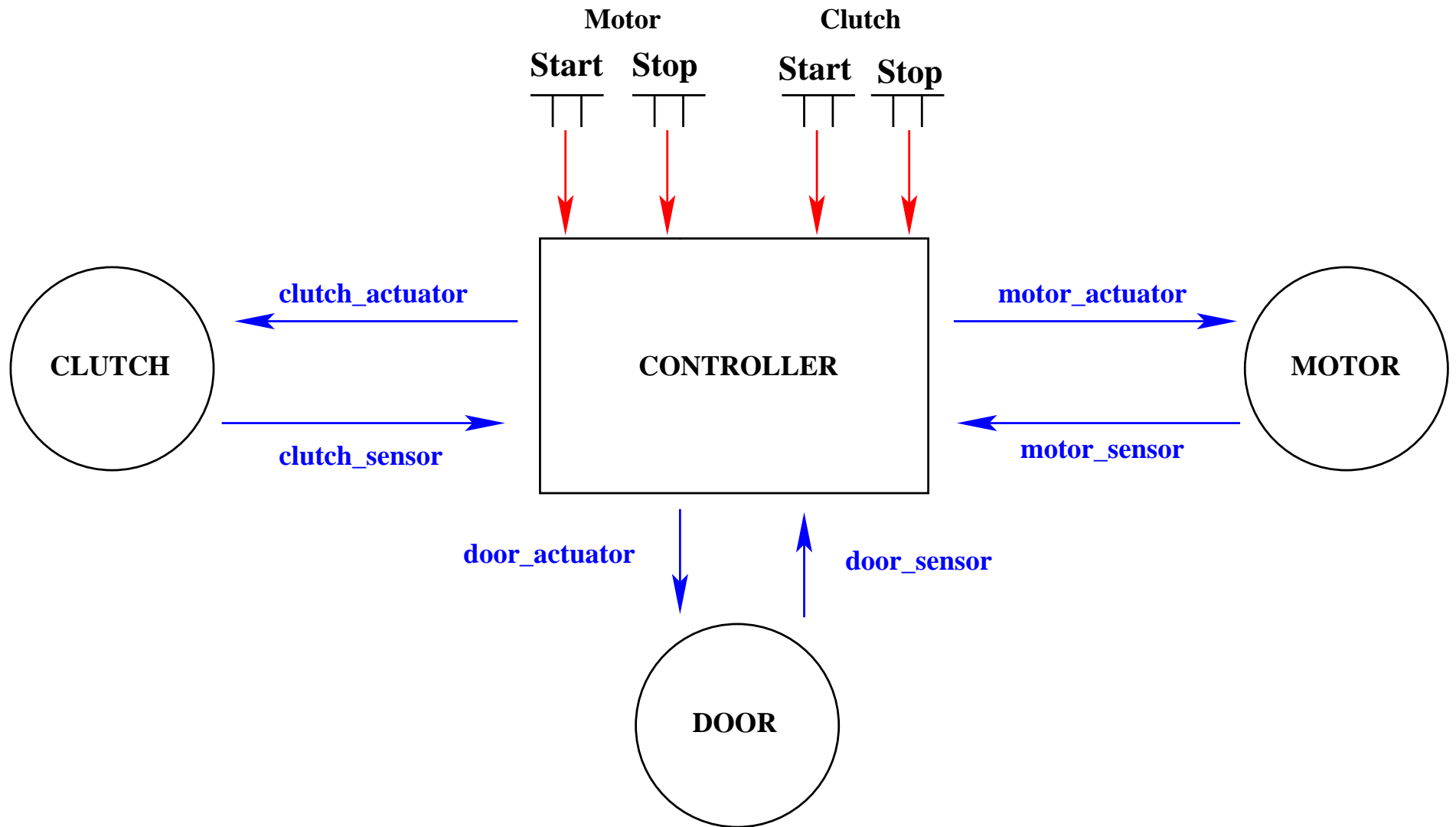
```
a_off
  when
     $a = 1$ 
     $r = 1$ 
  then
     $a := 0$ 
  end
```

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

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



3.4. Insights on the formal development



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 this equipment

EQP_3

Buttons and Controller are weakly synchronized	FUN_1
--	-------

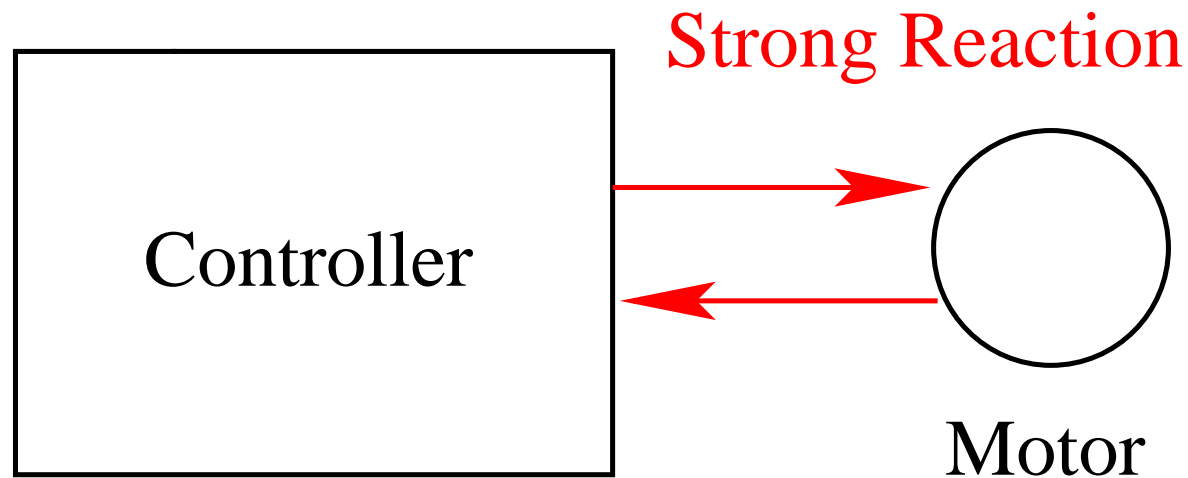
Controller are Equipment are strongly synchronized	FUN_2
--	-------

When the clutch is engaged, the motor must work	SAF_1
---	-------

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

- Initial model: Connecting the **controller to the motor**
- 1st refinement: Connecting the **motor buttons 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 **clutch** and **door**
- 7th refinement: Connecting the **clutch buttons to the controller**



Controller are Equipment are strongly synchronized

FUN_2

The counters have
been removed

```
init
   $a := 0$ 
   $r := 0$ 
```

```
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
```

set: *STATUS*

constants: *stopped*
 working

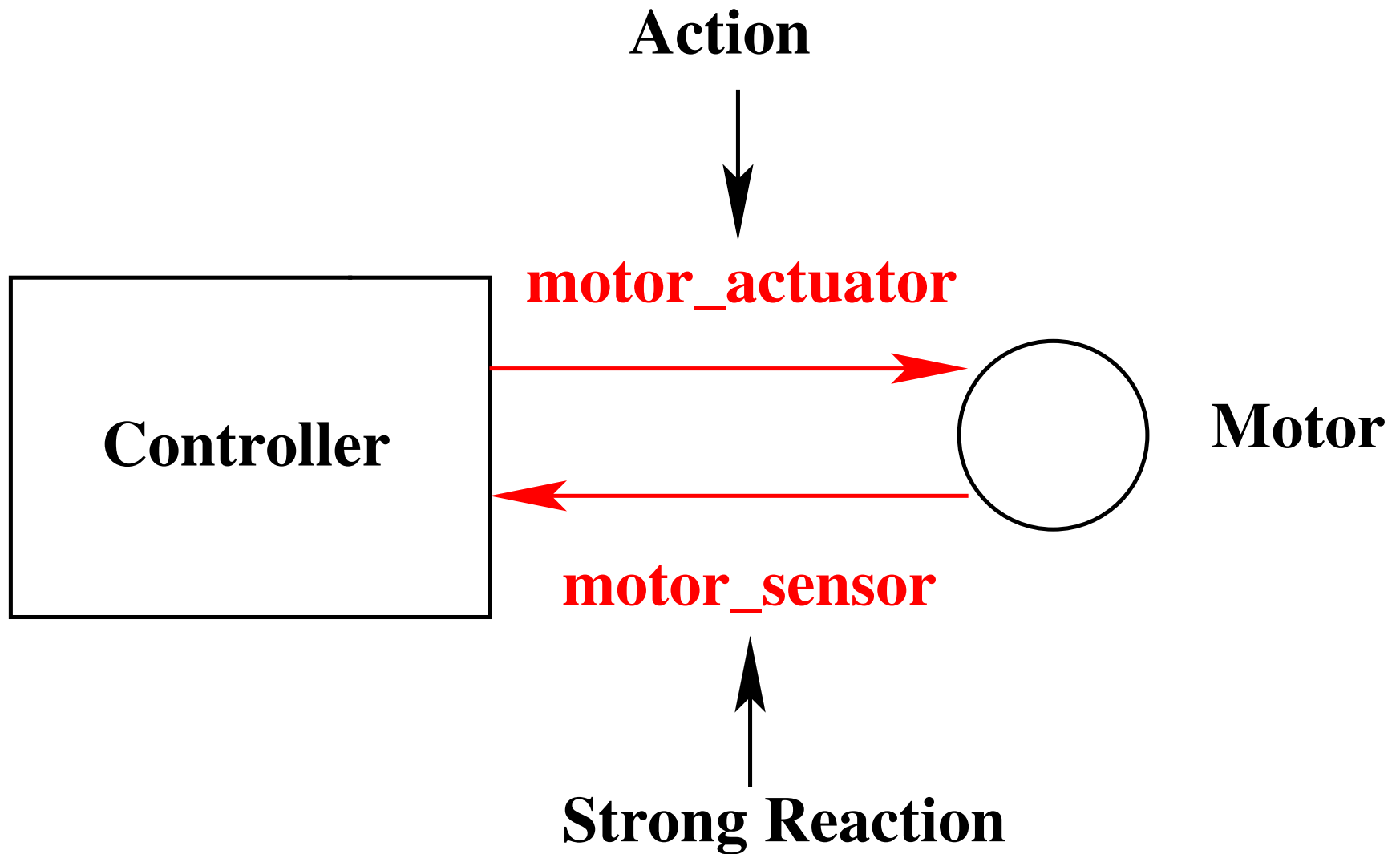
axm0_1: *STATUS = {stopped, working}*

axm0_2: *stopped ≠ working*

variables: *motor_actuator*
 motor_sensor

inv0_1: *motor_sensor* \in *STATUS*

inv0_2: *motor_actuator* \in *STATUS*



- We **instantiate the weak pattern** as follows:

<i>a</i>	\rightsquigarrow	<i>motor_actuator</i>
<i>r</i>	\rightsquigarrow	<i>motor_sensor</i>
<i>0</i>	\rightsquigarrow	<i>stopped</i>
<i>1</i>	\rightsquigarrow	<i>working</i>

a_on	\rightsquigarrow	treat_start_motor
a_off	\rightsquigarrow	treat_stop_motor
r_on	\rightsquigarrow	Motor_start
r_off	\rightsquigarrow	Motor_stop

- Convention: **Controller events** start with "**treat_**"

```
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
```

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

```
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
```

r_off

when

$r = 1$

$a = 0$

then

$r := 0$

end

Motor_stop

when

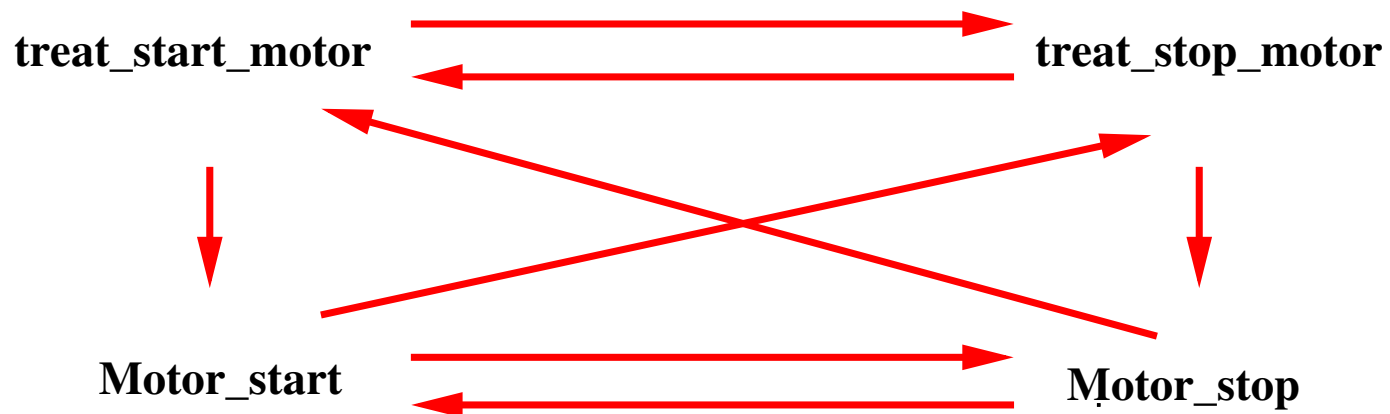
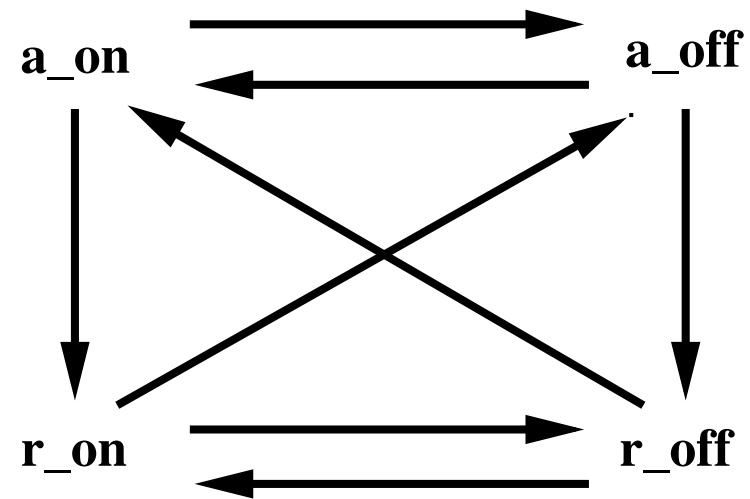
motor_sensor = working

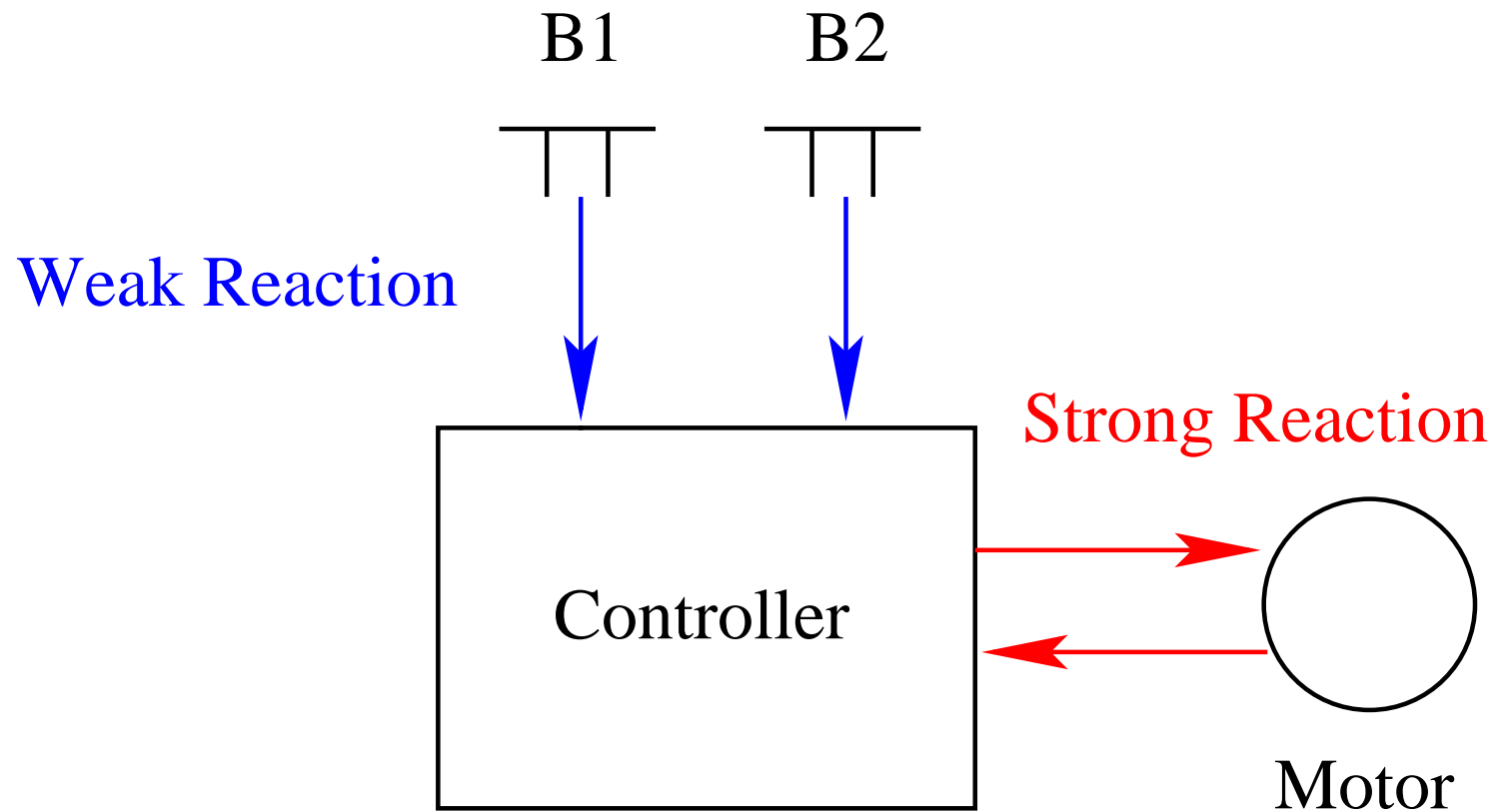
motor_actuator = stopped

then

motor_sensor := stopped

end





Buttons and Controller are weakly synchronized

FUN_1

The counters have
been removed

```
init
  a := 0
  r := 0
```

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

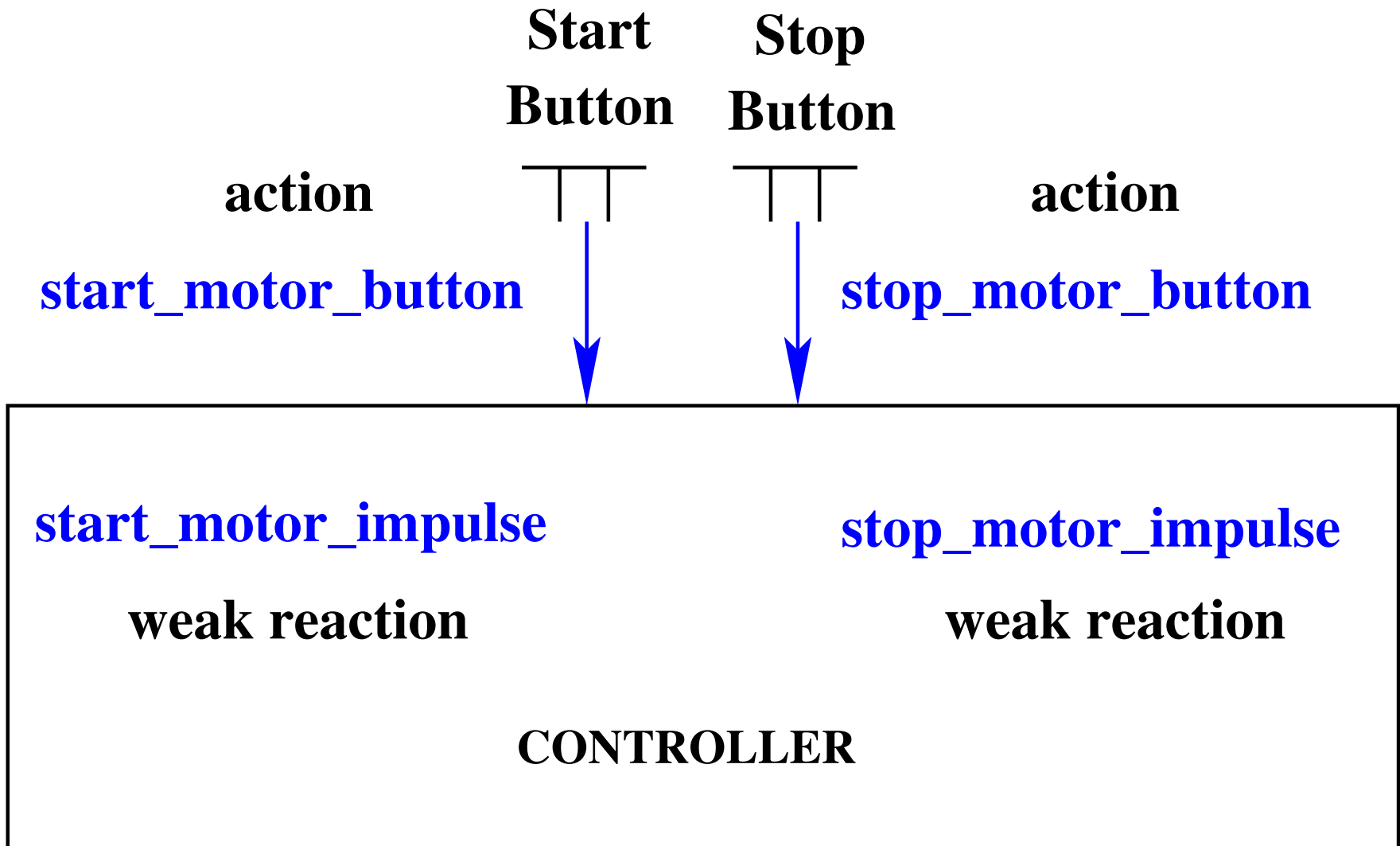
```
a_off
  when
    a = 1
  then
    a := 0
  end
```

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

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

variables: ...
 start_motor_button
 stop_motor_button
 start_motor_impulse
 stop_motor_impulse

inv1_1: *stop_motor_button* ∈ BOOL
inv1_2: *start_motor_button* ∈ BOOL
inv1_3: *stop_motor_impulse* ∈ BOOL
inv1_4: *start_motor_impulse* ∈ BOOL



- We **instantiate the pattern** as follows:

<i>a</i>	\rightsquigarrow	<i>start_motor_button</i>
<i>r</i>	\rightsquigarrow	<i>start_motor_impulse</i>
0	\rightsquigarrow	FALSE
1	\rightsquigarrow	TRUE

a_on	\rightsquigarrow	push_start_motor_button
a_off	\rightsquigarrow	release_stop_motor_button
r_on	\rightsquigarrow	treat_push_start_motor_button
r_off	\rightsquigarrow	treat_release_start_motor_button

- We rename **treat_start_motor** as **treat_push_start_motor_button**

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

```
push_start_motor_button  
  when  
     $start\_motor\_button = FALSE$   
  then  
     $start\_motor\_button := TRUE$   
  end
```

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

```
release_start_motor_button  
  when  
     $start\_motor\_button = TRUE$   
  then  
     $start\_motor\_button := FALSE$   
  end
```

```
r_on
```

```
  when
```

```
     $r = 0$ 
```

```
     $a = 1$ 
```

```
  then
```

```
     $r := 1$ 
```

```
end
```

```
treat_push_start_motor_button
```

```
  refines
```

```
    treat_start_motor
```

```
  when
```

```
     $start\_motor\_impulse = FALSE$ 
```

```
     $start\_motor\_button = TRUE$ 
```

```
     $motor\_actuator = stopped$ 
```

```
     $motor\_sensor = stopped$ 
```

```
  then
```

```
     $start\_motor\_impulse := TRUE$ 
```

```
     $motor\_actuator := working$ 
```

```
end
```

- This is the **most important** slide of the talk
- We can see how **patterns can be superposed**

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

r_on

when

r = 0

a = 1

then

r := 1

end

treat_push_start_motor_button

when

start_motor_impulse = FALSE

start_motor_button = TRUE

motor_actuator = stopped

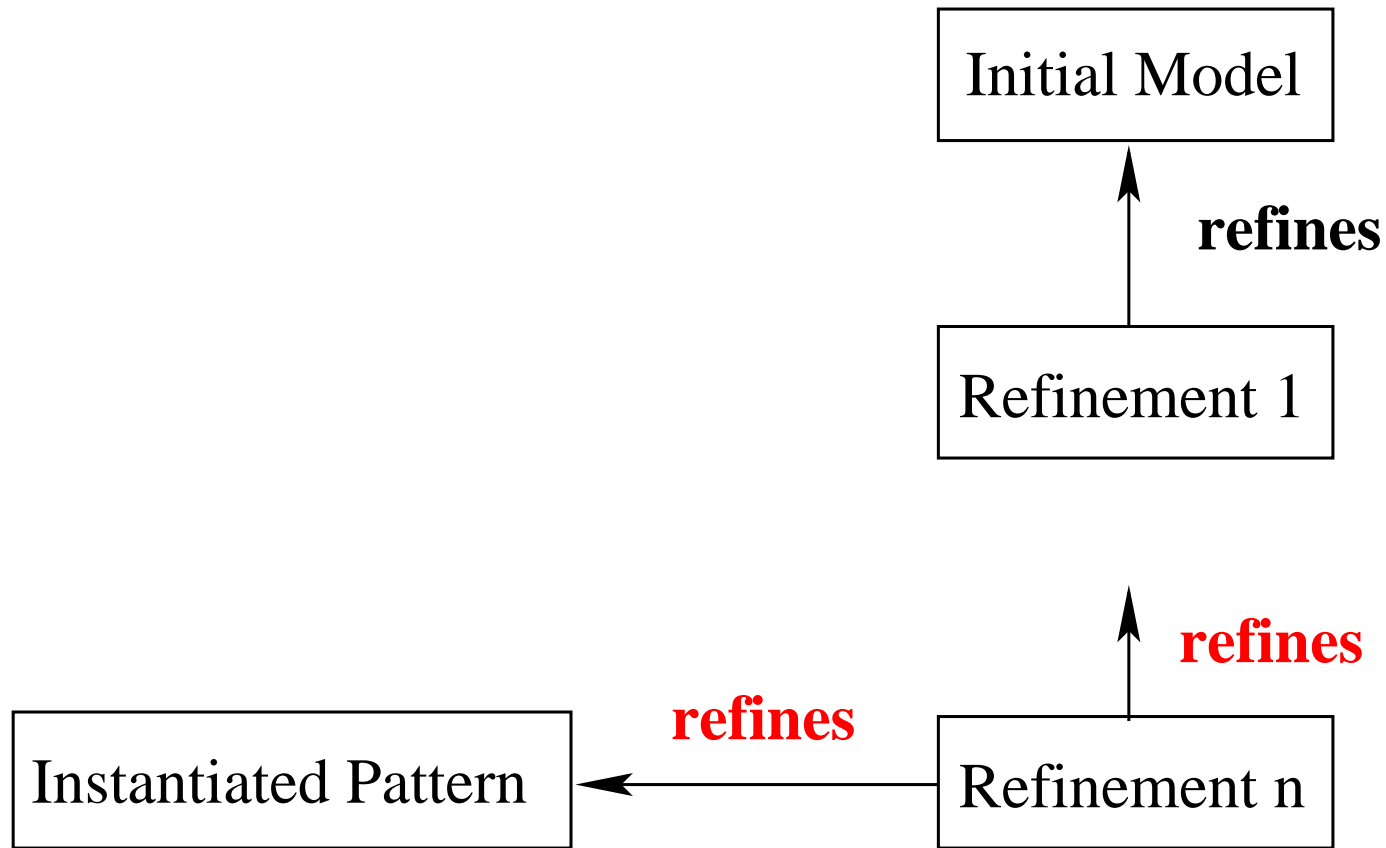
motor_sensor = stopped

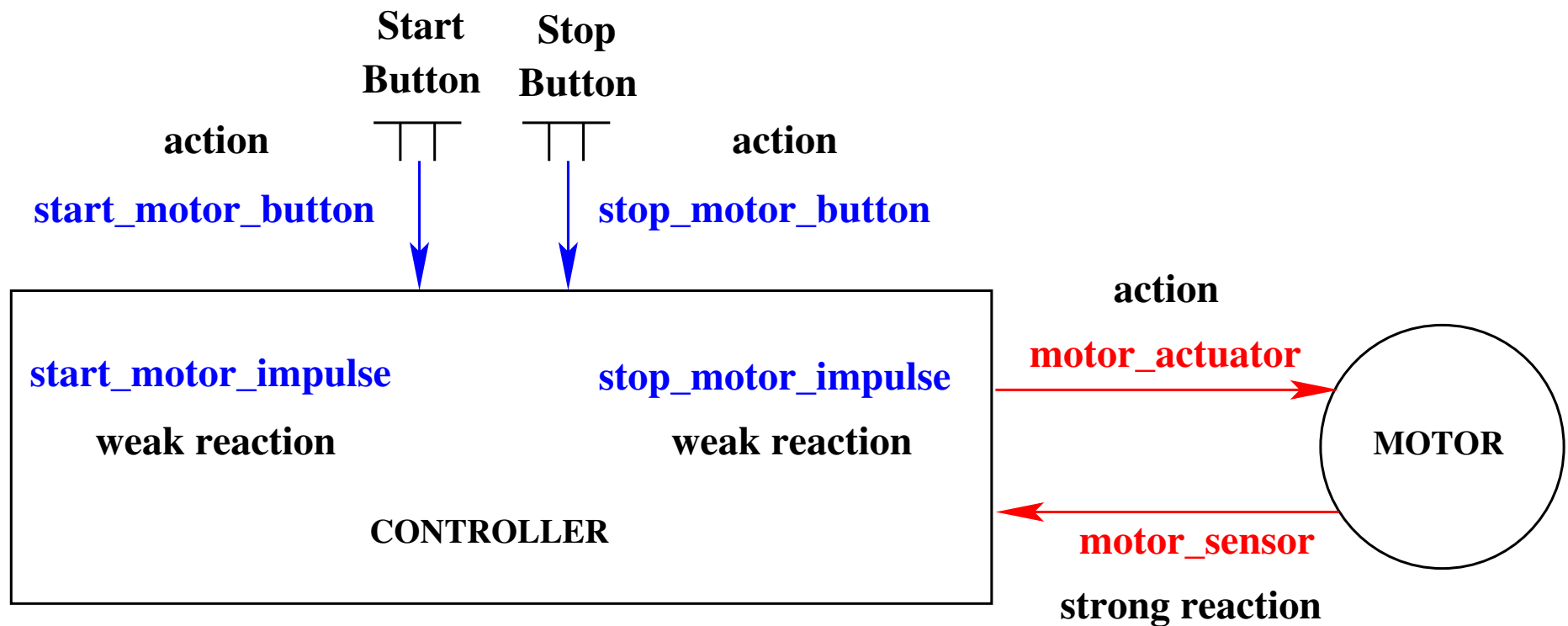
then

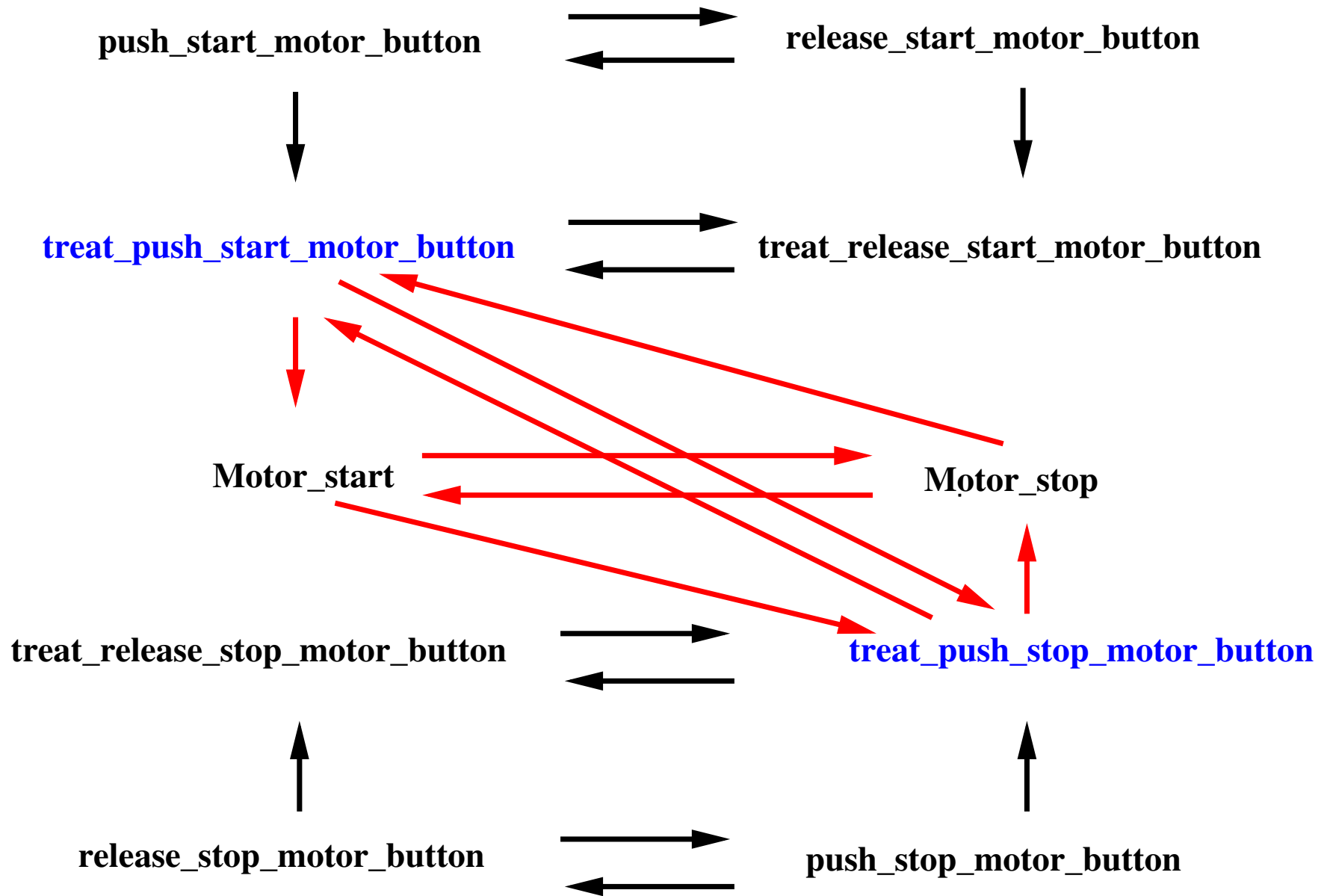
start_motor_impulse := TRUE

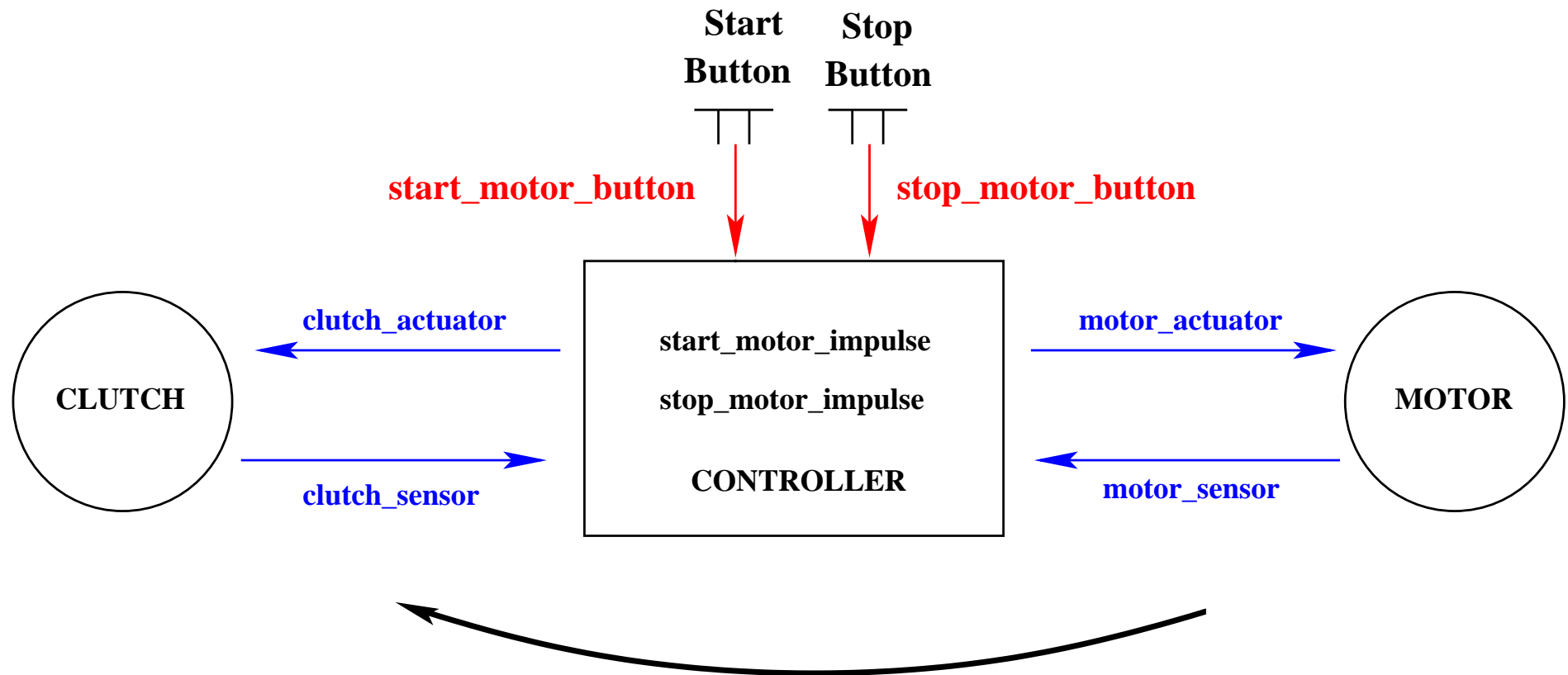
motor_actuator := working

end







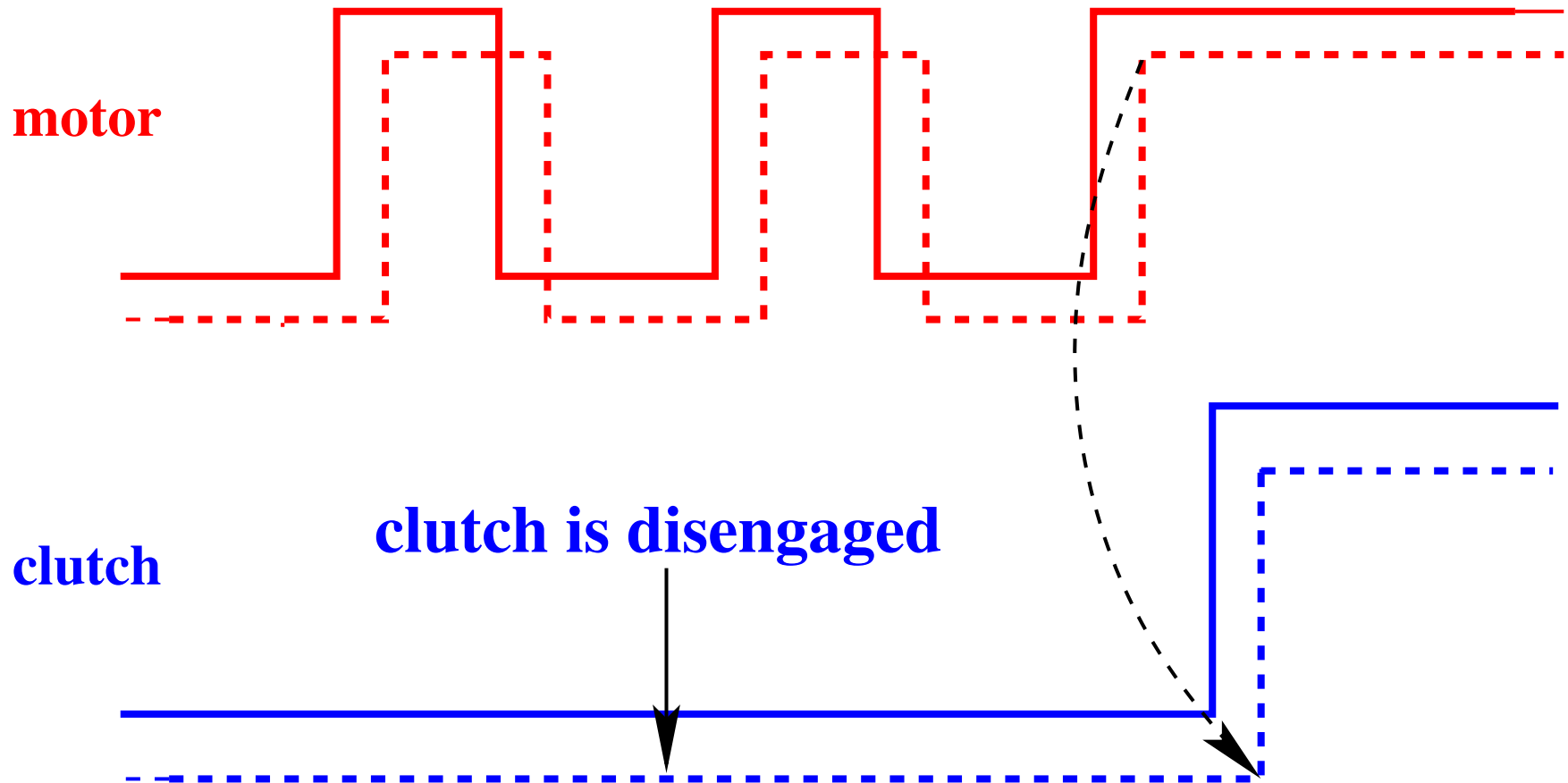


- We copy (after renaming "motor" to "door") what we have done in the initial model

- An additional **safety constraint**

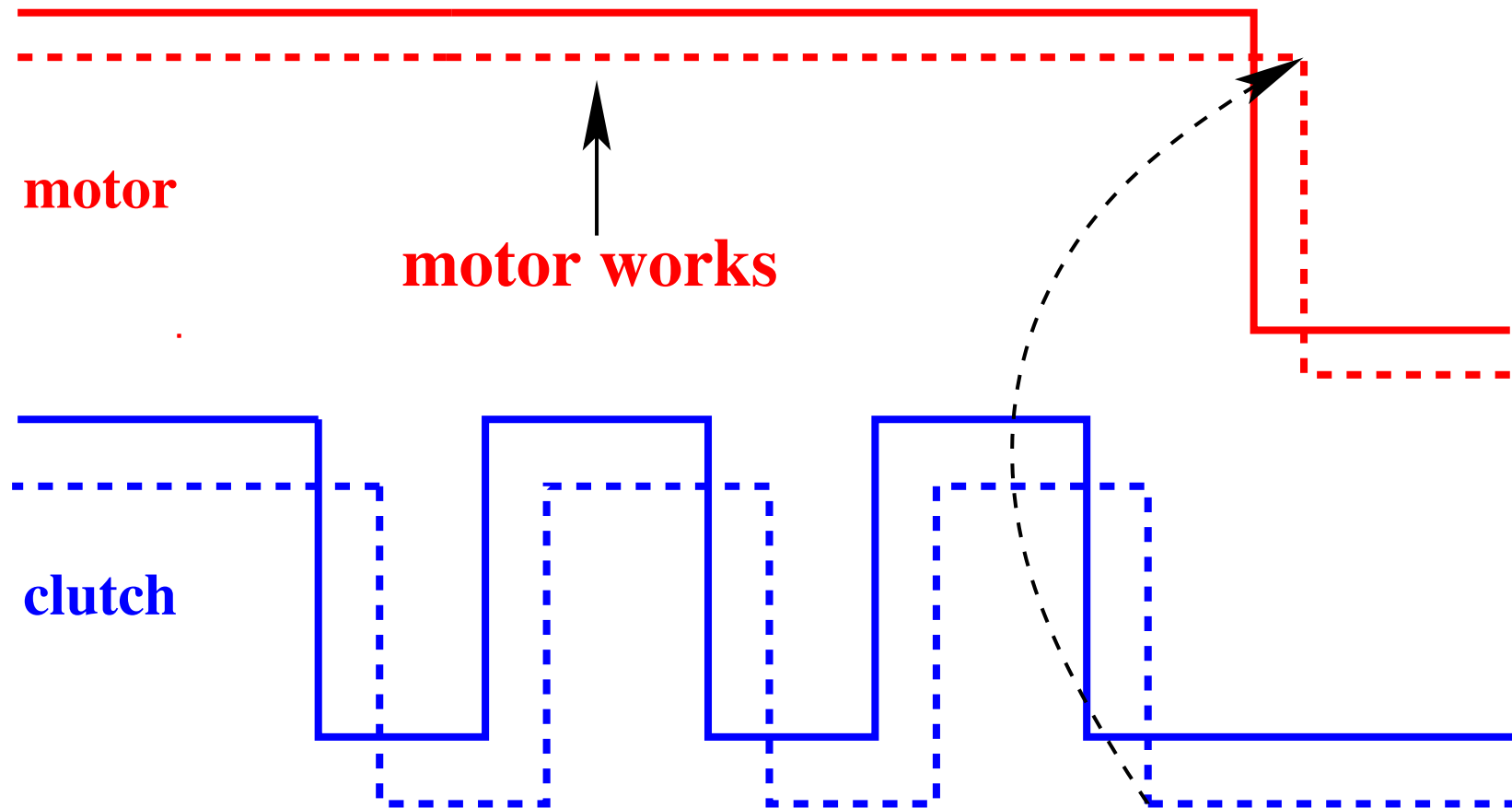
When the clutch is engaged, the motor must work	SAF_1
---	-------

- For this we develop **ANOTHER DESIGN PATTERN**
- It is called: **Weak synchronization of two Strong Reactions**



When the clutch is disengaged,
then

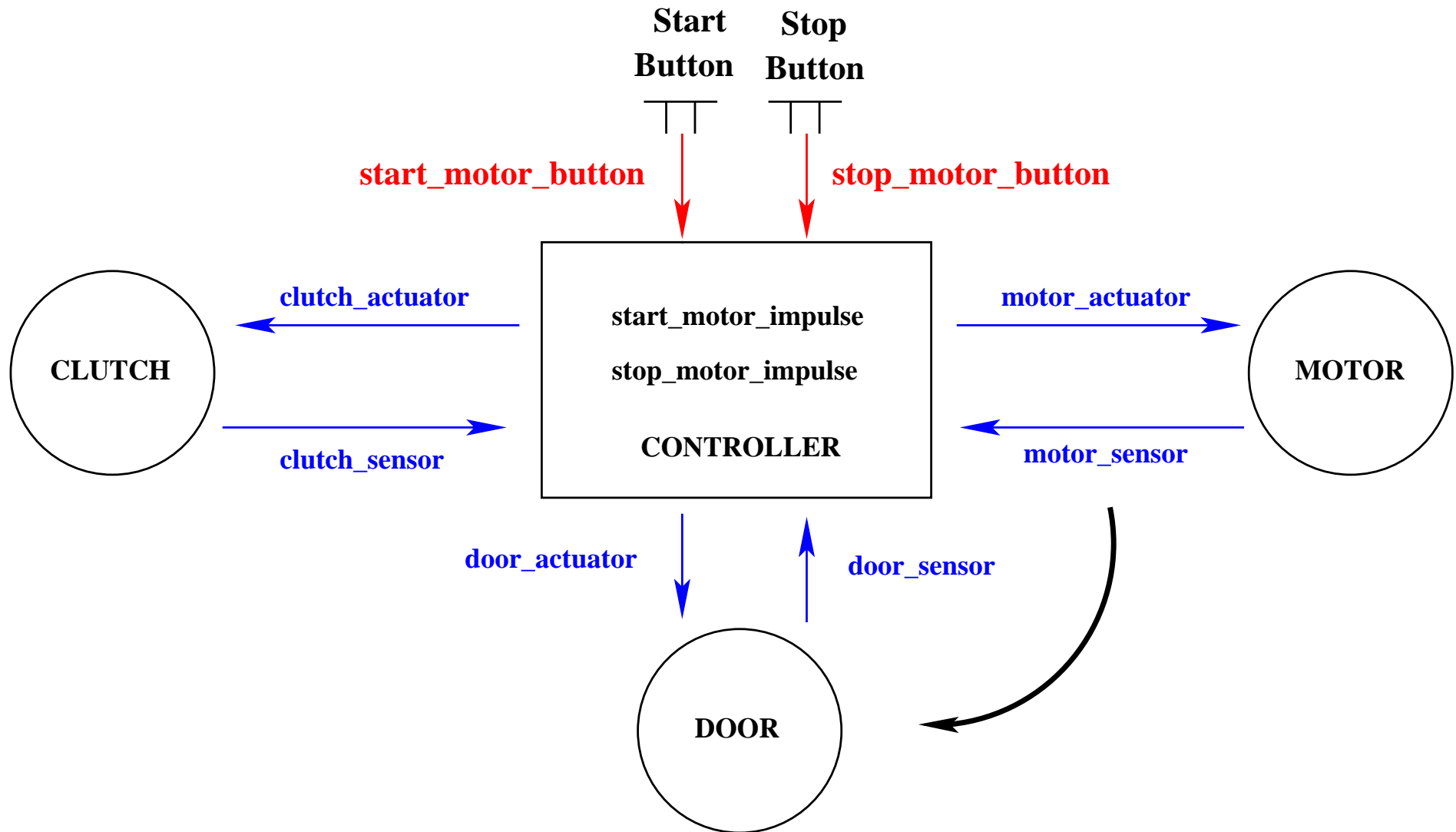
the motor can be started and stopped several times



When the motor works,

then

the clutch can be engaged and disengaged several times



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

- An additional **safety constraint**

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

- We copy (after renaming "motor" to "door") what has been done in the third model:

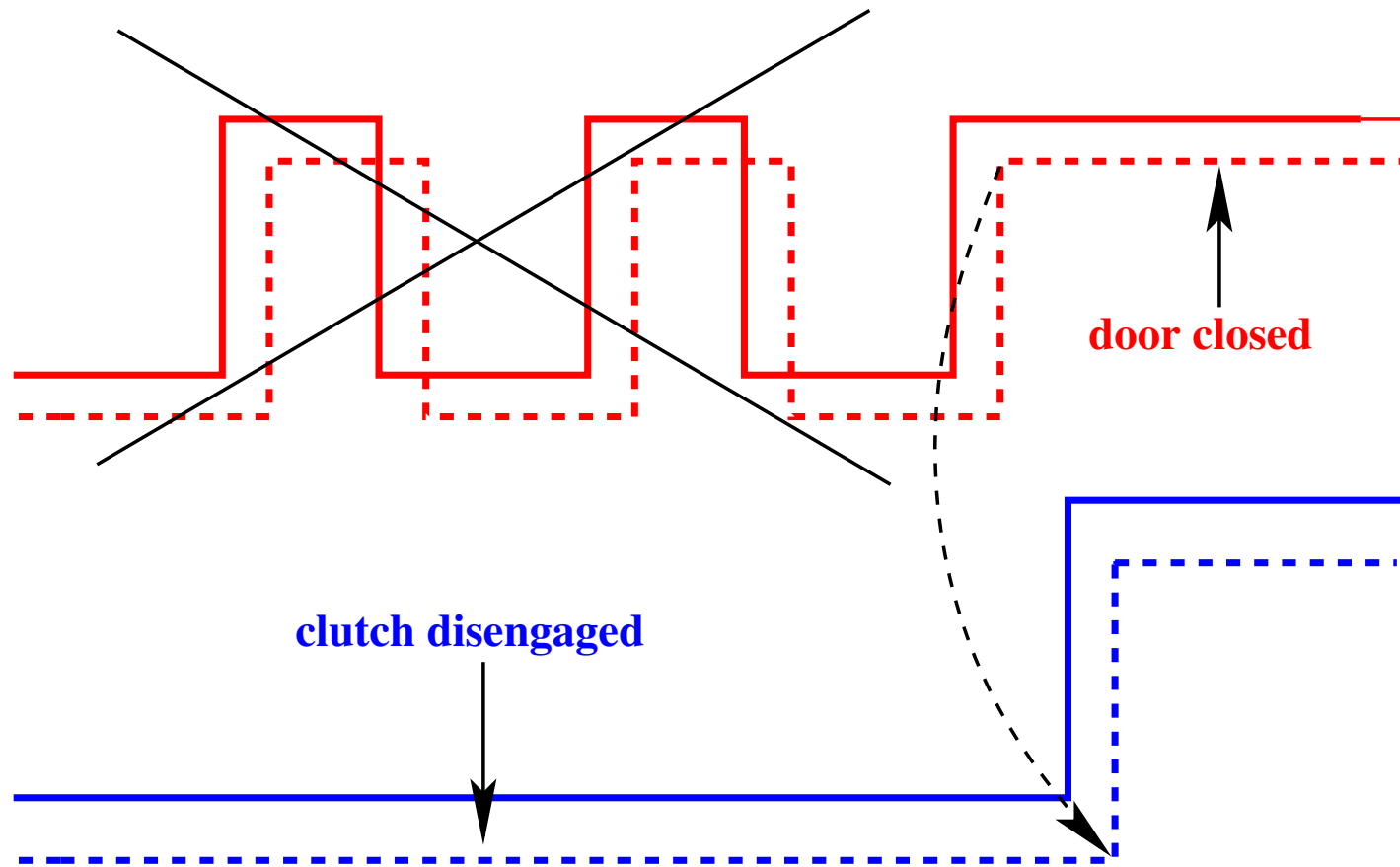
When the clutch is engaged, the motor must work	SAF_1
---	-------

- Adding two **functional constraints**

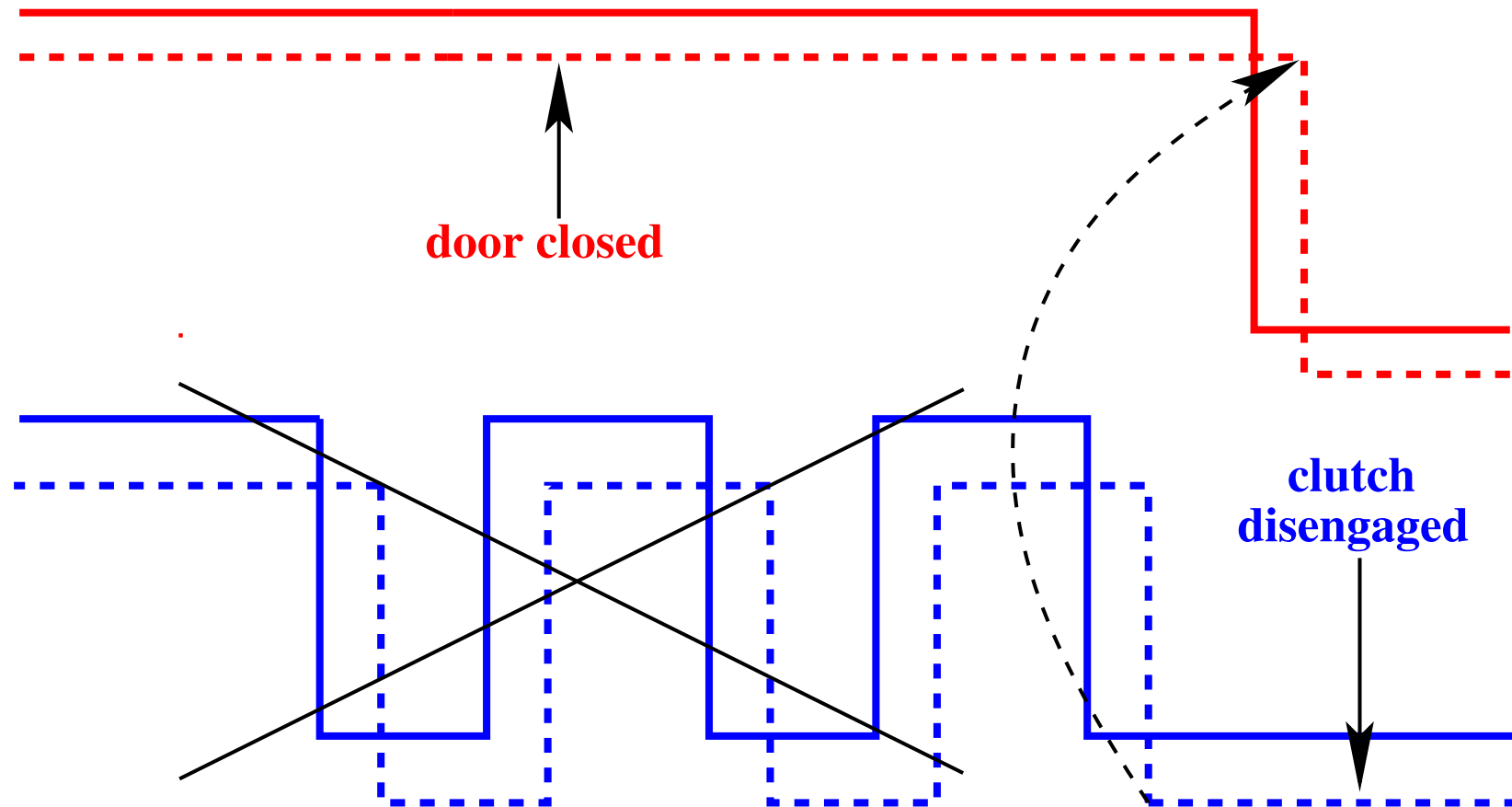
When the clutch is disengaged, the door cannot be closed several times, ONLY ONCE	FUN_3
---	-------

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

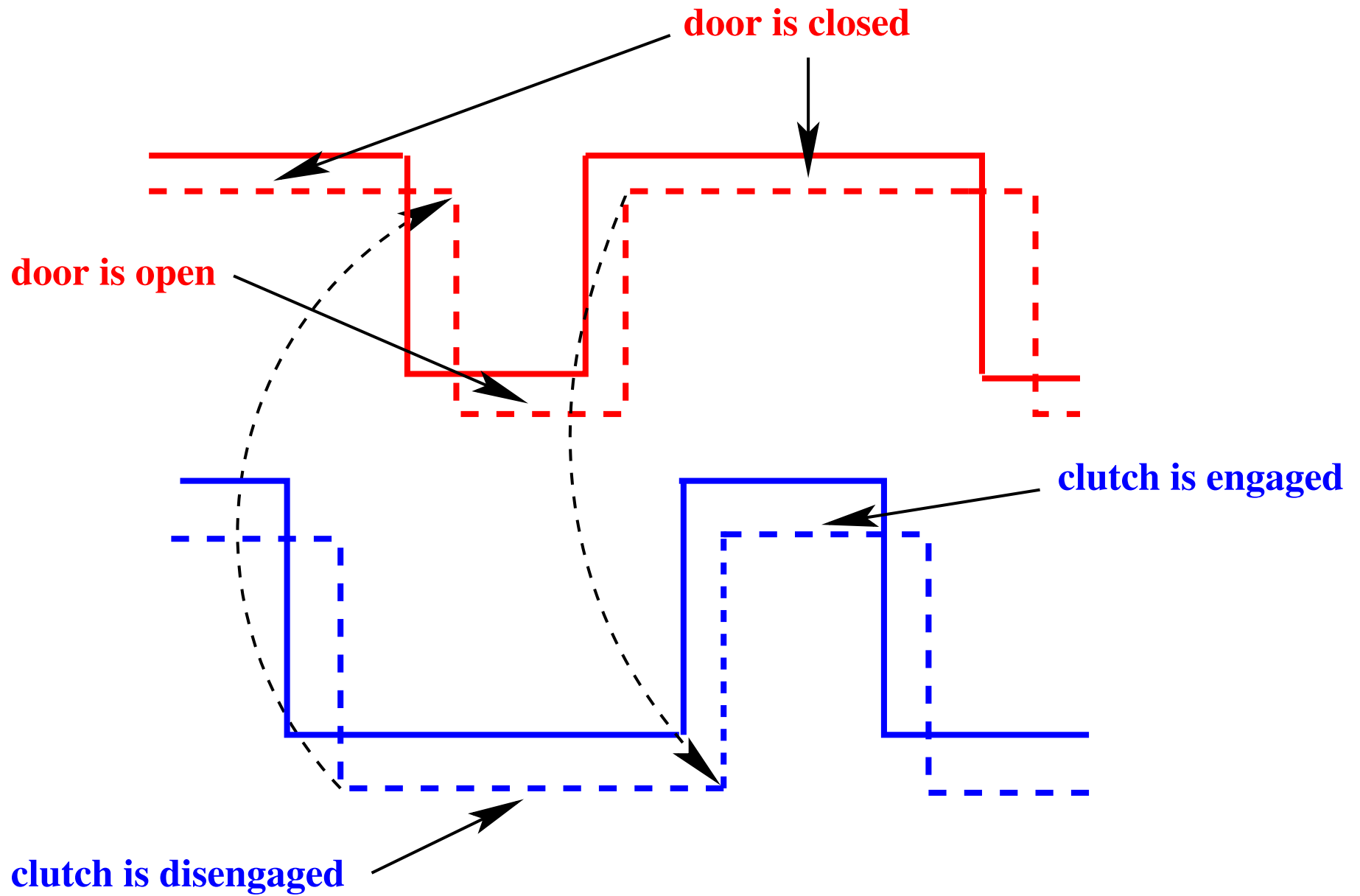
- For this we develop **ANOTHER DESIGN PATTERN**
- It is called: **Strong synchronization of two Strong Reactions**

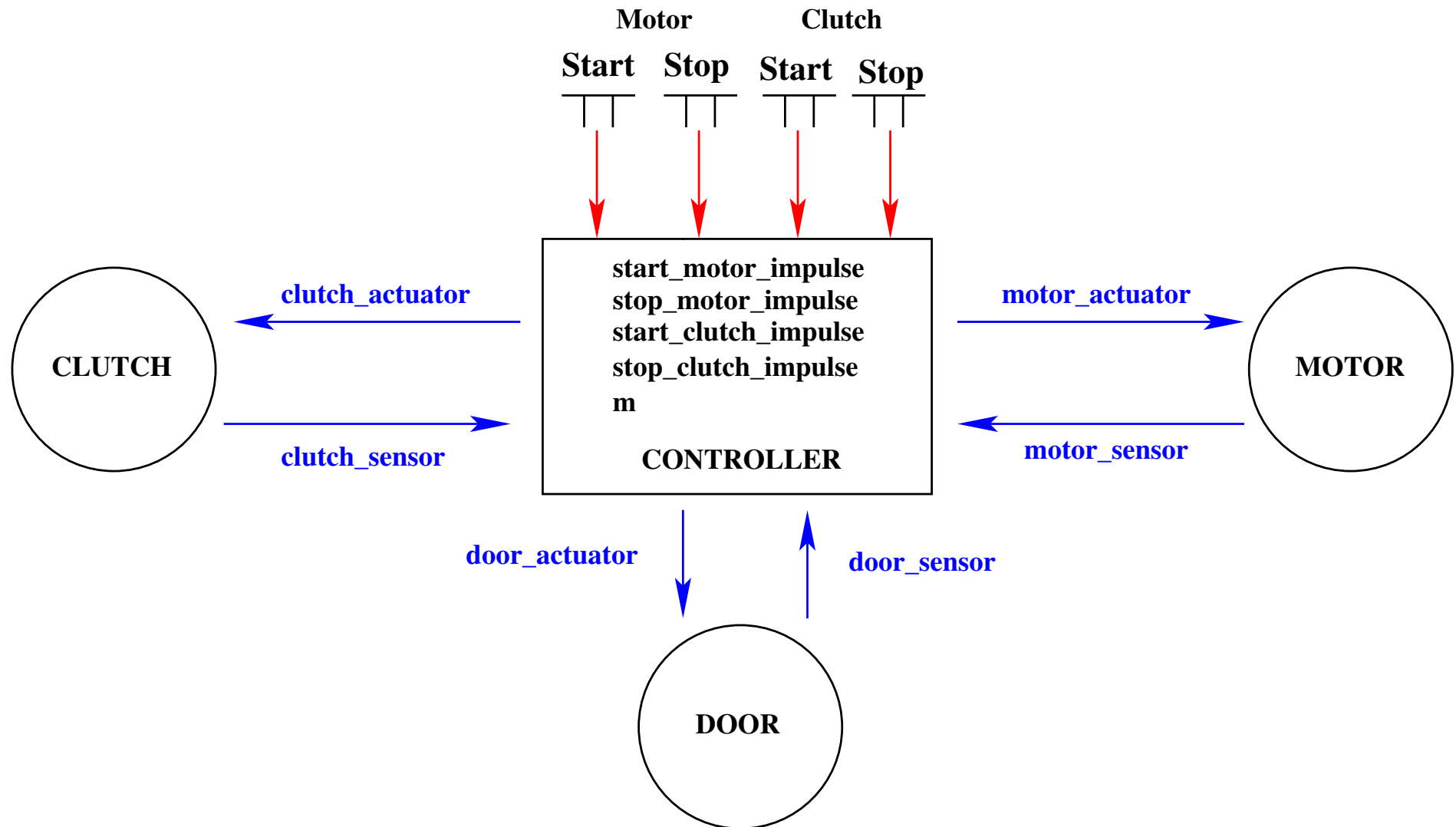


- When the clutch is disengaged, the door cannot be closed several times



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





- 7 sensor variables
- 3 actuator variables
- 5 controller variables
- 14 environment events,
- 14 controller events

- 4 **weak** reactions: 4 buttons (B1, B2, B3, B4)
- 3 **strong** reactions: 3 devices (motor, clutch, door)
- 3 **strong-weak** reactions: motor-clutch, clutch-door, motor-door
- 1 **strong-strong** reaction: clutch-door

- Weak reaction: 6
 - Strong reaction: 3
 - Strong-weak reaction: 16
 - Strong-strong reaction: 7
 - Total: 32
-
- Press (typing): 15
 - Total: 15

- Weak reaction: 18
- Strong reaction: 12
- Strong-weak reaction: 60
- Strong-strong reaction: 40
- Total: 130 (2 interactive)

- Press: 0

- PO saving: $4 \times 18 + 3 \times 12 + 3 \times 60 + 40 = 328$

- 600 lines of C code for the simulation,
- 470 lines come from a direct translation of the last refinement,
- 130 lines correspond to the hand-written interface.

D E M 0-1 (Simulation)

D E M 0-2 (Animation)

- This design pattern approach **seems to be fruitful**
- It results in a **very systematic** formal development
- We develop other patterns for **message handling** in protocols
- **More automation** has to be provided (**plug-in**)

Thanks for Listening

