Methods and Tools for System and Software Construction

1. Introduction

Jean-Raymond Abrial (ETHZ)

August 2008

Purpose of the Course

- To show that software (and systems) can be correct by construction
- Insights about modelling and formal reasoning using Event-B
- To show that this can be made practical with the Rodin Platform
- To illustrate this approach with examples
    - small sequential programs
    - a mechanical press controller
    - a file transfer protocol (time permitting)

What you Will Learn

- By the end of the course you should be "comfortable" with:
    - Modelling (versus programming)
    - Abstraction and refinement
    - Some mathematical techniques used for reasoning
    - The practice of proving as a means to construct programs
    - The usage of the Rodin Platform

Schedule

- 1. Introduction

- 2. Introduction (cont'd) and Rodin Platform

- 3. Mechanical Press

- 4. Mechanical Press (cont'd)

- 5. Transmission Protocol

Distributed Material

- Fours chapters of a book to be published this year by CUP
  - I. Introduction
  - III. A Mechanical Press Controller
  - IV. A Simple File Transfer Protocol
  - V. The Event-B Notation and Proof Obligation Rules

- Slides will be made available through the Summer School web site

Preamble 1: A bit of History

- 1968-2008: 40 years of Software "Engineering"

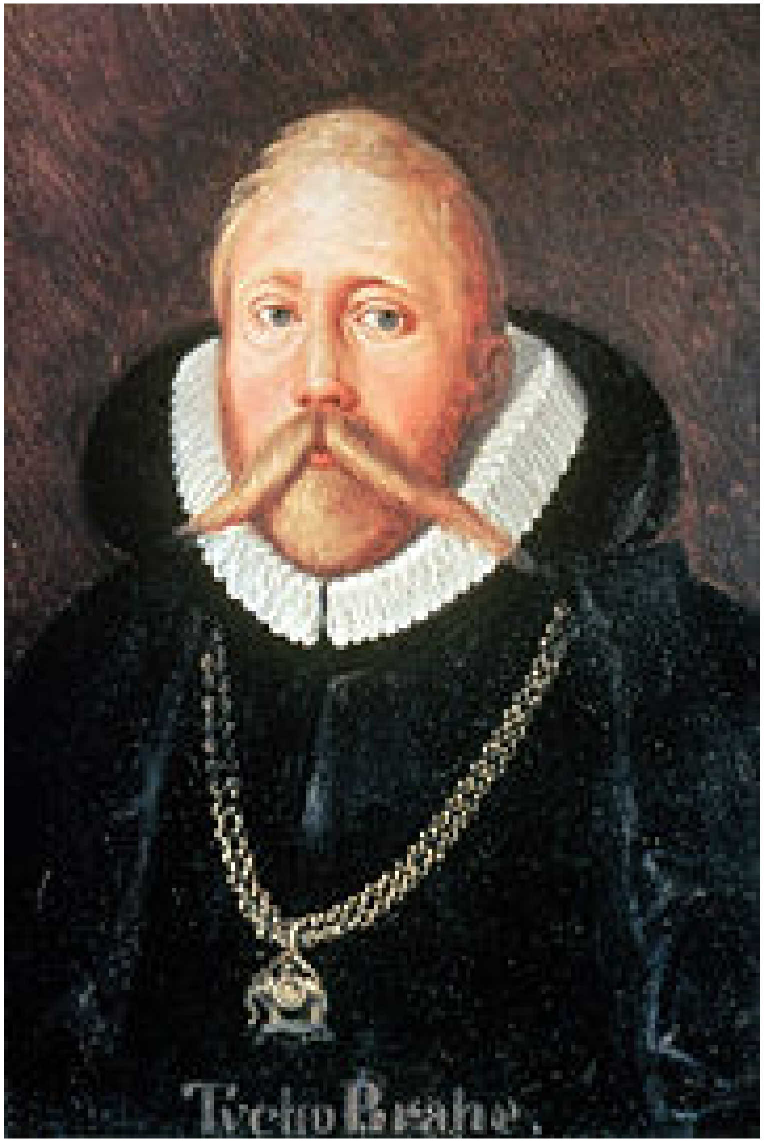
- Spending most of the time writing programs

- Would have been better (maybe) spending time designing programs

- How about generalizing the notion of program

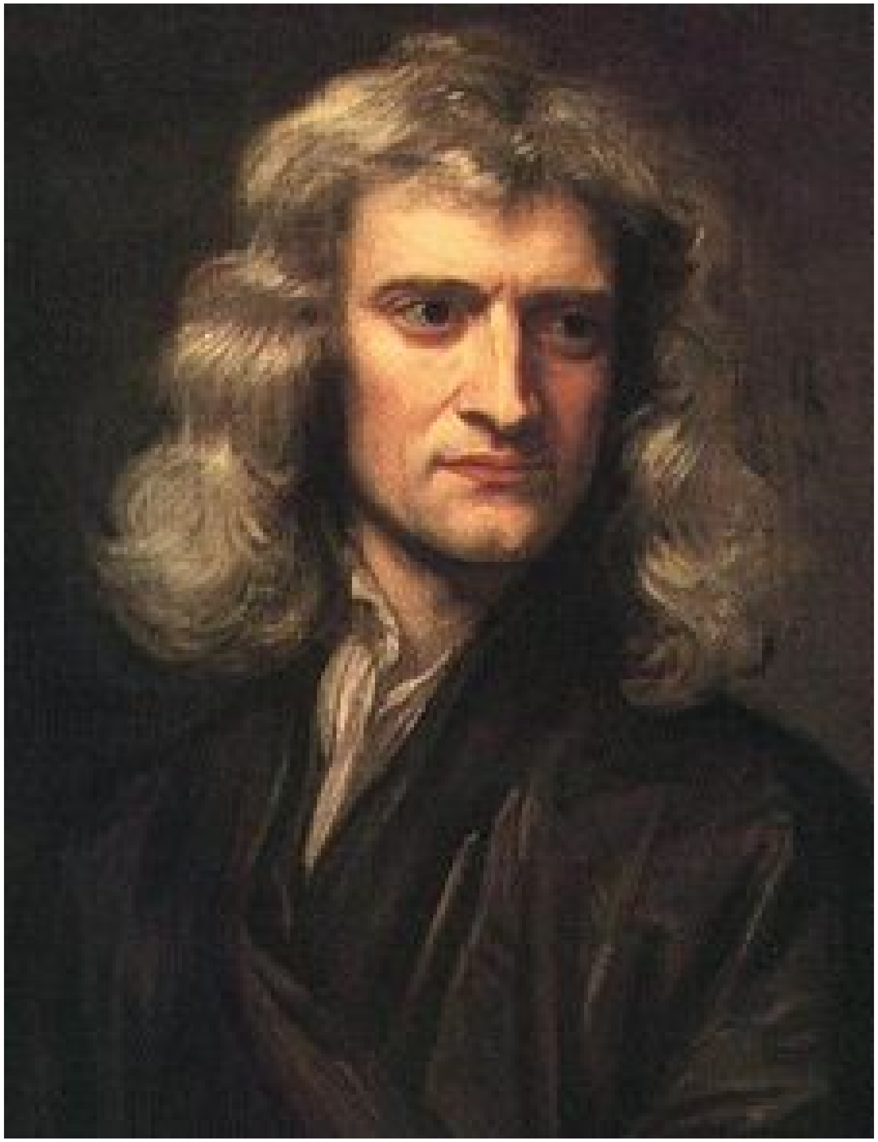
- Comparing these 40 years with other scientific achievements

Measuring: Tycho Brahe (1546-1601)



Observing: Kepler (1571-1630)

Explaining: Newton (1643-1727)



Explaining More: Einstein (1879-1955)

- 400 years vs.40 years: we are still very young!

Preamble 2: The Vasa Disaster

The Story

- August 10, 1628: The Swedish warship Vasa sank.

- This was her maiden voyage.

- She sailed about 1,300 meters only in Stockholm harbor.

- 53 lives were lost in the disaster.

Problems with the Vasa Construction

1. Changing requirements (by King Gustav II Adolf).
2. Lack of specifications (by Ship Builder Henrik Hybertsson).
3. Lack of explicit design (by Subcontractor Johan Isbrandsson)
   (No scientific calculation of the ship stability)
4. Test outcome was not followed (by Admiral Fleming)

References

- Enter keywords "Vasa disaster" in Google

- The Vasa: A Disaster Story with Software Analogies.
  By Linda Rising.
  The Software Practitioner, January-February 2001.
  http://members.cox.net/risingl1/articles/Vasa.pdf

- Why the Vasa Sank: 10 Problems and Some Antidotes
  for Software Projects.
  By Richard E. Fairley and Mary Jane Willshire.
  IEEE Software, March-April 2003.
  http://www.cse.ogi.edu/ dfairley/The_vasa.pdf

Preamble 3: The Ariane 5 Disaster

The Story

- June 4, 1996: The launch vehicle Ariane 5 exploded.

- This was its maiden voyage.

- It flew for about 37 Sec only in Kourou’s sky.

- No injury in the disaster.

The Story (cont’d)

- Normal behavior of the launcher for 36 Sec after lift-off

- Failure of both Inertial Reference Systems almost simultaneously

- Strong pivoting of the nozzles of the boosters and Vulcain engine

- Self-destruction at an altitude of 4000 m (1000 m from the pad)

More Details

- Both inertial computers failed because of overflow on one variable

- This caused a software exception and stops these computers

- These computers sent post-mortem info through the bus

- Normally the main computer receives velocity info through the bus

- The main computer was confused and pivoted the nozzles

More Details (cont’d)

- The faulty program was working correctly on Ariane 4

- The faulty program was not tested for A5 (since it worked for A4)

- But the velocity of Ariane 5 is far greater than that of Ariane 4

- The faulty program happened to be useless for Ariane 5

- It was kept for commonality reasons

References

- Enter keywords "flight 501" in Google

- Ariane 5 flight 501 Inquiry Board Report:
  http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf

- INRIA report challenging the Inquiry Board Report:
  ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-3079.pdf

Topics of these Introductory Lectures

1.1 About formal methods in general

1.2. About modelling

1.3. A light introduction to Event-B (with small examples)

1.1. About formal methods in general

Formal Methods: a Great Confusion

- What are they used for?

- When are they to be used?

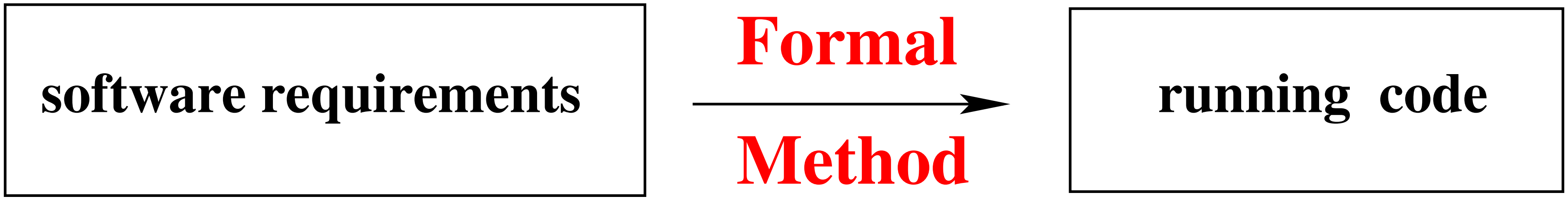
- Is UML a formal method?

- Are they needed when doing OO programming?

- What is their definition?

Purpose of Formal Methods

- Helping people in doing the following transformation:



software requirements

Formal

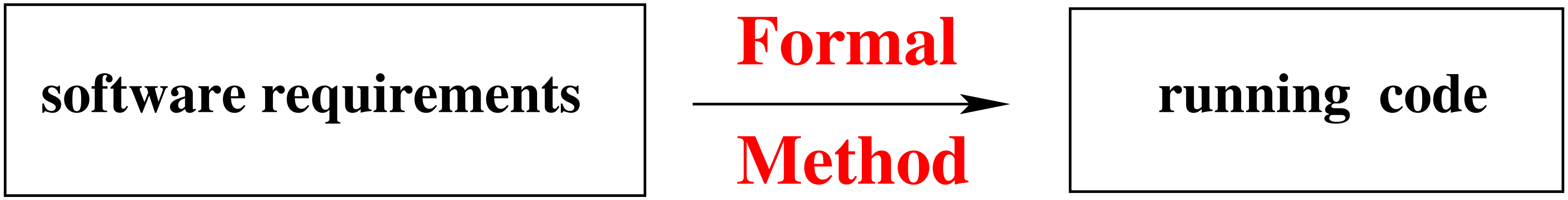
Method

running code

- It does not seem to be different from ordinary programming

Purpose of Formal Methods

- Helping people in doing the following transformation:



software requirements

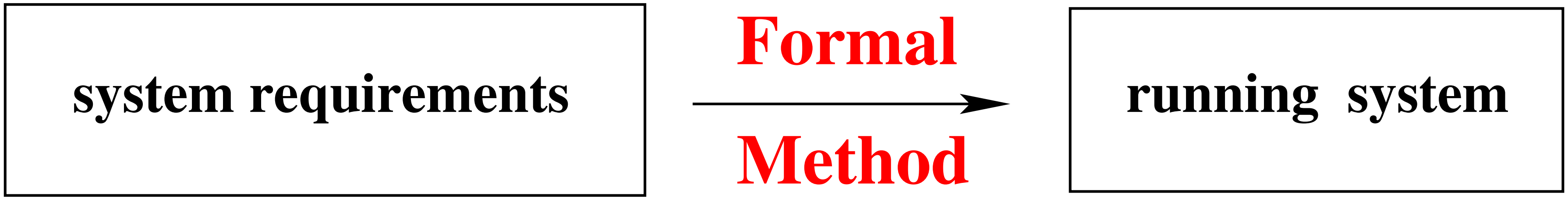
Formal

Method

running code

- It does not seem to be different from ordinary programming

- It can be generalized to:



system requirements

Formal

Method

running system

Preliminary Definitions and Categories

- A formal method is a systematic approach used to determine whether a program has certain wishful properties

- Different kinds of formal methods (according to this definition)
    - Type checking
    - Static analysis
    - Model checking
    - Theorem proving

Comparisons

	Nature	Checked Properties
type checking	programs	internal
model checking	models	external
static ananlysis	programs	external
theorem proving	models	internal

Theorem Proving: Model Construction

- This is the approach developed in these lectures

- It concentrates on the construction of models by successive refinements

- The properties to be proved are parts of the models: invariants and refinement

- At the end of the process, the most refined model is translated into a program

Why Using Formal Methods with Proofs?

- When there is nothing better available.

- When the risk is too high (e.g. in embedded systems).

- When people have already suffered enough.

- When people question their development process.

- Decision of using formal methods is always strategic.

Claimed Difficulties in Using Formal Methods with Proofs

- You have to be a mathematician.

- Formalism is hard to master.

- Not visual enough (no boxes, arrows, etc.).

- People will not be able to do formal proofs.

Genuine Difficulties (my own view)

- You have to think a lot before final coding.

- Incorporation in development process.

- Model building is an elaborate activity.

- Reasoning by means of proof is necessary.

- Poor quality of requirement documents.

What About Other Engineering Disciplines

- Some mature engineering disciplines:
    - Avionics,
    - Civil engineering,
    - Mechanical engineering,
    - Train systems,
    - Ship building.

- Are there any equivalent approaches to Formal Methods with Proofs?

- Yes, BLUE PRINTS

What is a Blue Print?

- A certain representation of the system we want to build

- It is not a mock-up (although mock-ups can be very useful too)

- The basis is lacking (you cannot “drive” the blue print of a car)

- Allows to reason about the future system during its design

- Is it important? (according to professionals) YES

Reasoning about the Intended System?

- Defining and calculating its behavior (what it does)

- Incorporating constraints (what it must not do)

- Defining architecture

- Based on some underlying theories
    - strength of materials,
    - fluid mechanics,
    - gravitation,
    - etc.

Techniques of “Blue Printing”

- Using pre-defined conventions (often computerized these days)

- Conventions should help facilitate reasoning

- Adding details on more accurate versions

- Postponing choices by having some open options

- Decomposing one blue print into several

- Reusing “old” blue prints (with slight changes)

1.2. About modelling

Back to the Questions about Formal Methods

- What are they used for?

- When are they to be used?

- Is UML a formal method?

- Are they needed when doing OO programming?

- What is their definition?

Definitions of Formal Methods Used in this Course

- Formal methods are techniques for building and studying blue prints ADAPTED TO OUR DISCIPLINE

- Our discipline is: design of hardware and software SYSTEMS

- Such blue prints are now called models

- Reminder:
    - Models allow to reason about a FUTURE system
    - The basis is lacking (hence you cannot “execute” a model)

Conventions for Model Writing and Reasoning

- Reminder (cont’d):
    - Using pre-defined conventions
    - Conventions should help facilitate reasoning (more to come)

- Consequence: Using ordinary discrete mathematical conventions:
    - Classical Logic (Predicate Calculus)
    - Basic Set Theory (sets, relations and functions)

Systems we are Interested to Develop

- a “classical” piece of software
- an electronic circuit
- a file transfer protocol
- an airline booking system
- a PC operating system
- a nuclear plant controller
- a Smart-Card electronic purse
- a launch vehicle flight controller
- a driverless train controller
- a mechanical press controller
- etc.

Characterizing such Systems (general)

- They are made of many parts

- They interact with a possibly hostile environment

- They involve several executing agents

- They require a high degree of correctness

- There construction spreads over several years

- Their specifications are subjected to many changes

Characterizing such Systems (more technical)

- These systems operate in a discrete fashion

- Their dynamical behavior can be abstracted by:
    - A succession of steady states
    - Intermixed with sudden jumps

- The possible number of state changes are enormous

- Usually such systems never halt

- They are called DISCRETE TRANSITION SYSTEMS

Reasoning about Discrete Transition Systems

- Test reasoning (a vast majority): VERIFICATION

- Blue Print reasoning (a very few): CORRECT CONSTRUCTION

Test Reasoning: VERIFICATION

- Based on laboratory execution
- Obvious incompleteness
- The oracle is usually missing
- Properties to be checked are chosen a posteriori
- Re-adapting and re-shaping after testing
- Reveals an immature technology

Blue Print Reasoning: CORRECT CONSTRUCTION

- Based on a formal model: the “blue print”

- Gradually describing the system with the needed precision

- Relevant Properties are chosen a priori

- Serious thinking made on the model, not on the final system

- Reasoning is validated by proofs

- Reveals a mature technology

Blue Print Reasoning: Outcome of Proving

- The proof succeeds

- The proof fails but refutes the statement to prove
    - the model is erroneous: it has to be modified

- The proof fails but is probably provable
    - the model is badly structured: it has to be reorganized

- The proof fails and is probably not provable nor refutable
    - the model is too poor: it has to be enriched

About Formal Proofs in Industry (Some Figures)

- Rules of Thumb:

n lines of final code implies $n/3$ proofs

95% of proofs discharged automatically

5% of proofs discharged interactively

350 interactive proofs per man-month

- 60,000 lines of final code $\leadsto$ 20,000 proofs $\leadsto$ 1,000 int. proofs
- 1,000 interactive proofs $\leadsto$ 1000/350 $\simeq$ 3 man-months
- Far less expensive than heavy testing

1.3. A Light Introduction to Event-B

(with Small Examples)

Topics

1.3.1. Introduction

1.3.2. First Example

1.3.3. Second Example

1.3.1. Introduction

Model Developments with 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

Model Developments with Event-B (cont'd)

- 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.
- In this lecture, we shall construct small sequential programs.

Main Influences

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)

The State of a Model

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

The Events of a Model (Transitions)

- A discrete model is also made of a number of events

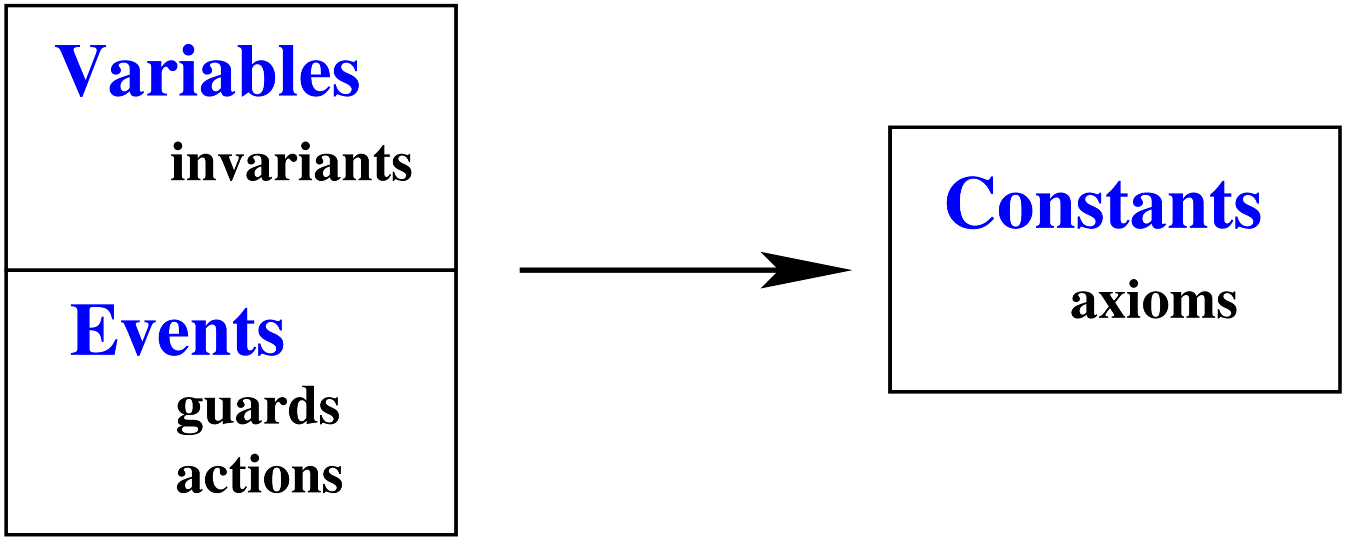
- An event is made of a guard and an action

- 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

A Model Schematic View



Variables

invariants

Events

guards

actions

Constants

axioms

Dynamic Parts

(Machines)

Static Parts

(Contexts)

Operational Interpretation

- An event execution is supposed to take no time

- Thus, no two events can occur simultaneously

- When all events have false guards, the discrete system stops

- When some events have true guards, one of them is chosen non-deterministically and its action modifies the state

- The previous phase is repeated (if possible)

Operational Interpretation

```
Initialize;
while (some events have true guards) {
    Choose one such event;
    Modify the state accordingly;
}
```

Comments on the Operational Interpretation

- Stopping is not necessary: a discrete system may run for ever

- This interpretation is just given here for informal understanding

- The meaning of such a discrete system will be given by the proofs which can be performed on it.

Being more Precise: Machines and Contexts

- A model is made of several components

- A component is either a machine or a context:

Machine

- variables
- invariants
- theorems
- events

Context

- carrier sets
- constants
- axioms
- theorems

Machines and Contexts (cont'd)

- Contexts contain the static structure of a discrete system (constants and axioms)

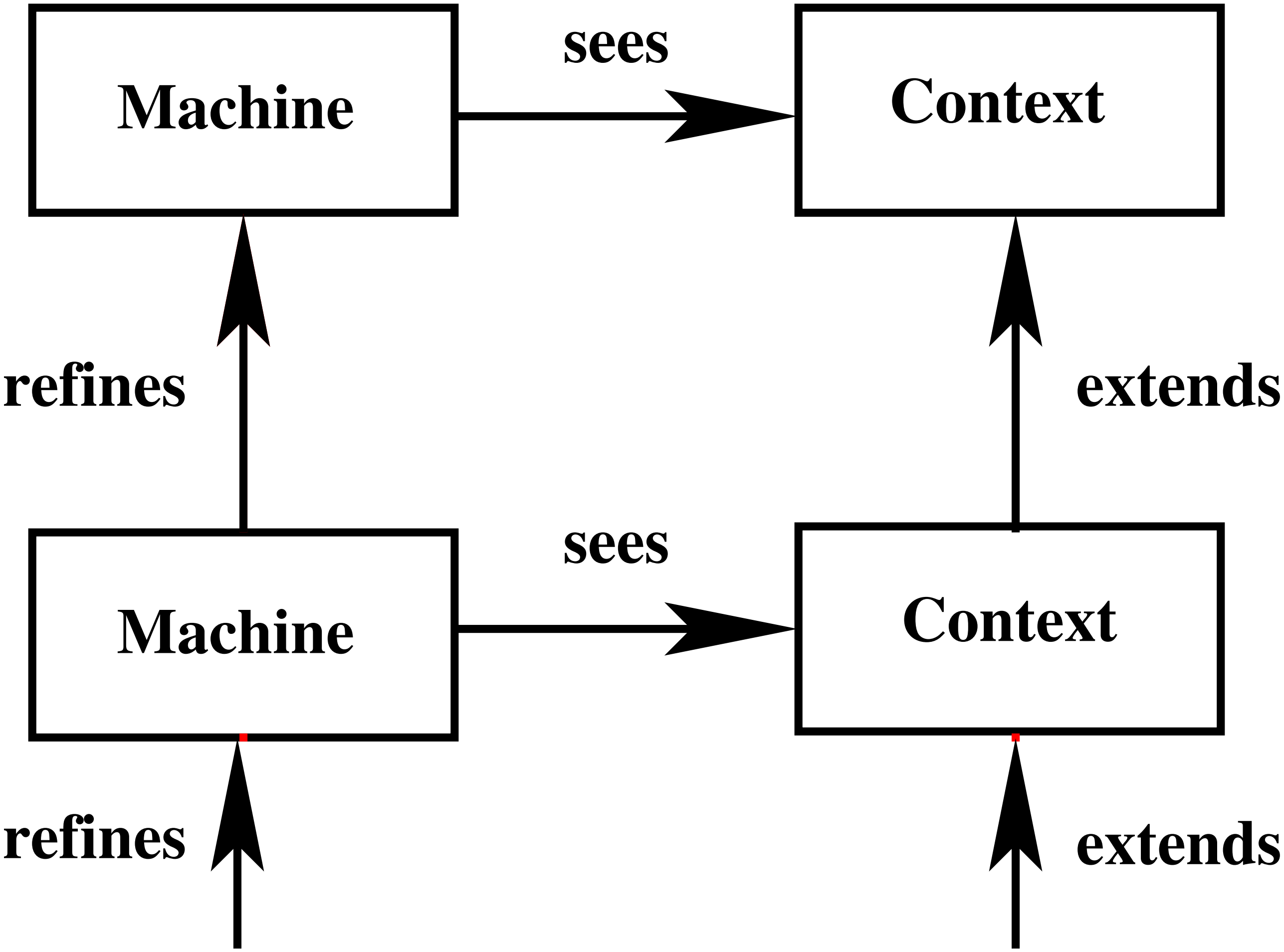
- Machines contain the dynamic structure of a discrete system (variables, invariants, and events)

- Machines see contexts

- Contexts can be extended

- Machines can be refined

Relationship Between Machines and Contexts



1.3.2. A First Simple Example

Requirements for a First Simple Example

We are given a non-empty finite array of natural numbers	FUN-1

We like to find the maximum of the range of this array	FUN-2

Requirements for a First Small Example

We are given a non-empty finite array of natural numbers	FUN-1

We like to find the maximum of the range of this array	FUN-2

We want to find that 10 is the greatest element of this array

9	**3**	**10**	**8**	**3**	**5**

Development Strategy

- First, we show an initial model specifying the problem

- Later, we refine our model to produce an algorithm.

- In the initial model, we have:
  - a context where the constant array is defined
  - a machine where the maximum is "computed"

Initial Model: the Context as presented in the slides

- Constant n denotes the size of the non-empty array,

- Constant f denotes the array,

- Constant M denotes a natural number.

constants: n
f
M

$$0 < n$$

$$f \in 1\,..\,n \longrightarrow 0\,..\,M$$

$$\mathrm{ran}(f) \neq \varnothing$$

- Mind the inference typing

Initial Model: the Context as presented in the slides

- Constant n denotes the size of the non-empty array,

- Constant f denotes the array,

- Constant M denotes a natural number.

constants: n
f
M

axm0_1: $0 < n$

axm0_2: $f \in 1\,..\,n \longrightarrow 0\,..\,M$

thm0_1: $\mathrm{ran}(f) \neq \varnothing$

- Mind the inference typing

The Context as presented in the tool

Context

sets

constants

axioms

theorems

Notice that we have no set

context
  maxi_ctx_0
constants
  n
  f
  M
axioms

  axm1 : $0 < n$

  axm2 : $f \in 1..n \rightarrow 0\,..\,M$

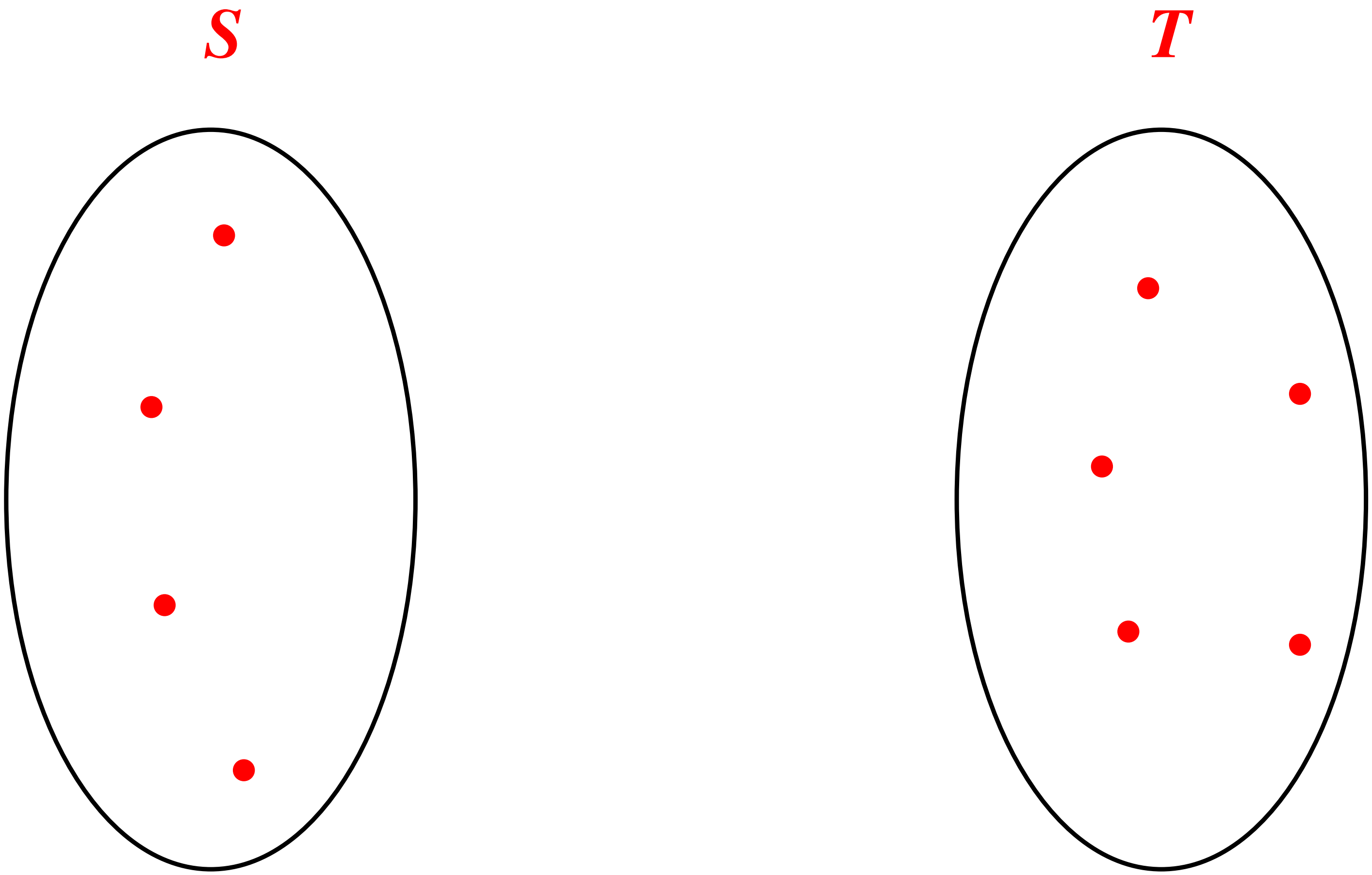
theorems

  thm1 : $\mathrm{ran}(f) \neq \varnothing$

end

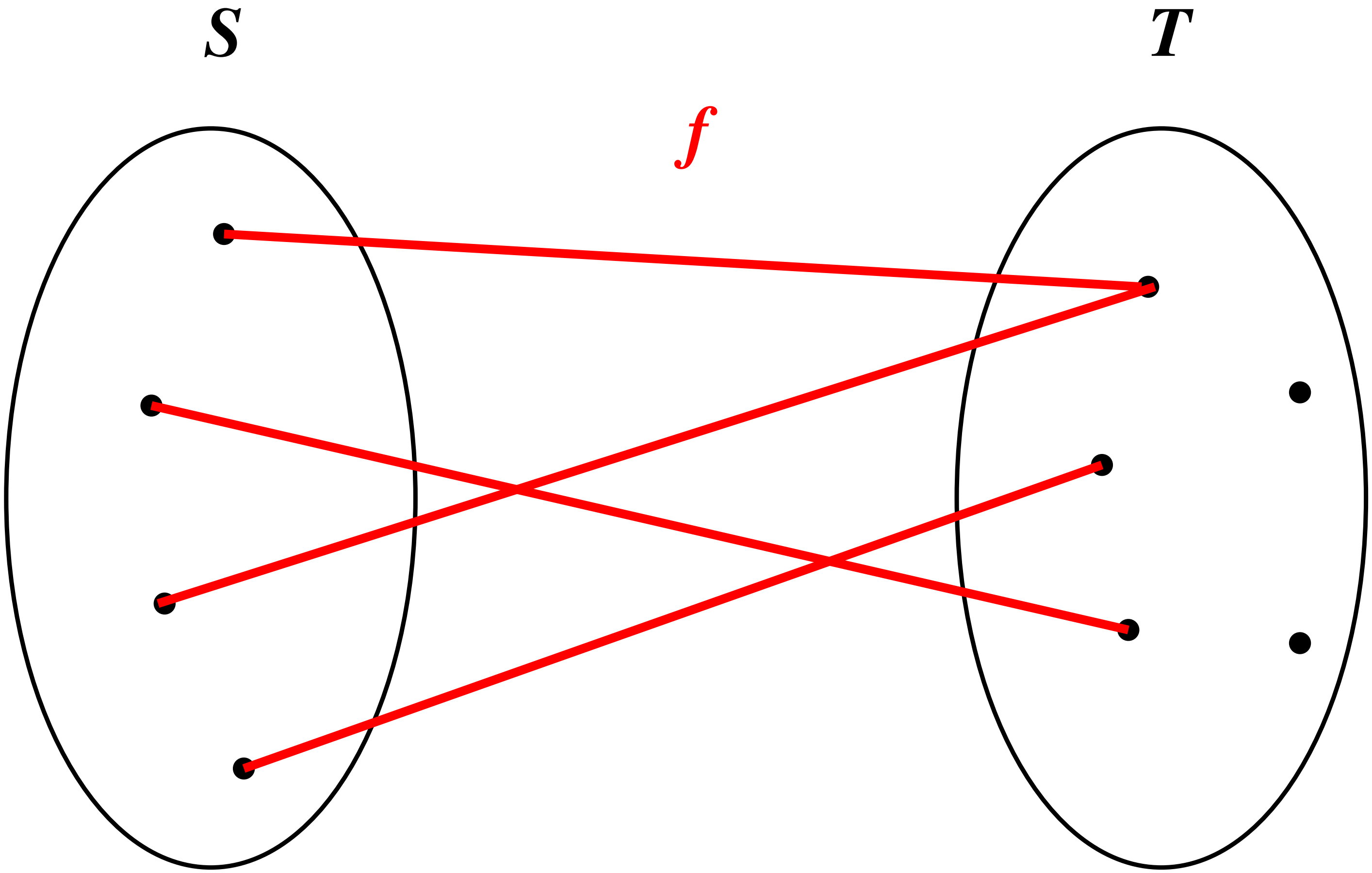
Math Refresher: Total Function and Range

- We are given two sets S and T



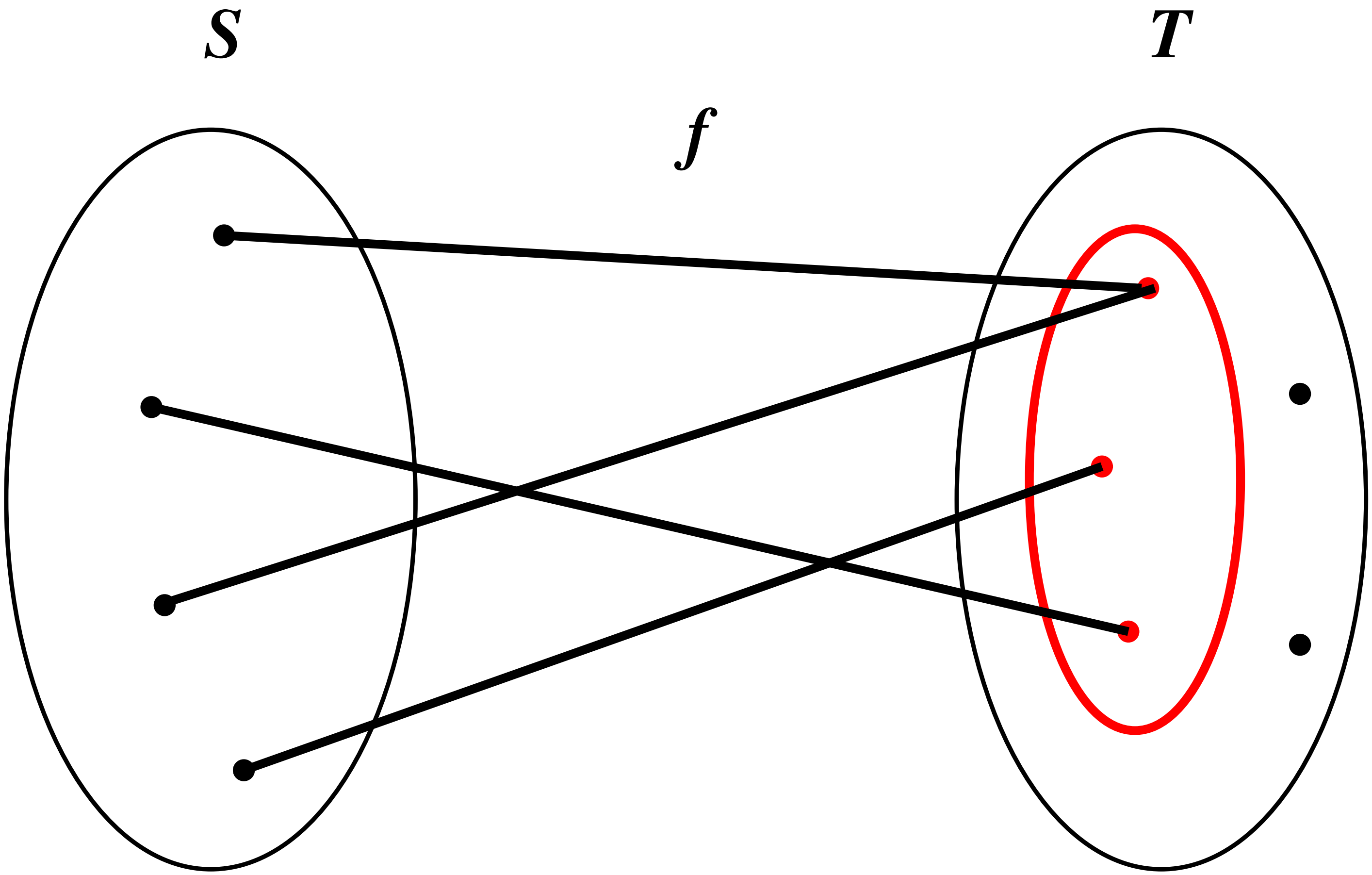
Math Refresher: Total Function and Range

- Here is a total function $\boldsymbol{f}$ from $\boldsymbol{S}$ to $\boldsymbol{T}$: $\boldsymbol{f} \in \boldsymbol{S} \rightarrow \boldsymbol{T}$



Math Refresher: Total Function and Range

- Here is the range of f



DEMO (showing a context)

Context Structure

context
$< context\_identifier >$
sets
$< set\_identifier >$
$\ldots$
constants
$< constant\_identifier >$
$\ldots$
axioms
$< label >$: $< predicate >$
$\ldots$
theorems
$< label >$: $< predicate >$
$\ldots$
end

Explaining Context Sections

- "**sets**" lists various sets, which define pairwise disjoint types

- "**constants**" lists the different constants introduced in the context

- "**axioms**" defines the properties of the constants

- "**theorems**" denotes properties to be proved from the axioms

Initial Model: the Machine as presented in the slides

- Variable m denotes the result.

variable: m

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

- Next are the two events:

```
INIT
  begin
    $m := 0$
  end
```

```
maximum
  begin
    $m := \max(\mathrm{ran}(f))$
  end
```

- Event maximum presents the final intended result (in one shot)

Machine Example as presented in the tool

Machine

variables
invariants
theorems
events

machine
maxi_0
sees
ctx_0
variables
i
invariants

inv1 : $i \in 1\,..\,n$

events
$\ldots$
end

Machine (and Context) Example as presented in the tool

machine
  maxi_0
sees
  maxi_ctx_0
variables
  m
invariants

  inv1 : $m \in \mathbb{N}$

events
  $\ldots$
end

context
  maxi_ctx_0
sets
  D
constants
  n
  f
  v
axioms

  axm1 : $0 < n$

  axm2 : $f \in 1..n \longrightarrow 0\,..\,M$

theorems

  thm1 : $\mathrm{ran}(f) \neq \varnothing$

end

DEMO (showing a machine)

Machine Structure

machine
$< machine\_identifier >$
sees
$< context\_identifier >$
$\ldots$
variables
$< variable\_identifier >$
$\ldots$
invariants
$< label >: \; < predicate >$
$\ldots$
theorems
$< label >: \; < predicate >$
$\ldots$
events
$\ldots$
end

Explaining Machine Sections

- "**variables**" lists the state variables of the machine

- "**invariants**" states the properties of the variables

- "**theorems**" are provable from invariants and seen axioms and thms

- "**events**" defines the dynamics of the transition system (next slides)

Event

- An event defines a transition of our discrete system

- An event is made of a Guard G and an Action A

- G defines the enabling conditions of the transition

- A defines a parallel assignment of the variables

Kind of Events

```
begin
    $A$
end
```

No guard

```
when
    $G$
then
    $A$
end
```

Simple guard

```
any $x$ where
    $G(x)$
then
    $A(x)$
end
```

Quantified guard

Kind of Events

```
begin
    $A$
end
```

No guard

```
when
    $G$
then
    $A$
end
```

Simple guard

```
any $x$ where
    $G(x)$
then
    $A(x)$
end
```

Quantified guard

Our event (so far) have no guards

```
INIT
    begin
        $m := 0$
    end
```

```
maximum
    begin
        $m := \max(\mathrm{ran}(f))$
    end
```

Summary

constants: n
f
M

axm0_1: $0 < n$

axm0_2: $f \in 1\,..\,n \rightarrow 0\,..\,M$

thm0_1: $\mathrm{ran}(f) \neq \varnothing$

variable: m

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

```
INIT
    begin
        $m := 0$
    end
```

```
maximum
    begin
        $m := \max(\mathrm{ran}(f))$
    end
```

Our Task is not Completed

- We have to perform some proofs:
    - **thm0_1** holds
    - Invariant **inv0_1** is established by event "INIT"
    - Invariant **inv0_1** is maintained by event "maximum"
    - Expression "$\mathbf{max}(\mathbf{ran}(f))$" is well-defined

Summary of what is to be Proved

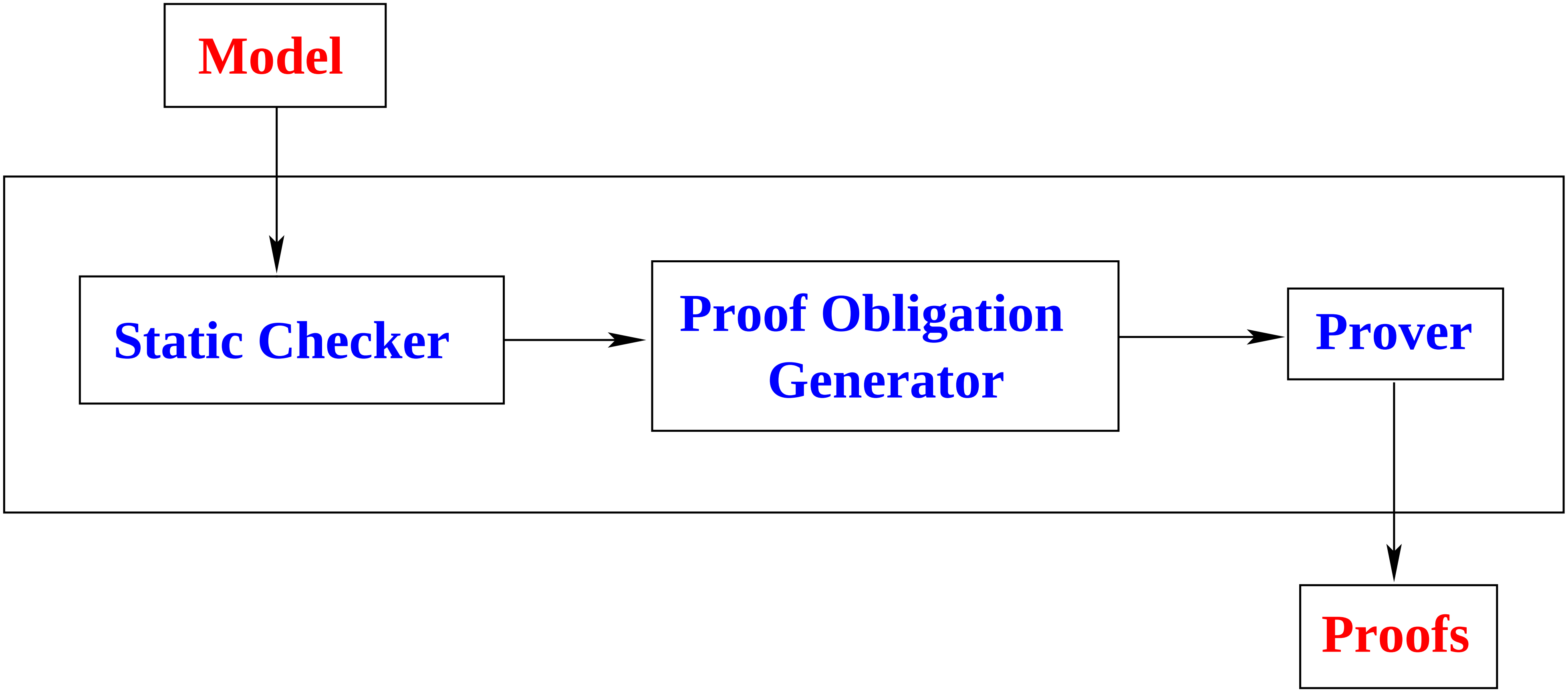
- Stated theorems

- Invariant maintenance

- Well-definedness

DEMO (showing proof obligations)

The Rodin Platform Kernel Tools



Model

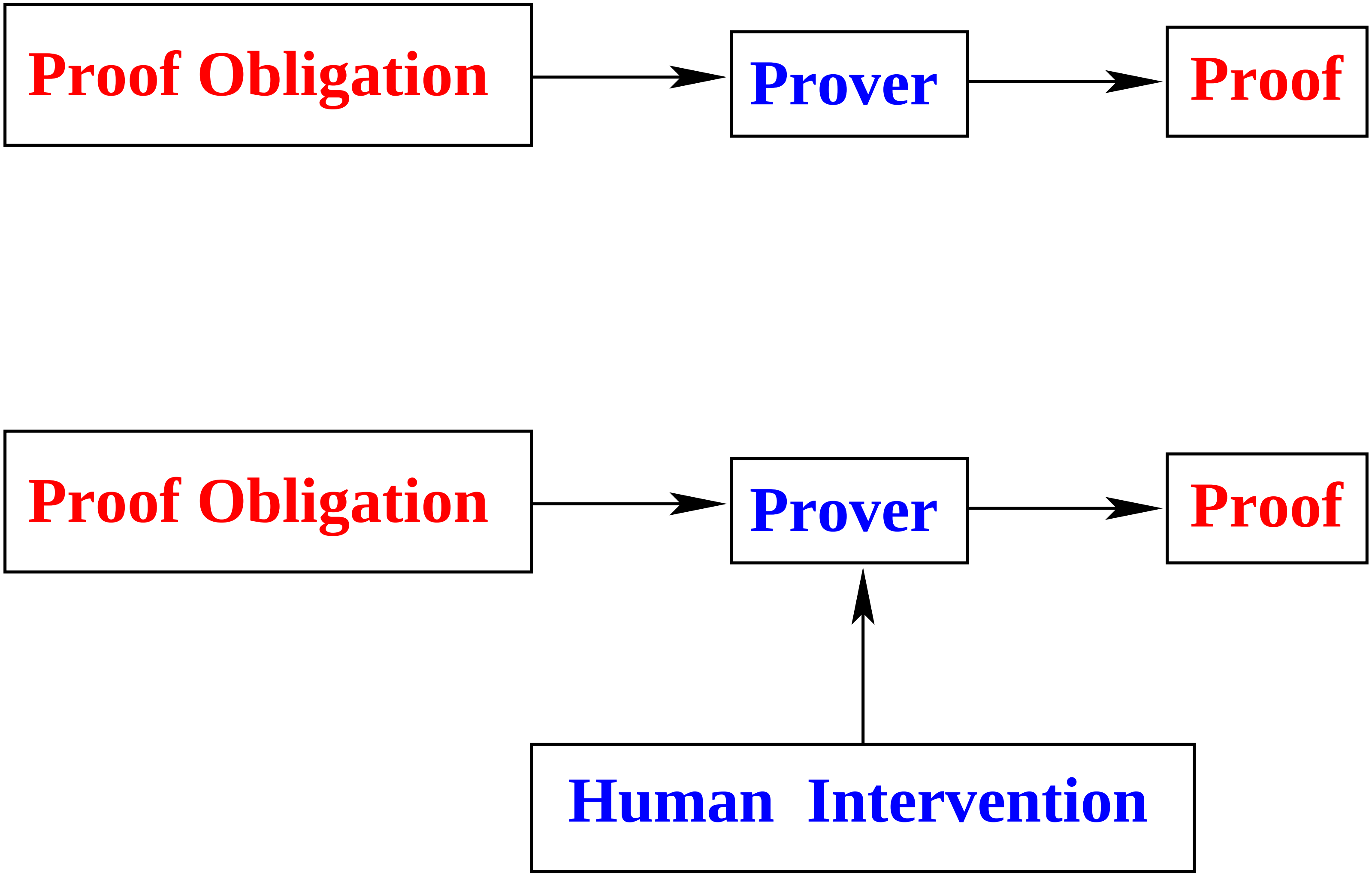
Static Checker

Proof Obligation Generator

Prover

Proofs

Automatic and Interactive Modes for the Prover



Proof Obligation

Prover

Proof

Proof Obligation

Prover

Proof

Human Intervention

Refinement: the New Variables

- We introduce two new variables in our model

- Variables p and q denote two indices in the domain of f.

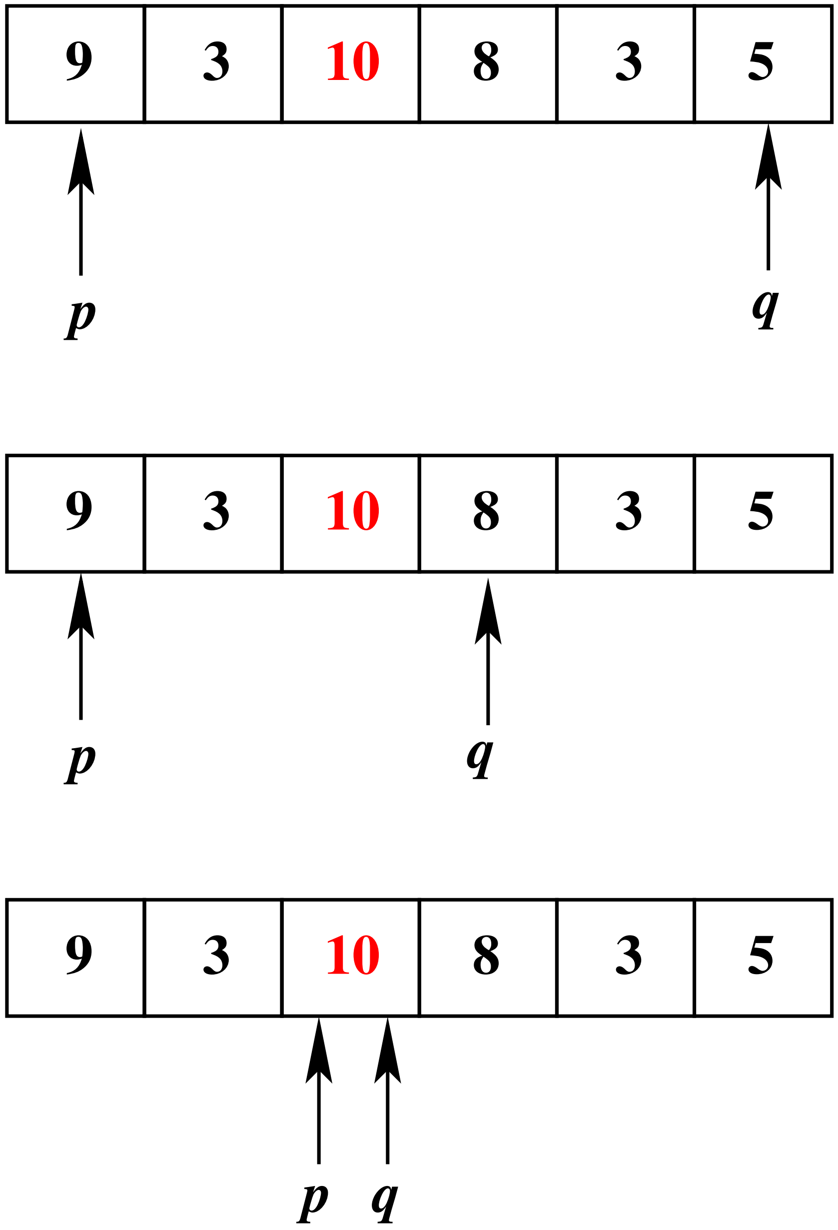
variables: m
p
q

inv1_1: $p \in 1\,..\,n$

inv1_2: $q \in 1\,..\,n$

Initial, Intermediate, and Final Situations

The maximum is always "between" p and q



Refinement: the Main Invariant

- Interval $p \mathrel{..} q$ is never empty (**inv1_3**)

- The maximum is always in the image of $p \mathrel{..} q$ under f (**inv1_4**)

variables: m
p
q

inv1_1: $p \in 1 \mathrel{..} n$

inv1_2: $q \in 1 \mathrel{..} n$

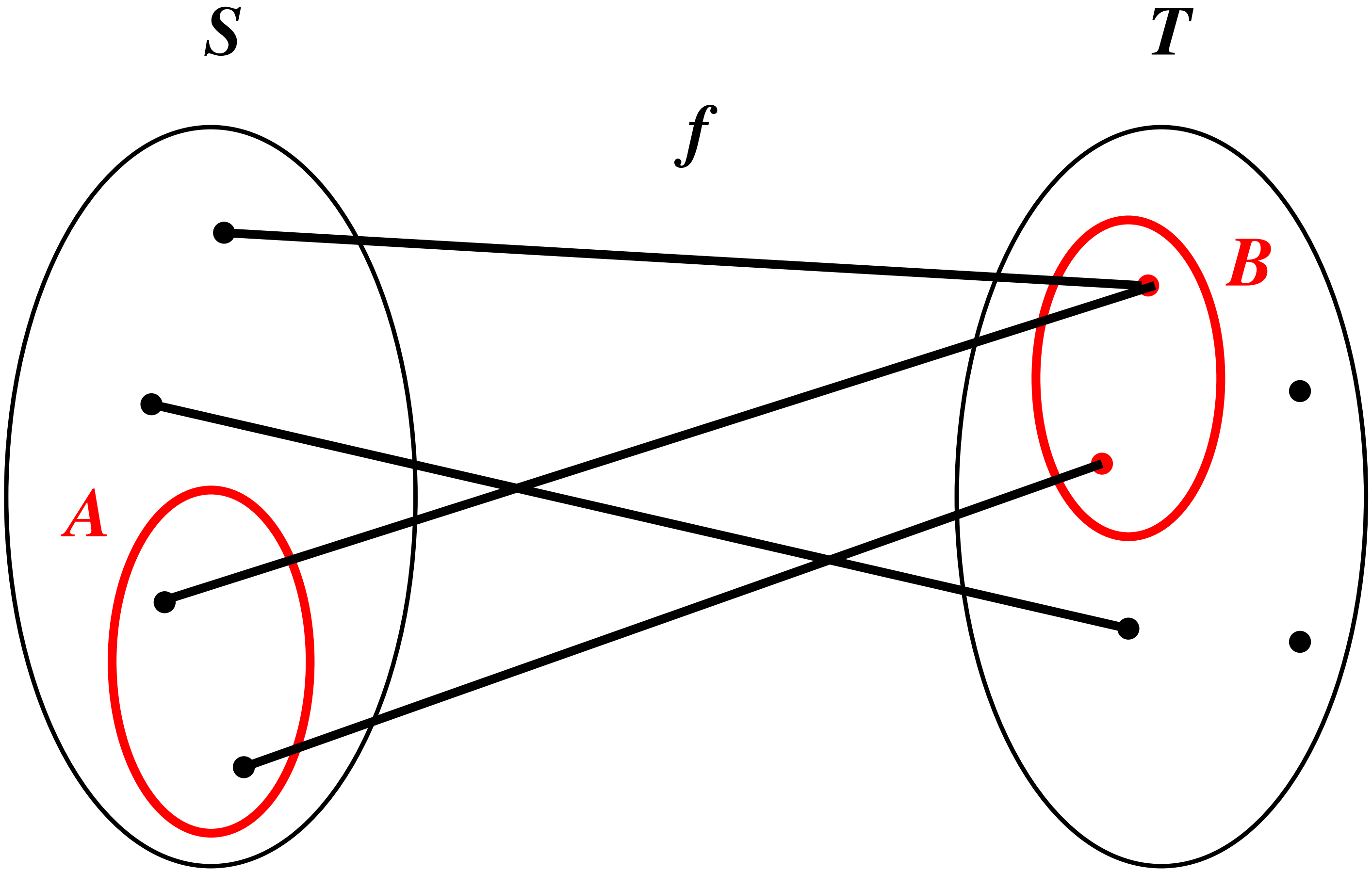
inv1_3: $p \leq q$

inv1_4: $\max(\mathrm{ran}(f)) \in f[p \mathrel{..} q]$

- **inv1_4** is the main invariant

Math Refresher: Image

- B is the image of A under f: $B = f[A]$



Refinement: Initial and Final Events

```
INIT
  begin
    $m := 0$
    $p := 1$
    $q := n$
  end
```

```
maximum
  when
    $p = q$
  then
    $m := f(p)$
  end
```

Refinement: two New Events

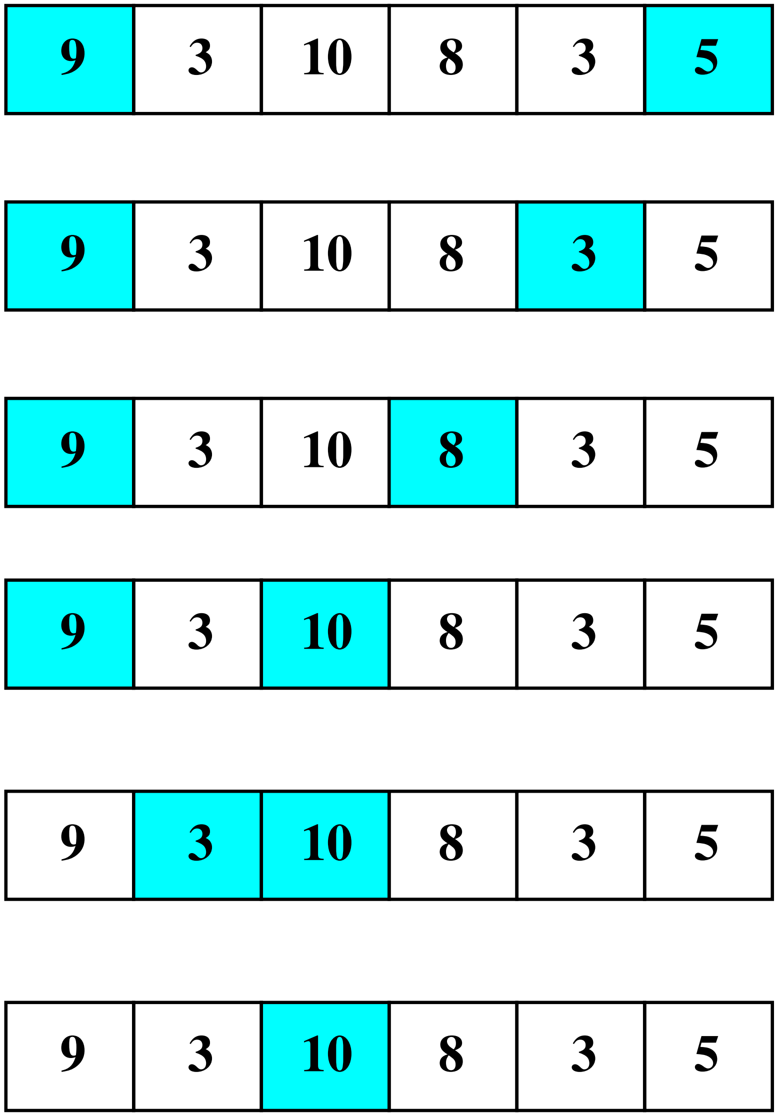
```
INIT
  begin
    $m := 0$
    $p := 1$
    $q := n$
  end
```

```
maximum
  when
    $p = q$
  then
    $m := f(p)$
  end
```

```
increment
  when
    $p < q$
    $f(p) \leq f(q)$
  then
    $p := p + 1$
  end
```

```
decrement
  when
    $p < q$
    $f(q) < f(p)$
  then
    $q := q - 1$
  end
```

Trace



9	3	10	8	3	5

5<9 (decrement)

9	3	10	8	3	5

3<9 (decrement)

9	3	10	8	3	5

8<9 (decrement)

9	3	10	8	3	5

9<10 (increment)

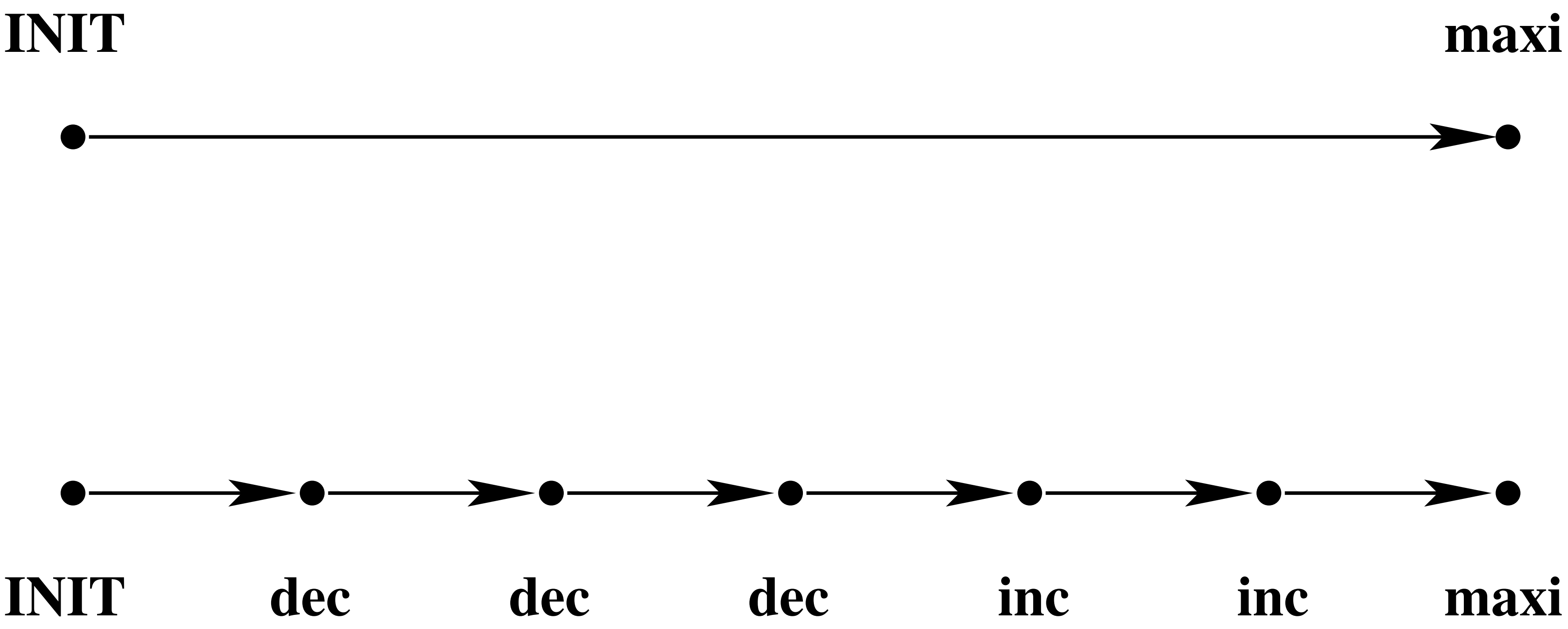
9	3	10	8	3	5

3<10 (increment)

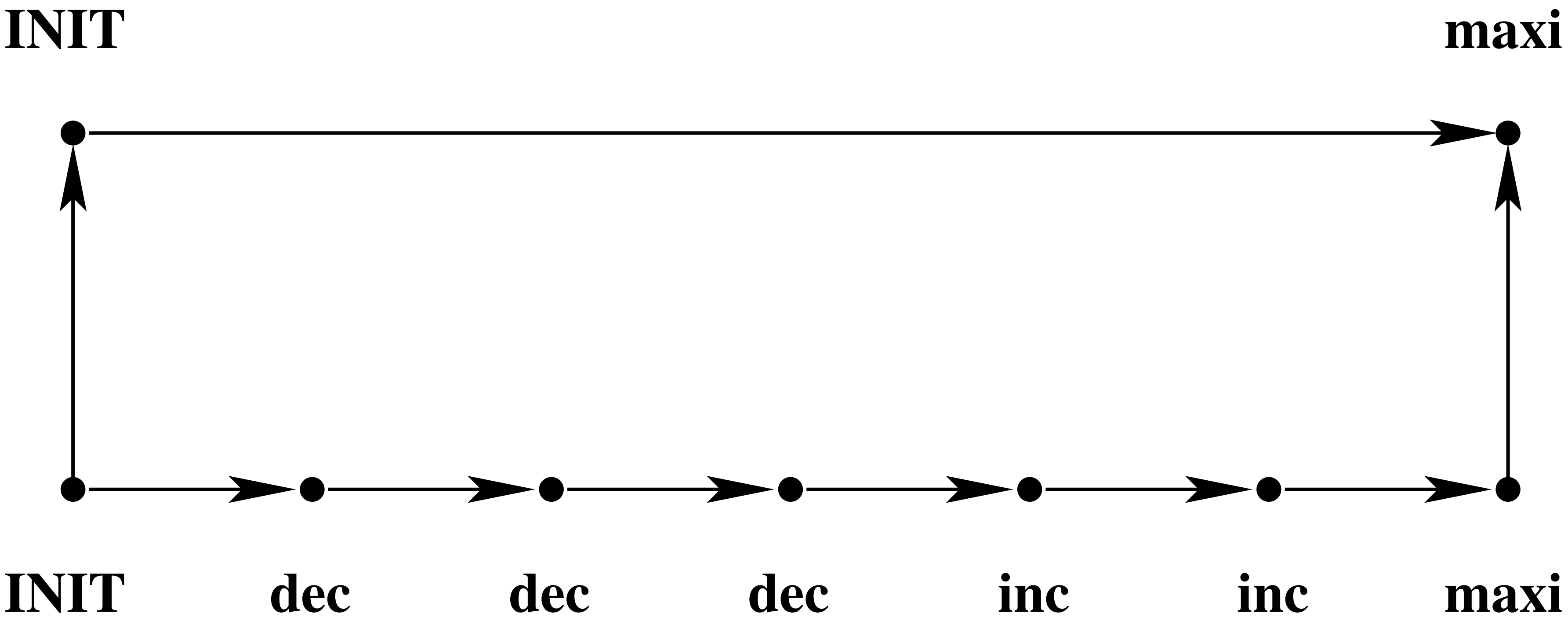
9	3	10	8	3	5

D E M O (showing a refinement)

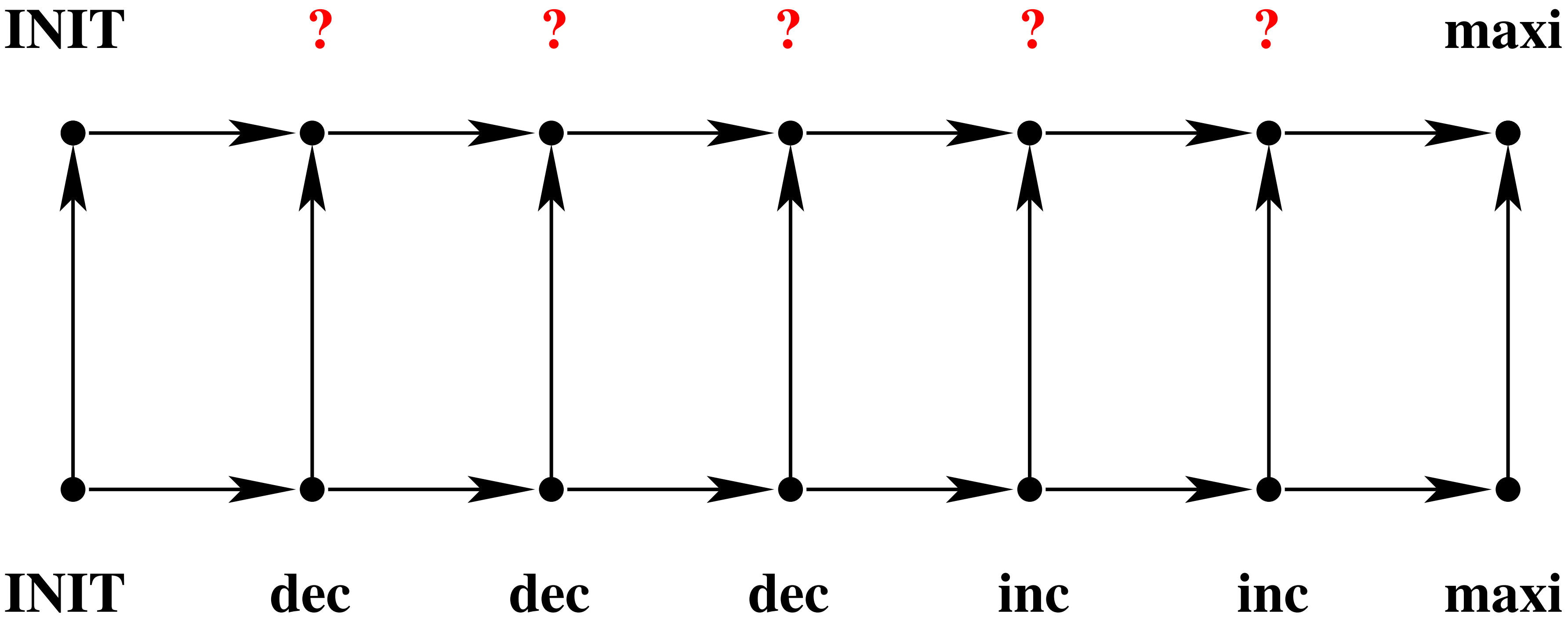
Abstract and Concrete Traces



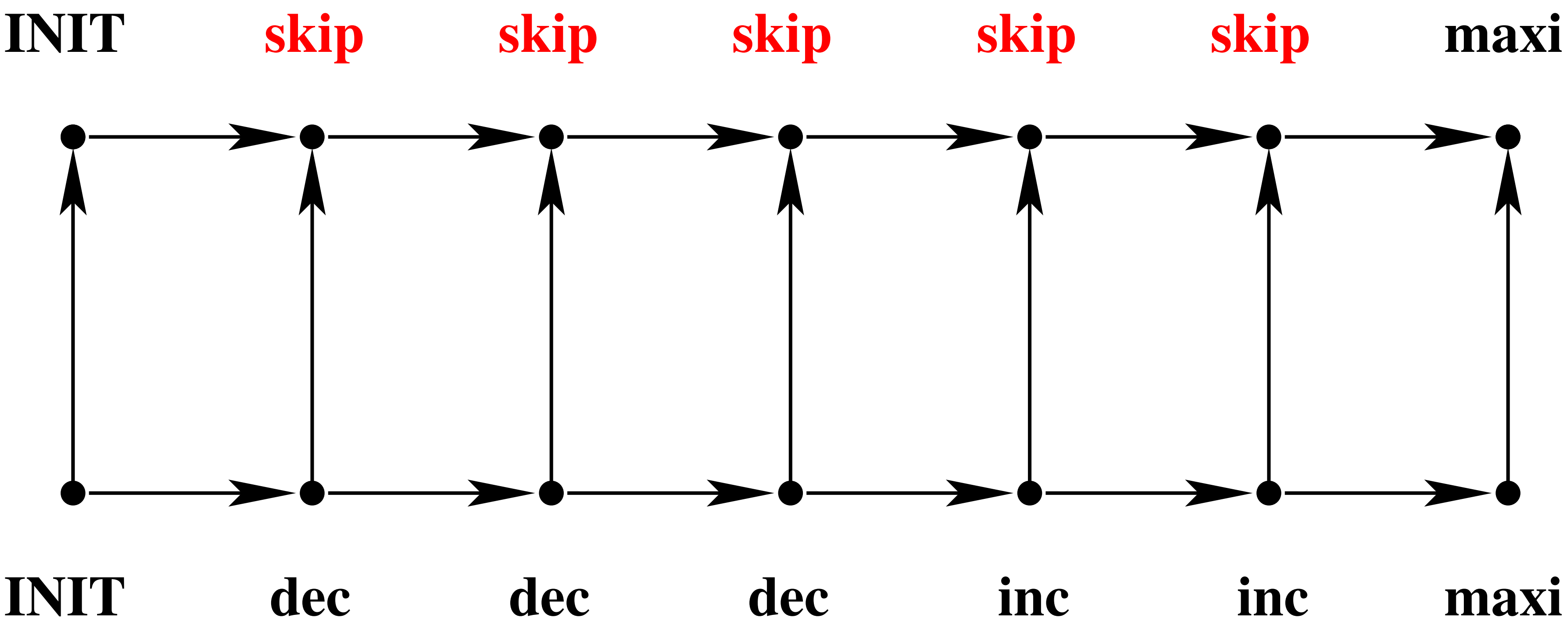
Old Events must Refine their Abstractions



How about New Events?



New Events refine "skip"



To be Proved

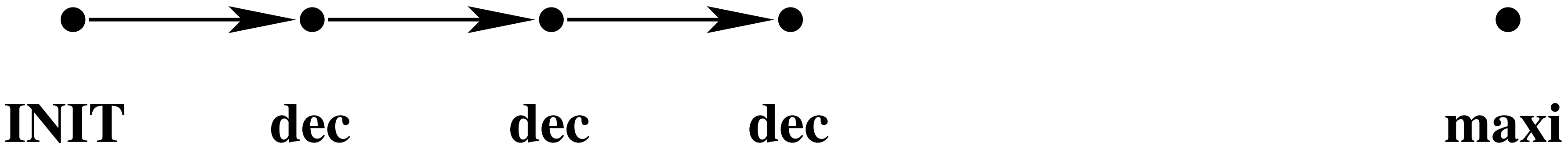
- Invariant maintenance

- Event refinement
    - guard strengthening
    - concrete action simulates the abstract one

- Well-definedness

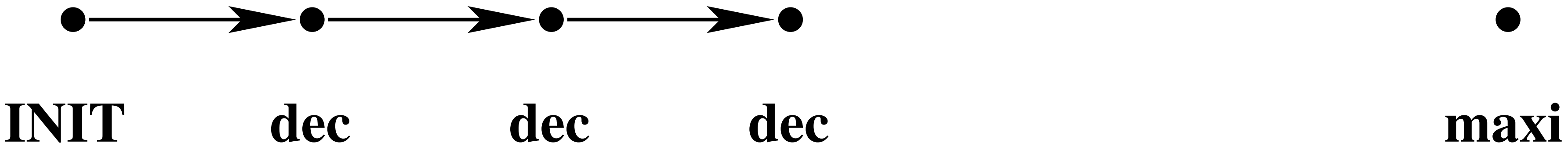
Pathologies which Must be Avoided on the Concrete Trace

- Early deadlock

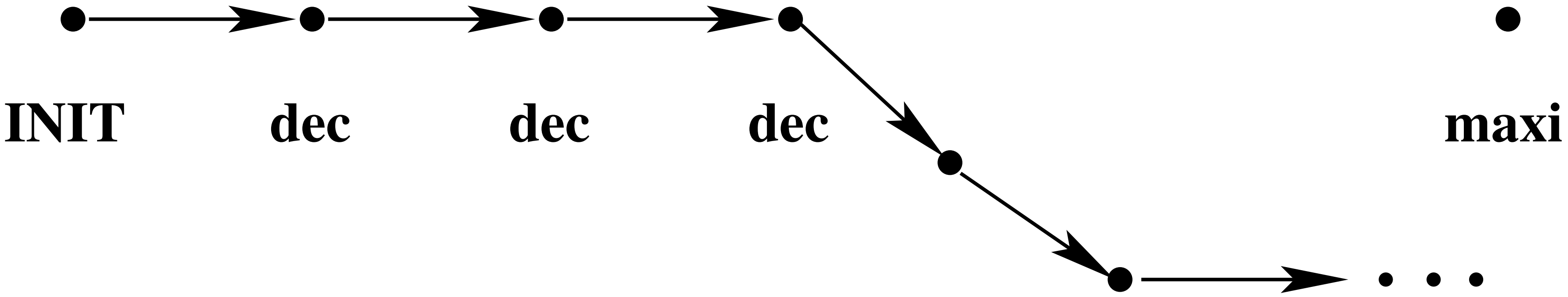


Pathologies which Must be Avoided on the Concrete Trace

- Early deadlock



- Divergence



To be Proved (more)

- Invariant maintenance
- Event refinement
  - guard strengthening
  - concrete action simulates the abstract one
- Well-definedness
- Trace refinement
  - Disjunction of guards must hold (no early deadlock)
  - New events must be convergent (must decrease a variant)

DEMO (showing more proof obligations)

Towards the Final Construction

```
INIT
  begin
    $m := 0$
    $p := 1$
    $q := n$
  end
```

```
maximum
  when
    $p = q$
  then
    $m := f(p)$
  end
```

```
increment
  when
    $p \neq q$
    $f(p) \leq f(q)$
  then
    $p := p + 1$
  end
```

```
decrement
  when
    $p \neq q$
    $f(q) < f(p)$
  then
    $q := q - 1$
  end
```

D E M O (showing an animation)

Statements for a Pidgin Programming Language

while $\mathit{condition}$ **do** $\mathit{statement}$ **end**

if $\mathit{condition}$ **then** $\mathit{statement}$ **else** $\mathit{statement}$ **end**

$\mathit{statement}\;;\;\mathit{statement}$

$\mathit{variable\_list} \;:=\; \mathit{expression\_list}$

IF Merging Rule

```
when
  $P$
  $Q$
then
  $S$
end
```

```
when
  $P$
  $\neg\, Q$
then
  $T$
end
```

$\leadsto$

```
when
  $P$
then
  if $Q$ then
    $S$
  else
    $T$
  end
end
```

M_IF

- The two events must have been introduced at the same step

Applying Rule M_IF

```
decrement
  when
    $p \neq q$
    $f(q) < f(p)$
  then
    $q := q - 1$
  end
```

```
increment
  when
    $p \neq q$
    $f(p) \leq f(q)$
  then
    $p := p + 1$
  end
```

```
decrement_increment
  when
    $p \neq q$
  then
    if  $f(q) < f(p)$  then
      $q := q - 1$
    else
      $p := p + 1$
    end
  end
```

WHILE Merging Rule (special case)

when Q **then** S **end**	**when** $\neg\, Q$ **then** T **end**	$\leadsto$	**while** Q **do** S **end**; T	**M_WHILE**

- The first event must have been introduced at one refinement step below the second one.

Applying Rule M_WHILE (special case)

```
decrement_increment
   when
      $p \neq q$
   then
      if  $f(q) < f(p)$  then
         $q := q - 1$
      else
         $p := p + 1$
      end
   end
```

```
maximum
   when
      $p = q$
   then
      $m := f(p)$
   end
```

```
decrement_increment_maximum
   while  $p \neq q$  do
      if  $f(q) < f(p)$  then
         $q := q - 1$
      else
         $p := p + 1$
      end
   end;
   $m := f(p)$
```

WHILE Merging Rule (general case)

```
when
  $P$
  $Q$
then
  $S$
end
```

```
when
  $P$
  $\neg Q$
then
  $T$
end
```

$\leadsto$

```
when
  $P$
then
  while $Q$ do
    $S$
  end;
  $T$
end
```

M_WHILE

- P must be invariant under S
- The first event must have been introduced at one refinement step below the second one.

Final Rule M_INIT

- Once we have obtained an “event” without guard

- We add to it the event init by sequential composition

- We then obtain the final “program”

The Program: Putting the Events Together

```
$m, p, q := 0, 1, n;$                          INIT
while $p < q$ do
   if $f(q) < f(p)$ then
      $q := q - 1$                             decrement
   else
      $p := p + 1$                             increment
   end
end;
$m := f(p)$                                    maximum
```

```
INIT
   begin
      $m := 0$
      $p := 1$
      $q := n$
   end
```

```
decrement
   when
      $p < q$
      $f(q) < f(p)$
   then
      $q := q - 1$
   end
```

```
increment
   when
      $p < q$
      $f(p) \leq f(q)$
   then
      $p := p + 1$
   end
```

```
maximum
   when
      $p = q$
   then
      $m := f(p)$
   end
```

Exercise

- Modify the development to search for the minimum of the array

```
$m, p, q := 0, 1, n;$                     INIT
while $p < q$ do
    if $f(p) > f(q)$ then
        $p := p + 1$                      increment
    else
        $q := q - 1$                      decrement
    end
end;
$m := f(p)$                               maximum
```

1.3.3. A Second Simple Example

Requirements for a Second Simple Example

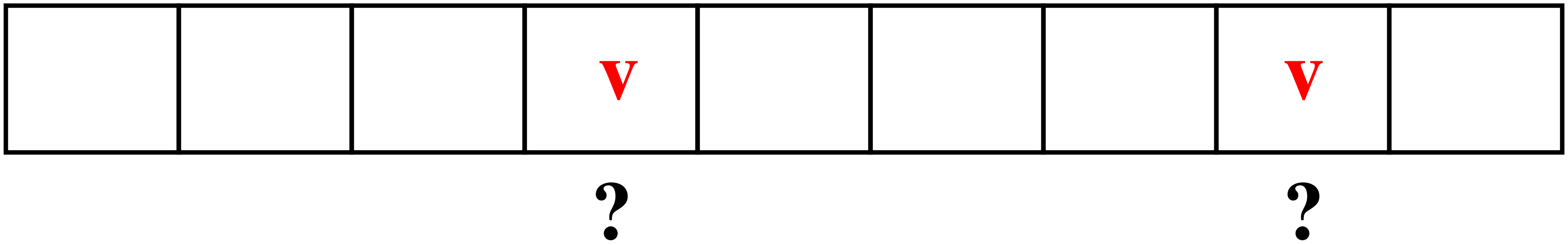
We are given a non-empty finite array	FUN-1

We know that a value v is in this array	FUN-2

We like to find an index with v	FUN-3

Requirements for a Second Simple Example

We are given a non-empty finite array	FUN-1
We know that a value v is in this array	FUN-2
We like to find an index with v	FUN-3



Development Strategy

- First, we show an initial model specifying the problem

- Later, we refine our model to produce an algorithm.

- In the initial model, we have:

    - a context where the constant array is defined

    - a machine where the search is done (non-deterministically)

Context

sets: D

axm1: $n \in \mathbb{N}$

axm2: $f \in 1\,..\,n \longrightarrow D$

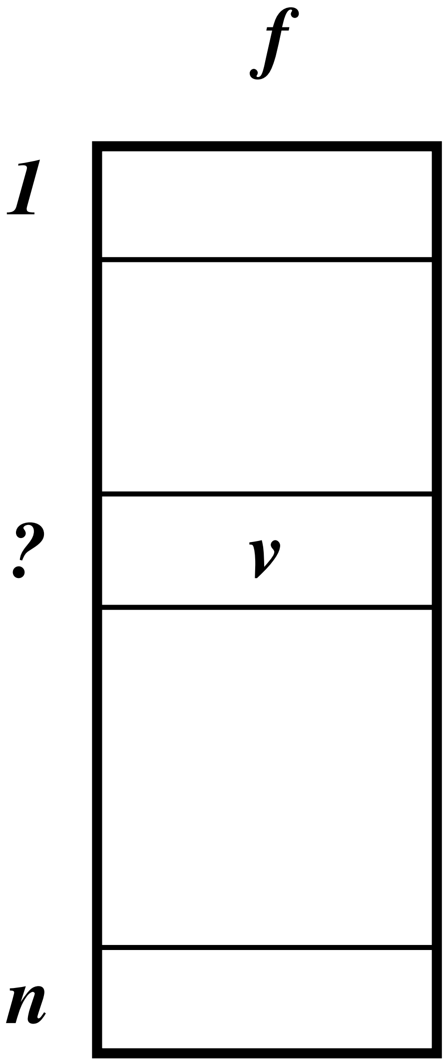
axm3: $v \in \mathrm{ran}(f)$

constants: n
f
v

thm1: $n > 0$

- This context is generic: the set D is not specified
  (just supposed to be non-empty)

Pictorial Representation of the Context



Machine

variables: i

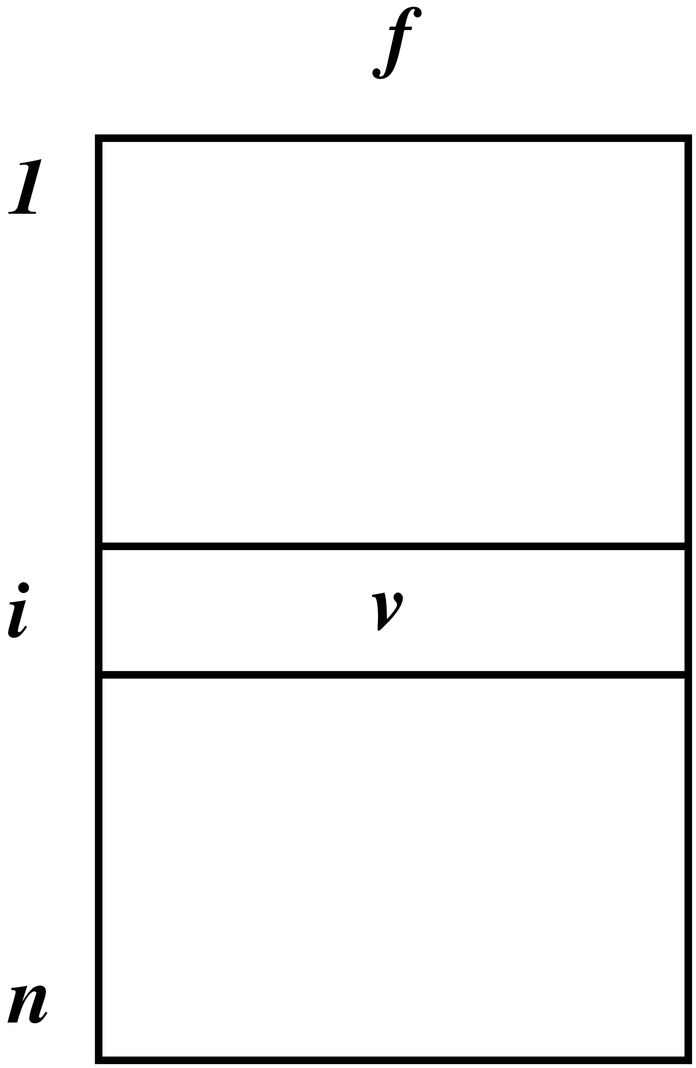
inv1: $i \in 1\,..\,n$

```
INIT
  begin
    $i := 1$
  end
```

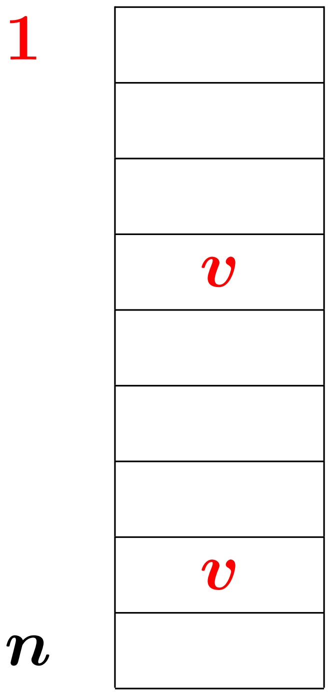
```
search
  any
    $k$
  where
    $k \in 1\,..\,n$
    $f(k) = v$
  then
    $i := k$
  end
```

Notice the quantified guard
in event "search"

Pictorial Representation of the State after Event "search"



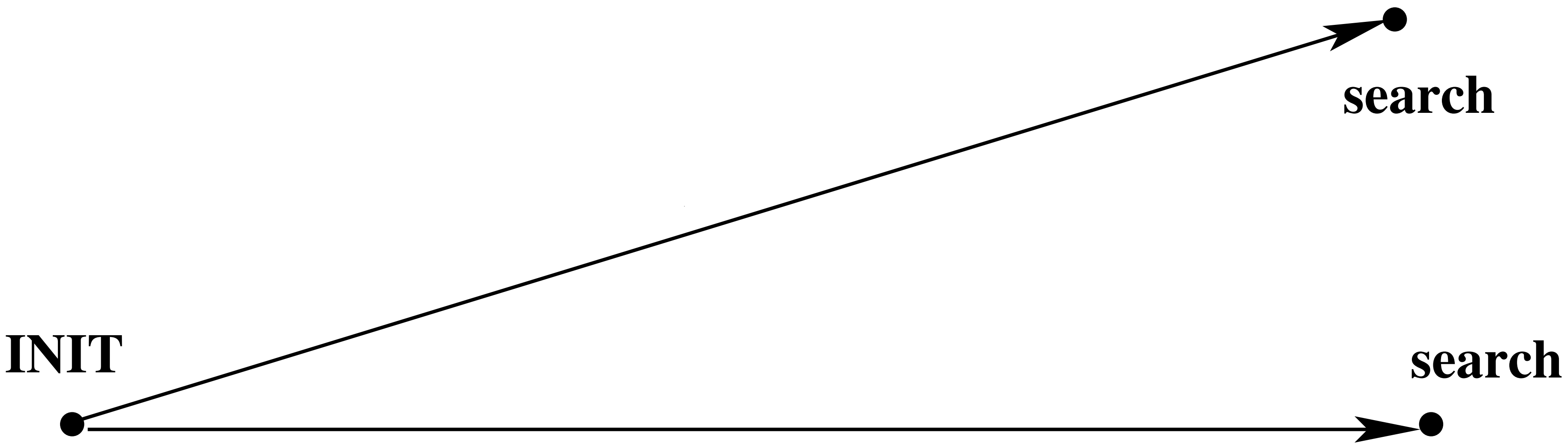
Abstract Trace: Where is v?



Abstract Trace: v is in 4, v is in 8

The abstract trace is non-deterministic

Abstract Trace is Non-deterministic



D E M O (showing the context and the machine)

Refinement: New Variable j and New Event progress

variables: i
j

inv1_1: $j \in 0 \,..\, n$

inv1_2: $v \notin f[1 \,..\, j]$

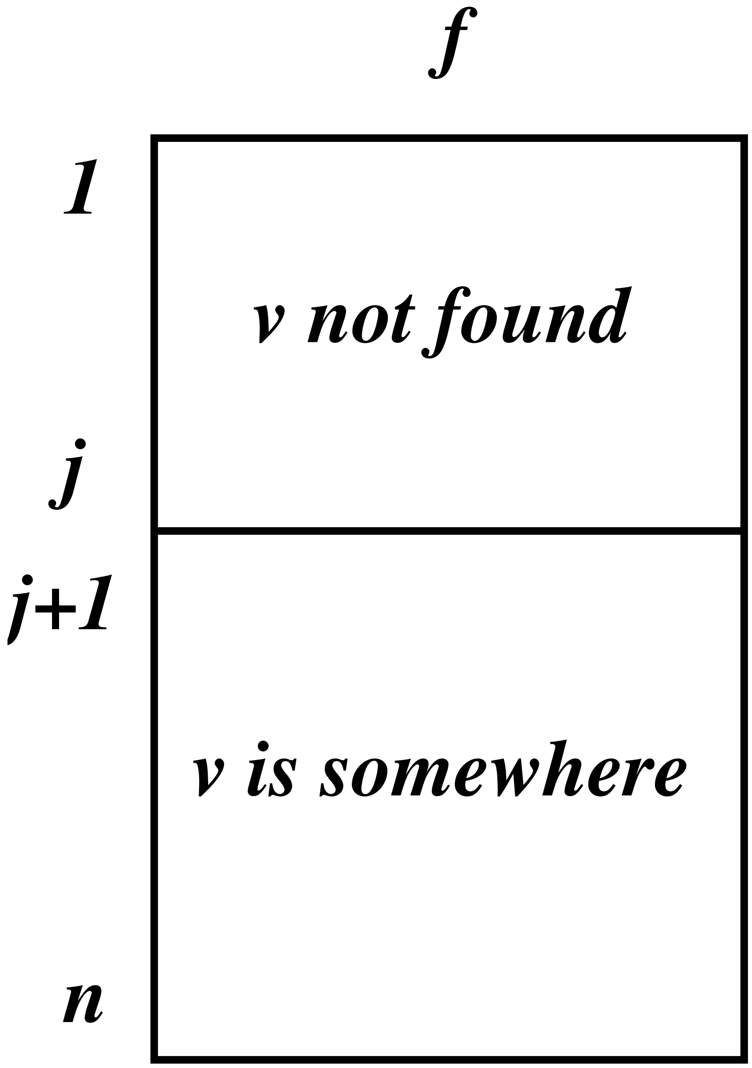
thm1_1: $j + 1 \in 1..n$

INIT
  when
    $i := 1$
    $j := 0$
  end

progress
  when
    $f(j+1) \neq v$
  then
    $j := j+1$
  end

success
  when
    $f(j+1) = v$
  then
    $i := j+1$
  end

Pictorial Representation of the State



f

1

v not found

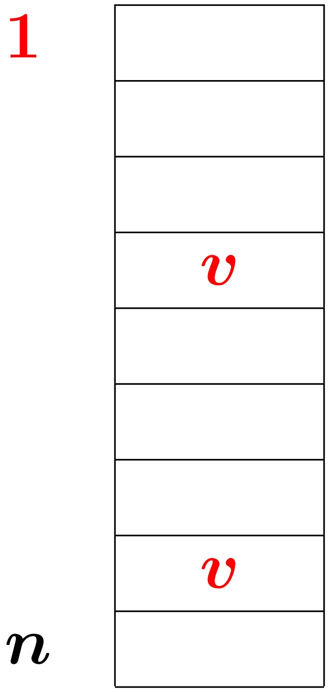
j

j+1

v is somewhere

n

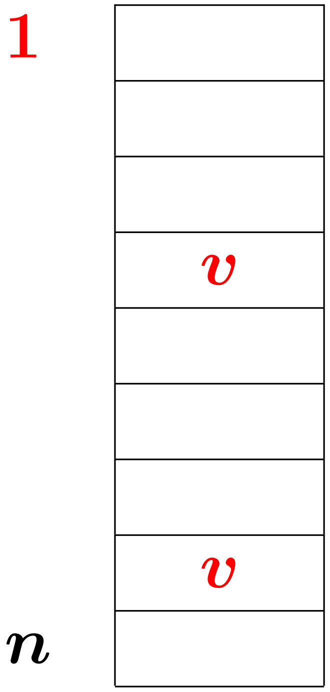
Abstract Trace: Where is v?



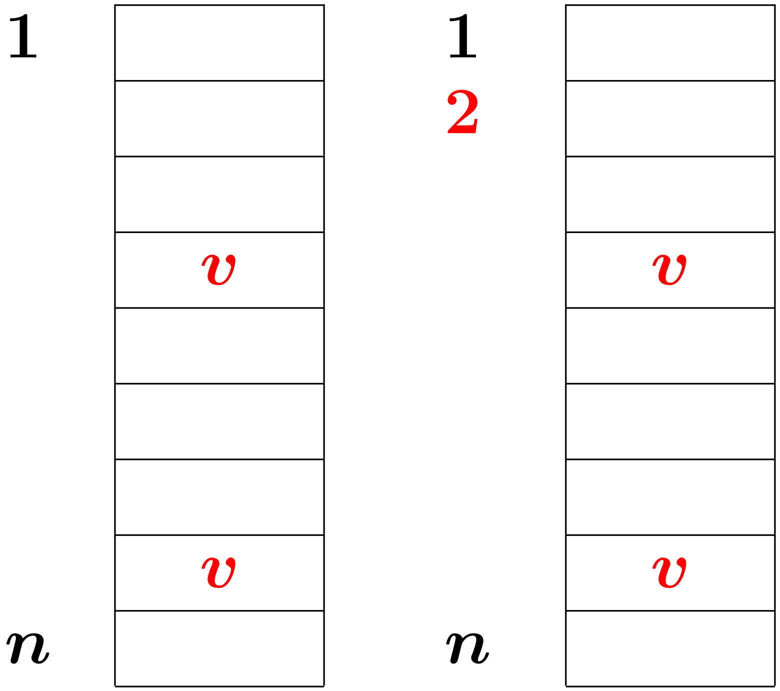
Abstract Trace: v is in 4, v is in 8

The abstract trace is non-deterministic

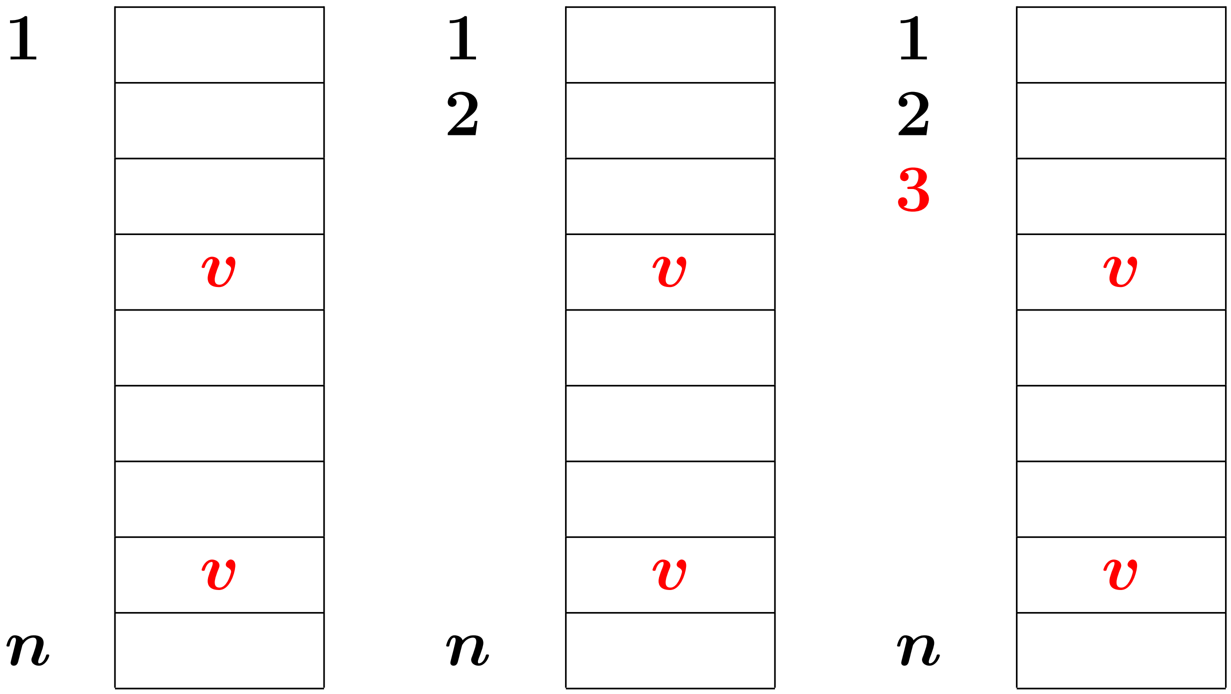
Concrete Trace: v is not in 1



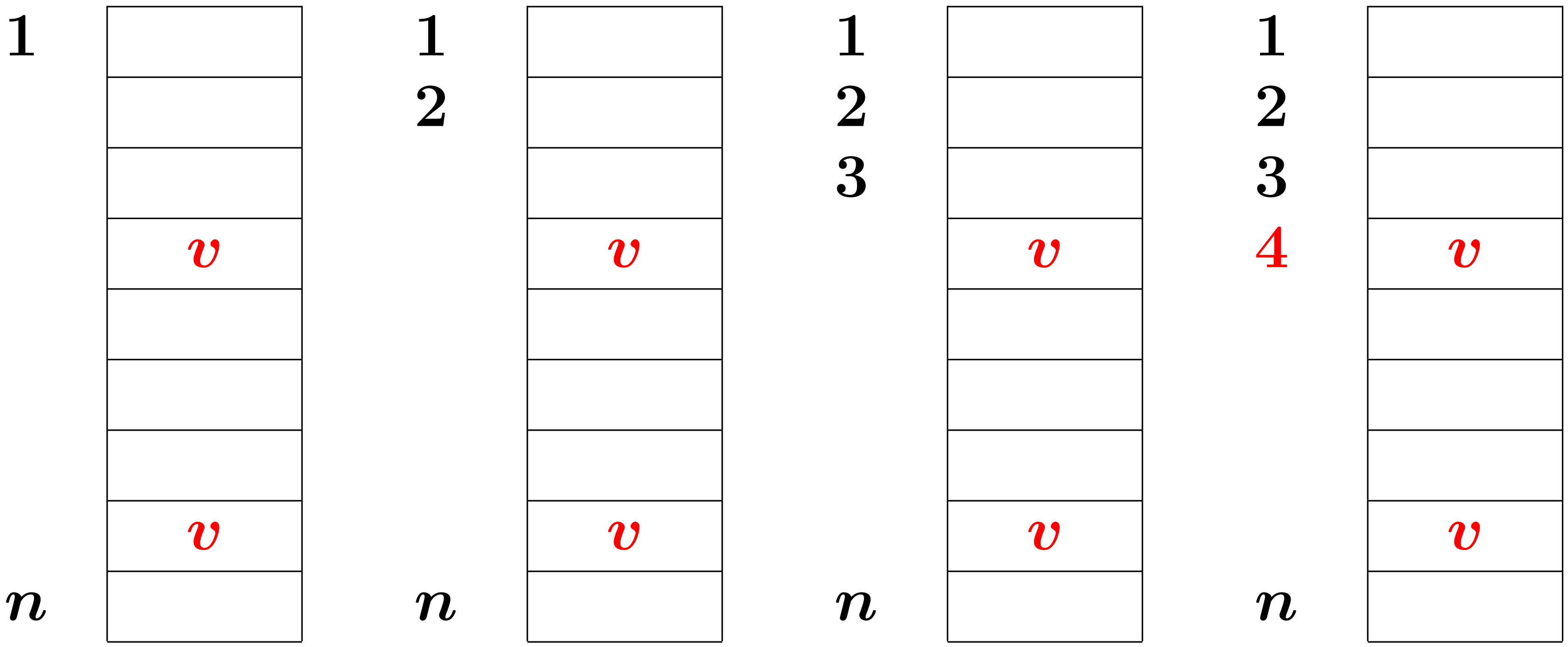
Concrete Trace: v is not in 2



Concrete Trace: v is not in 3

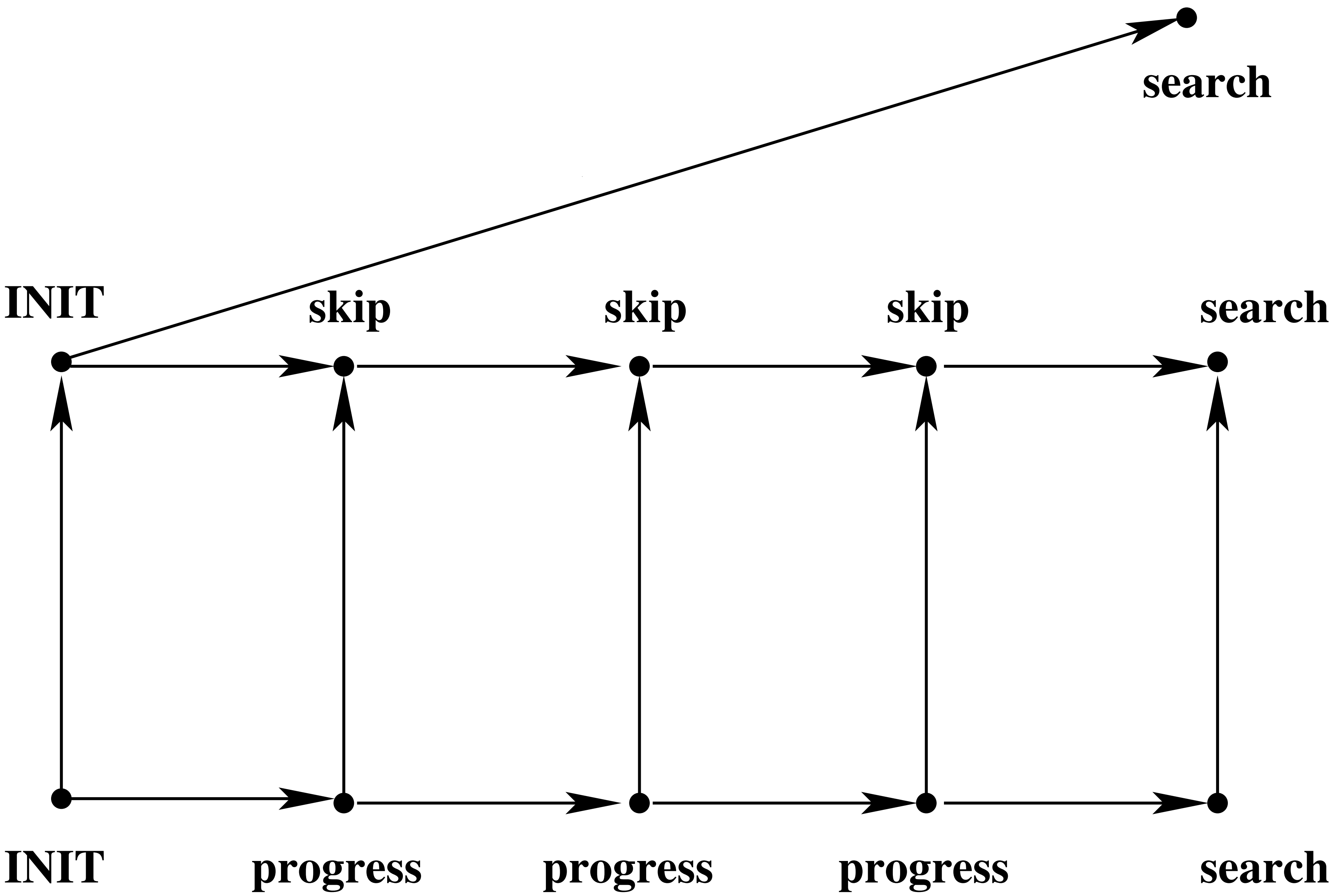


Concrete Trace: v is in 4



The concrete trace is deterministic

Concrete Trace is Deterministic



DEMO (showing the proof obligation)

Constructing the Final Program

```
$i, j := 1, 0$ ;                        INIT
while  $f(j+1) \neq v$  do
   $j := j + 1$                         progress
end ;
$i := j + 1$                            search
```

```
INIT
   begin
      $i := 1$
      $j := 0$
   end
```

```
progress
   when
      $f(j+1) \neq v$
   then
      $j := j + 1$
   end
```

```
success
   when
      $f(j+1) = v$
   then
      $i := j + 1$
   end
```

Exercises

- Modify the development in order to obtain the following program:

```
$i, j := 1, n + 1$ ;                                  INIT
while $f(j-1) \neq v$ do
    $j := j - 1$                                      progress
end ;
$i := j - 1$                                          search
```

- Develop more elaborate array searching algorithms:
    - from both sides alternatively,
    - from somewhere inside and alternatively,
    - on a sorted array
    - …

Conclusion: Importance of Refinement

- Refinement allows us to build models gradually
- We build an ordered sequence of more precise models
- Each model is a refinement of the one preceding it
- A useful analogy: looking through a microscope
- Spatial (more variables) as well as temporal (more events) extensions

Methods and Tools for System and Software Construction

2. A Mechanical Press Controller

Jean-Raymond Abrial (ETHZ)

August 2008

Outline

1. Informal presentation of the example
2. Presentation of some design patterns
3. Writing the requirement document
4. Proposing a refinement strategy
5. Development of the model using refinements and design patterns
6. Demos

1. Informal Presentation of the Example

Presentation of the Example

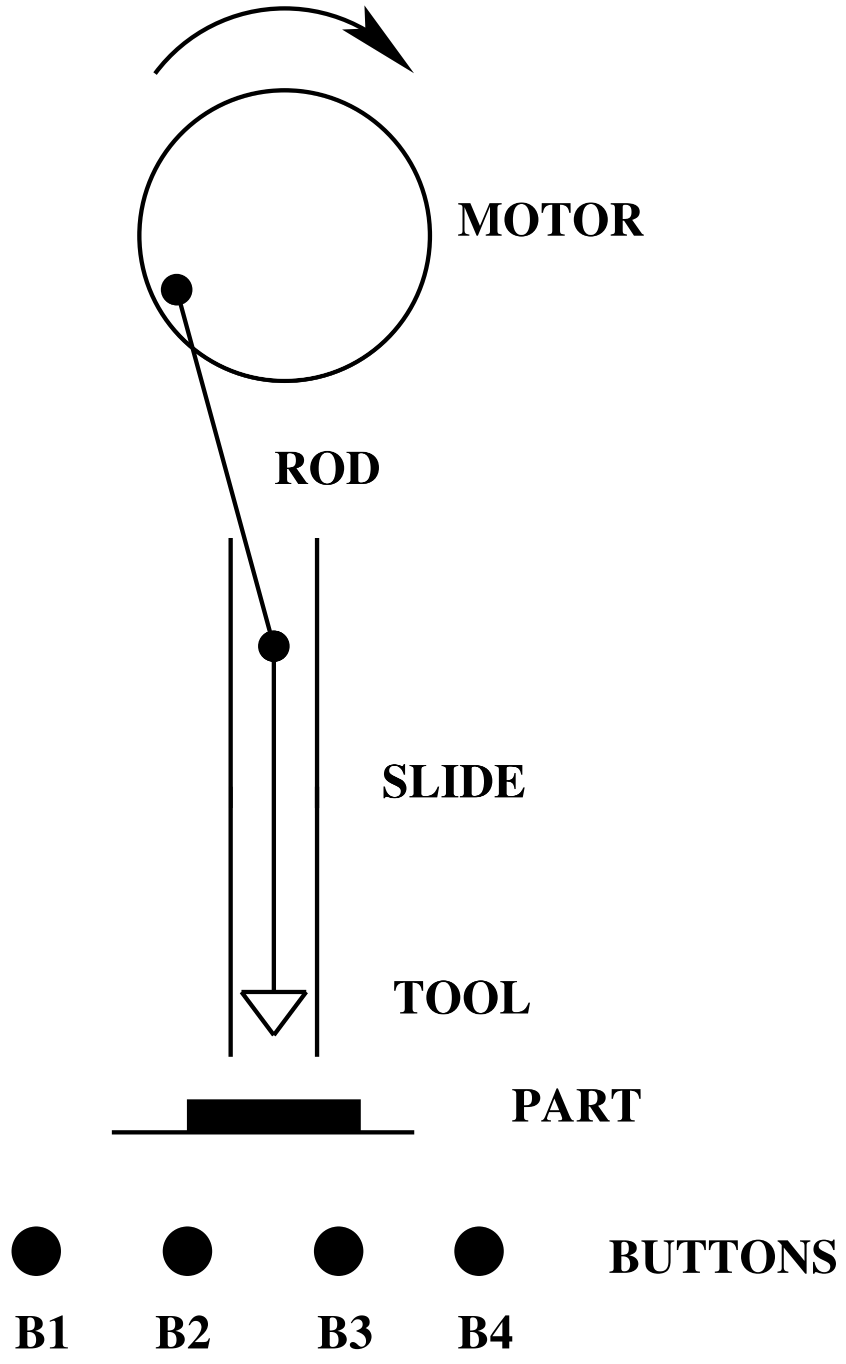
- A mechanical press controller

- Adapted from a real system

- The real system is coming from INRST:

  Institut National de la Recherche sur la Sécurité du Travail

Mechanical Press Schema



Basic Equipment

- A Vertical Slide with a tool at its lower extremity
- An electrical Rotating Motor
- A Rod connecting the motor to the slide.
- A Clutch engaging or disengaging the motor on the rod
- When the clutch is disengaged, the slide stops “immediately”

Basic Commands

- Button B1: start motor

- Button B2: stop motor

- Button B3: engage clutch

- Button B4: disengage clutch

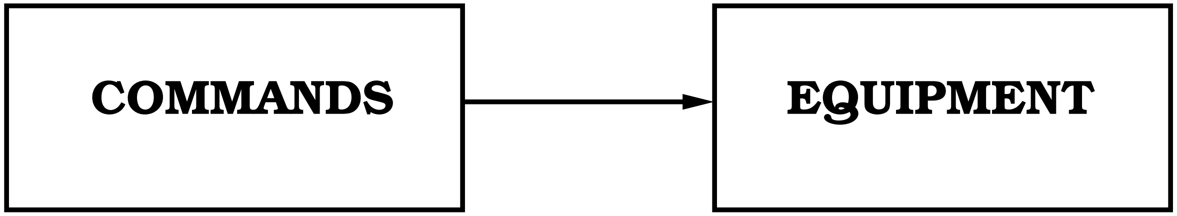
Basic User Actions

- 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

First Schematic View



COMMANDS

EQUIPMENT

A Typical User Session

1. start the motor (button B1)
2. change the tool (action 1)
3. put a part (action 2),
4. engage the clutch (button B3): the press now works,
5. disengage the clutch (button B4): the press does not work,
6. remove the part (action 3),
7. repeat zero or more times steps 3 to 6,
8. repeat zero or more times steps 2 to 7,
9. stop the motor (button B2).

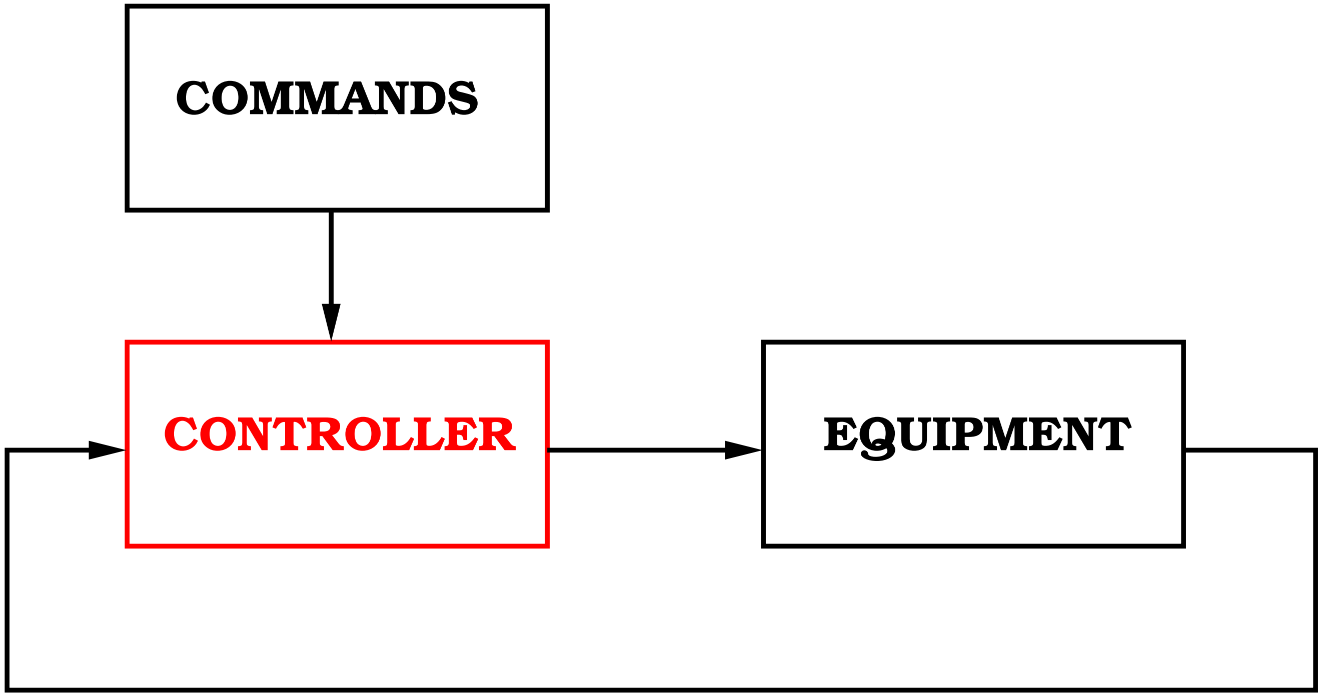
Danger: Necessity of a Controller

- step 2 (change the tool),

- step 3 (put a part),

- step 6 (remove the part) are all DANGEROUS

Second Schematic View

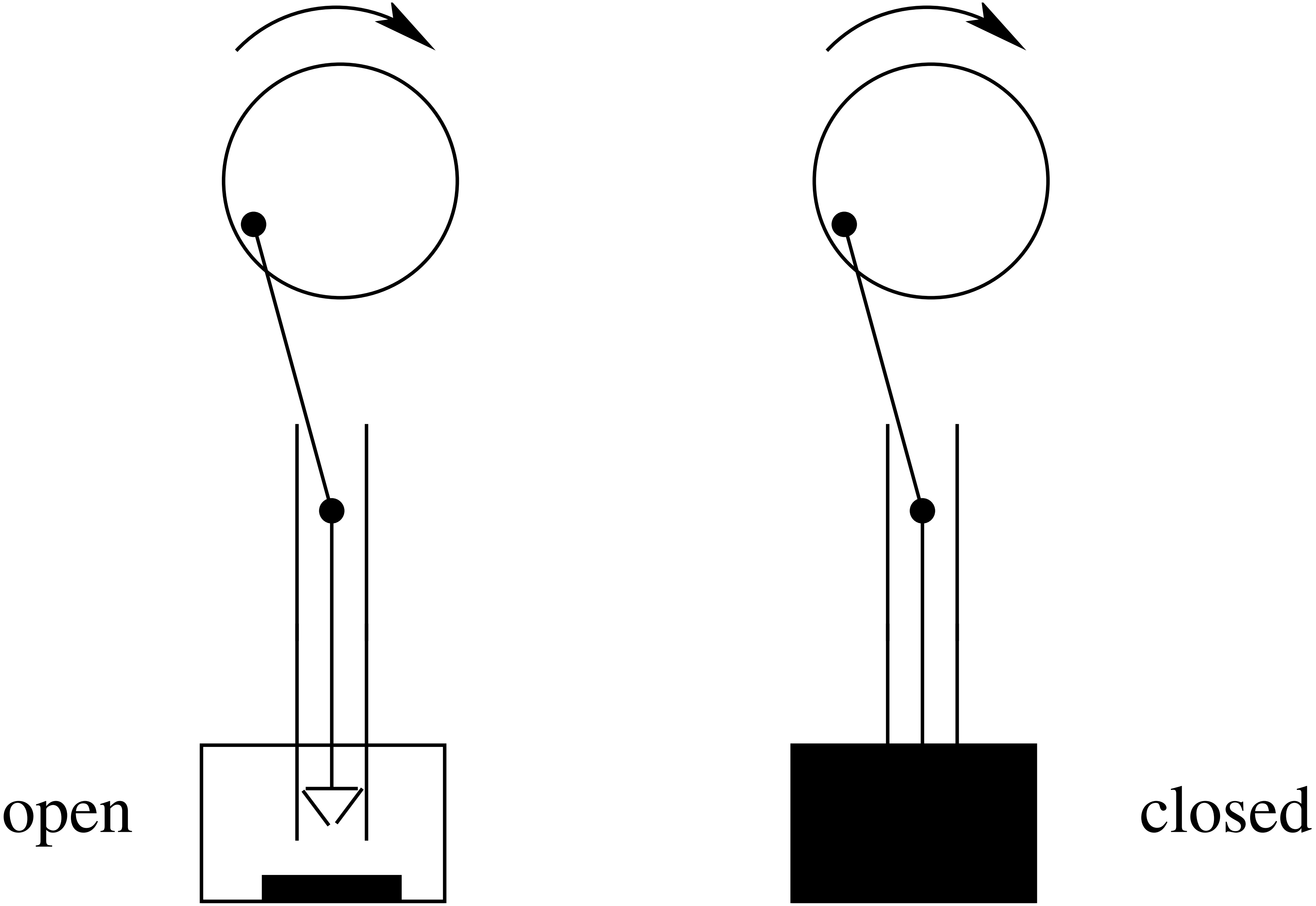


More Elaborate Commands for Protecting the User

- Controlling the way the clutch is engaged or disengaged

- Protection by means of a Front Door

The Front Door



The Front Door: Behavior

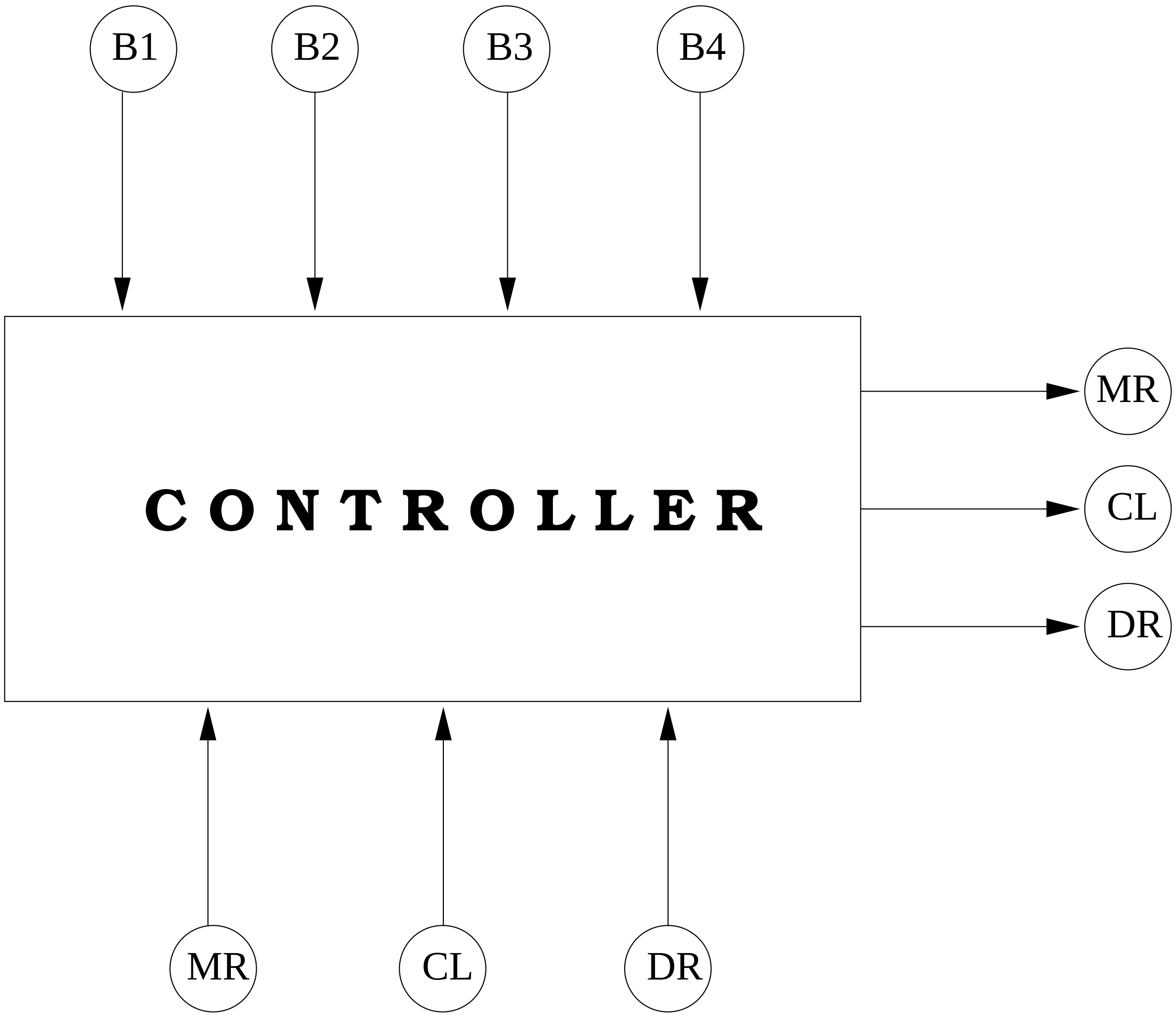
- Initially, the door is open

- When the user presses button B3 to engage the clutch,
  the door is first closed BEFORE engaging the clutch

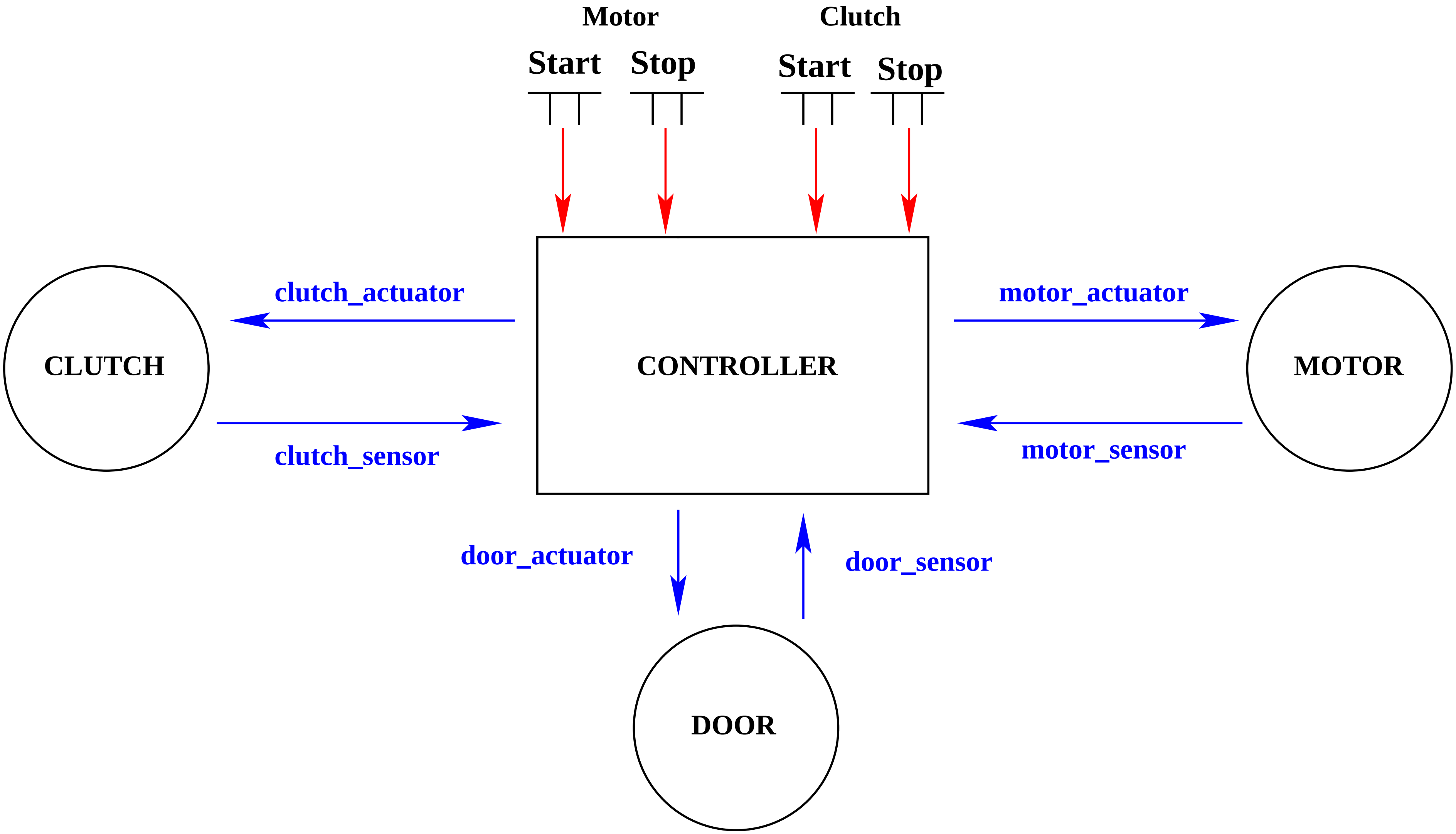
- When the user presses button B4 to disengage the clutch,
  the door is opened AFTER disengaging the clutch

- Notice: The door has no button.

Summary of Connections



Overview


Motor
Start
Stop
Clutch
Start
Stop
clutch_actuator
CLUTCH
clutch_sensor
CONTROLLER
motor_actuator
MOTOR
motor_sensor
door_actuator
door_sensor
DOOR

2. Presentation of some Design Patterns

Motivations

- A number of similar behaviors

- Some complex situations to handle

Similar Behavior (1)

- 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
  - . . .

Similar Behavior (2)

- 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

Similar Behavior (3)

- Making an action dependent of another one:

- Engaging the clutch implies closing the door first

- Disengaging the clutch means opening the door afterwards

Motivation: Example of Some Complex Situation

- Here is a sequence of events:

(1) User pushes button B1 (start motor)

(1’) User does not remove his finger from button B1

(2) Controller sends the starting command to the motor

(3) Motor starts and sends feedback to the controller

(4) Controller is aware that the motor works

(5) User pushes button B2 (stop motor)

(6) Controller sends the stop command to the motor

(7) Motor stops and sends feedback to the controller

(8) Controller is aware that the motor does not work

(9) Controller must not send the starting command to the motor

Motivation: Example of Some Complex Situation

- Here is a sequence of events:

  (1) User pushes button B1 (start motor)

  (2) Controller sends the starting command to the motor

  (3.1) Motor starts and sends feedback to the controller

  (3.2) User pushes button B2 (stop motor)

- (3.1) and (3.2) may occur simultaneously

- If controller treats (3.1) before (3.2): motor is stopped

- If controller treats (3.2) before (3.1): motor is not stopped

Design Patterns in Formal Developments

- We want to build systems which are correct by construction

- We want to have more methods for doing so

- "Design pattern" is an Object Oriented concept

- We would like to borrow this concept for doing formal developments

- A preliminary tentative with reactive system developments

- Advantage: systematic developments and also refinement of proofs

A Personal View on Design Patterns

- 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?

- Let’s apply this approach to the design of reactive systems

A Wikipedia View on Design Patterns (1)

- A design pattern isn’t a finished design that can be transformed into code

- It is a template for how to solve a problem that can be used in many different situations

- Patterns originated as an architectural concept by Christopher Alexander

- "Design Patterns: Elements of Reusable Object-Oriented Software" published in 1994 (Gamma et al)

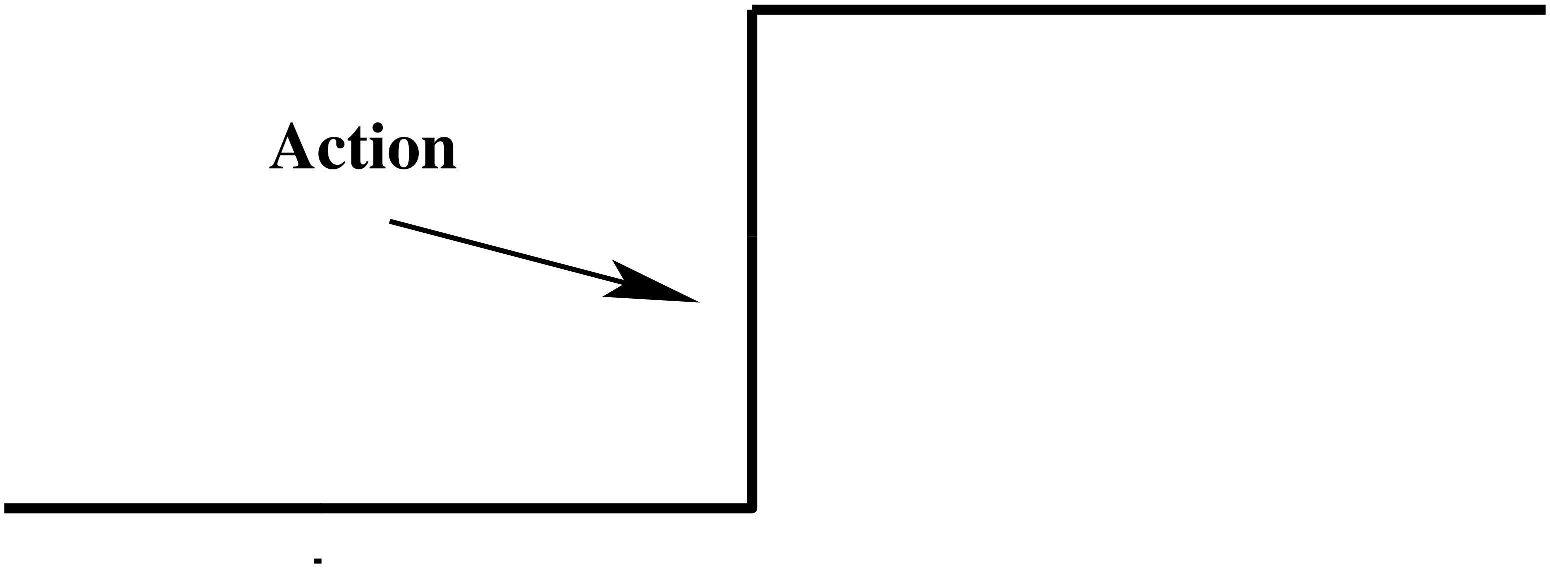
A Wikipedia View on Design Patterns (2)

- Design pattern can speed up the development process by providing tested and proven development paradigms

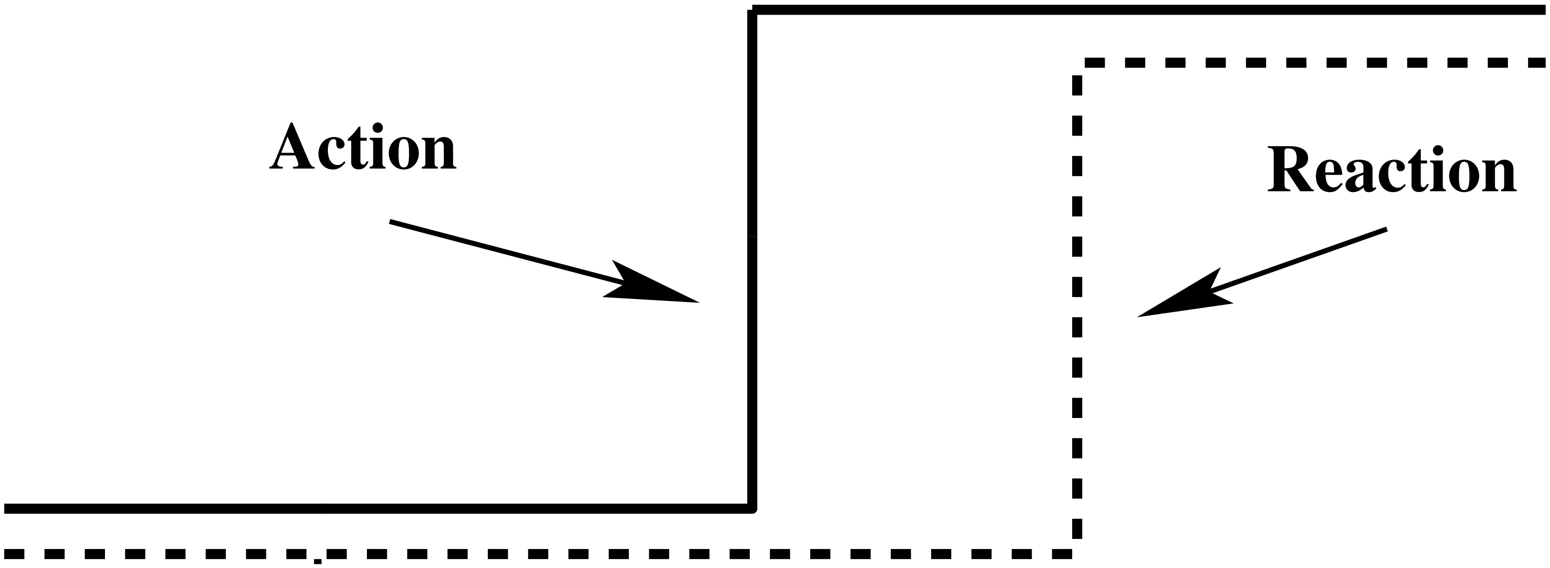
- The documentation for a design pattern should contain enough information about the problem that the pattern addresses, the context in which it is used, and the suggested solution.

- Some feel that the need for patterns results from using computer languages or techniques with insufficient abstraction

An Action Pattern



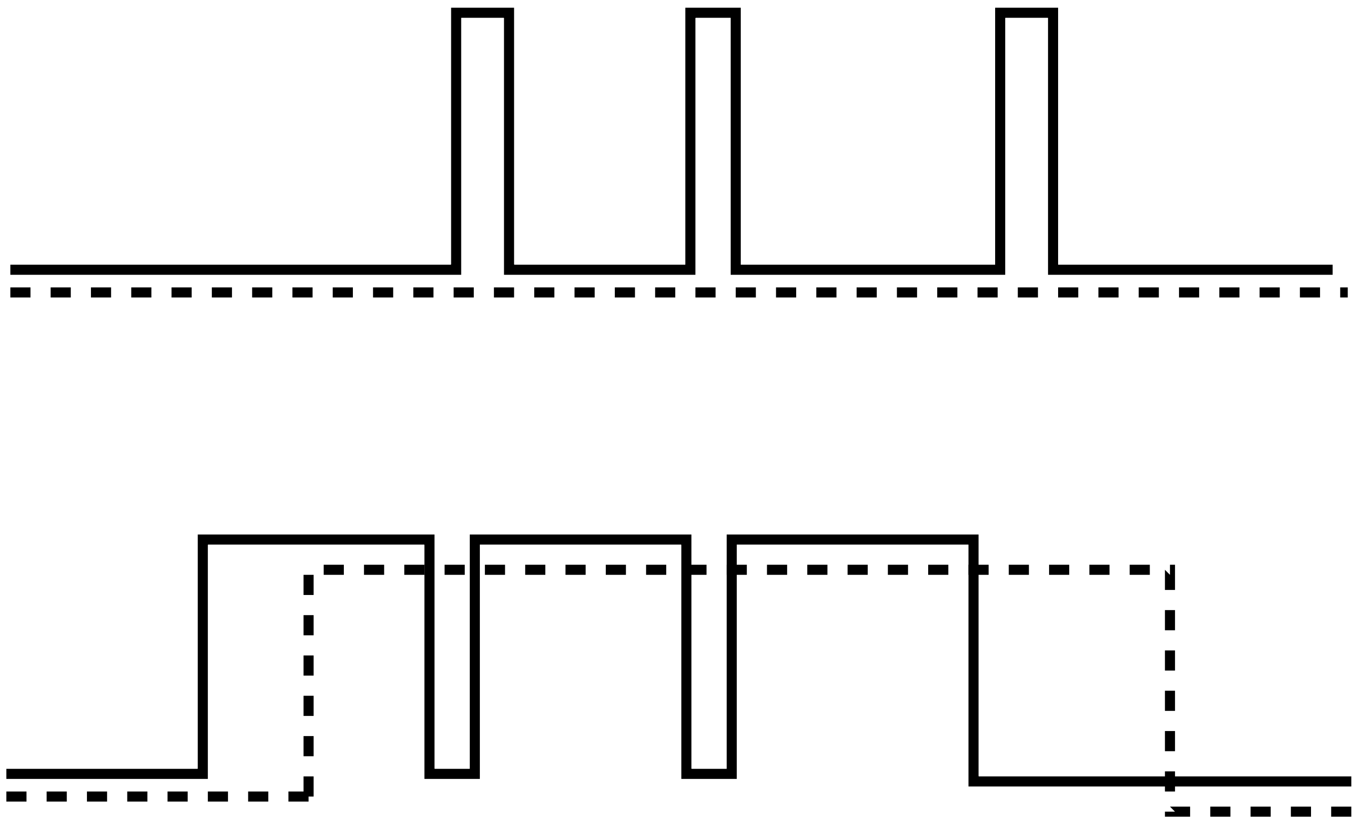
Action and Reaction Patterns



Action and Weak Reaction Patterns

- Sometimes, the reaction has not enough time to react

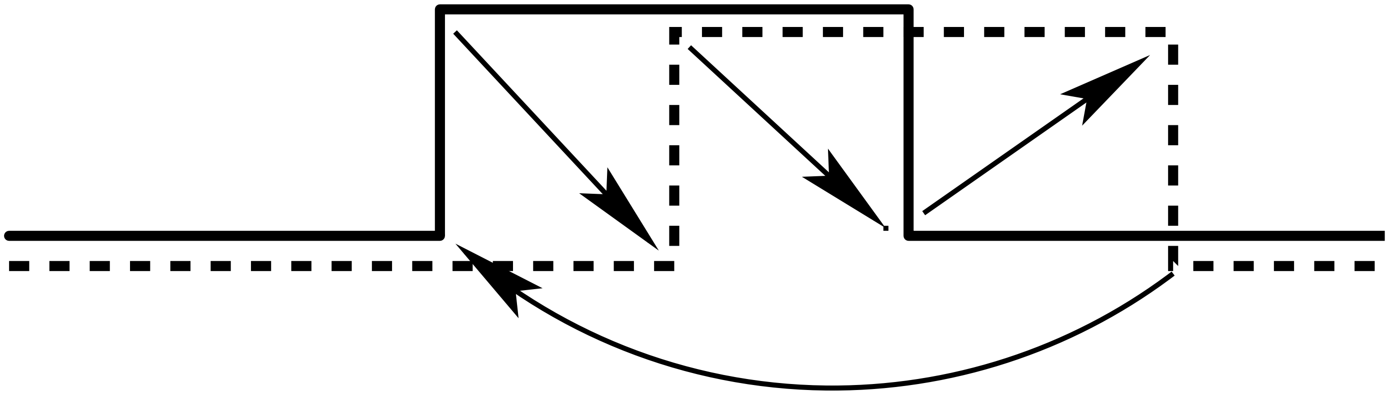
- Because the action moves too quickly



Action and Strong Reaction Patterns

- Sometimes, the reaction always follows the action

- They are both synchronized



Constructing Models of Actions and Reactions

- We built first a model of a weak reaction

- The strong reaction will be a refinement of the weak one

Model for weak action and reaction: the State

variables: a
r

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

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

- a denotes the action

- r denotes the reaction

Model for weak action and reaction: the State

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: $cr \in \mathbb{N}$

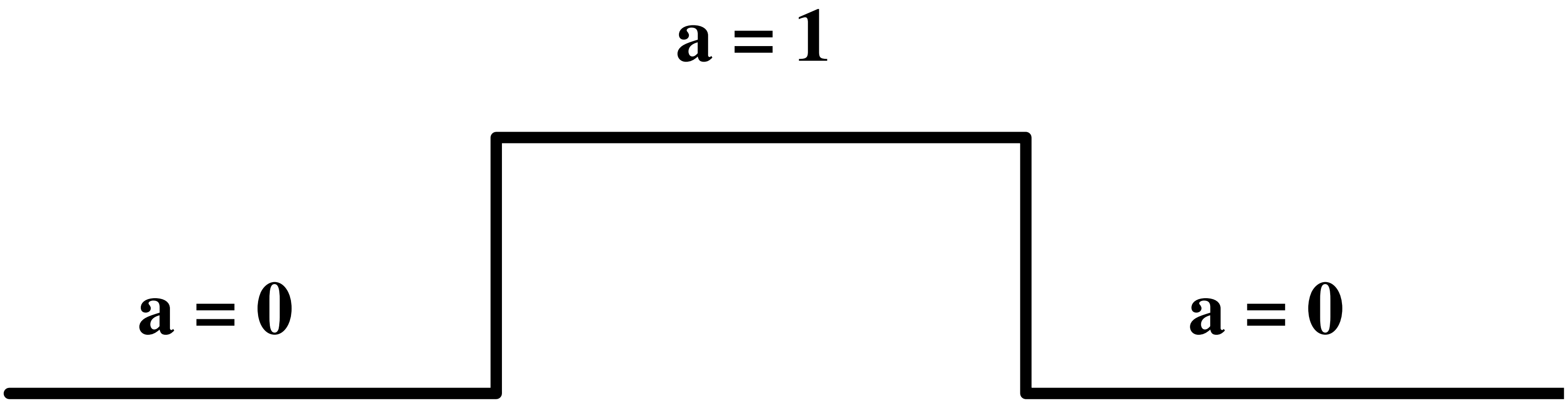
pat0_5: $cr \leq ca$

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

Model for weak action and reaction: the Events (1)

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

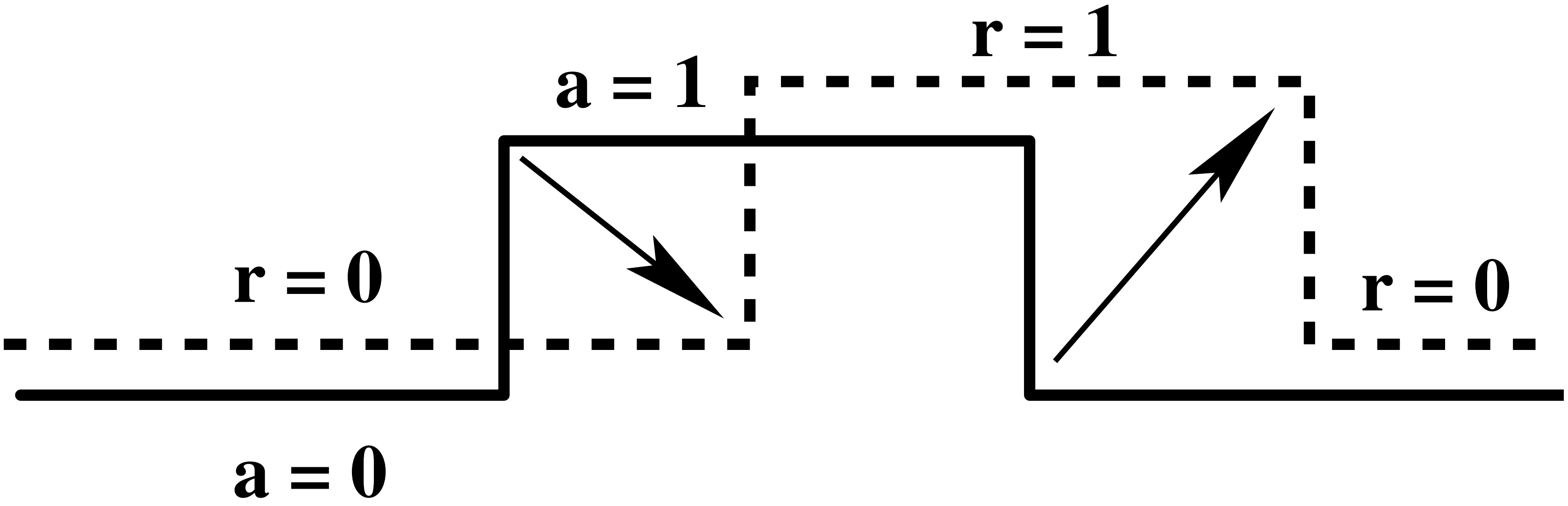
```
a_off
   when
      $a = 1$
   then
      $a := 0$
   end
```



Model for weak action and reaction: the Events (2)

```
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
```



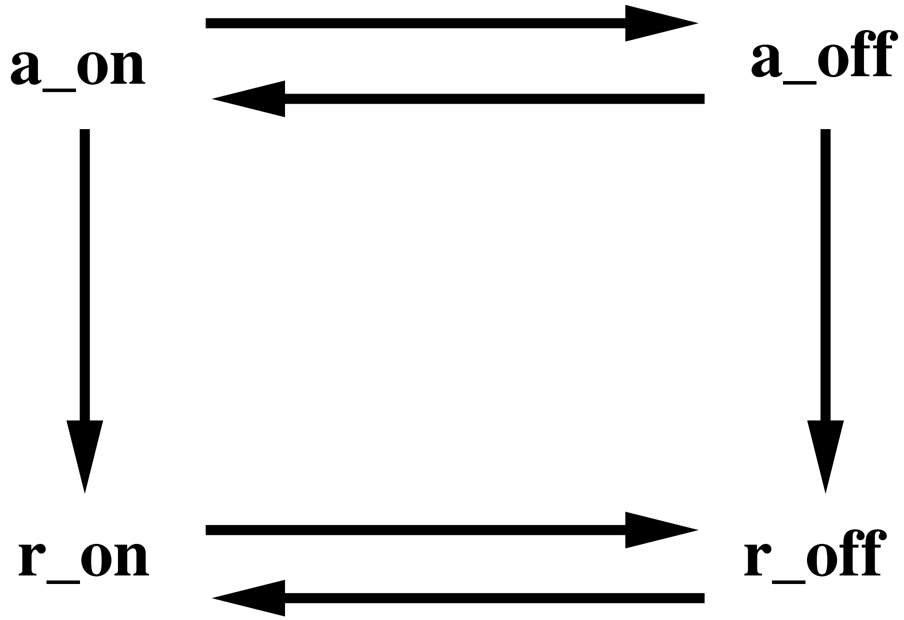
Summary of Events

```
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
```



Summary of Weak Synchronization

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: $cr \leq ca$

```
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
```

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

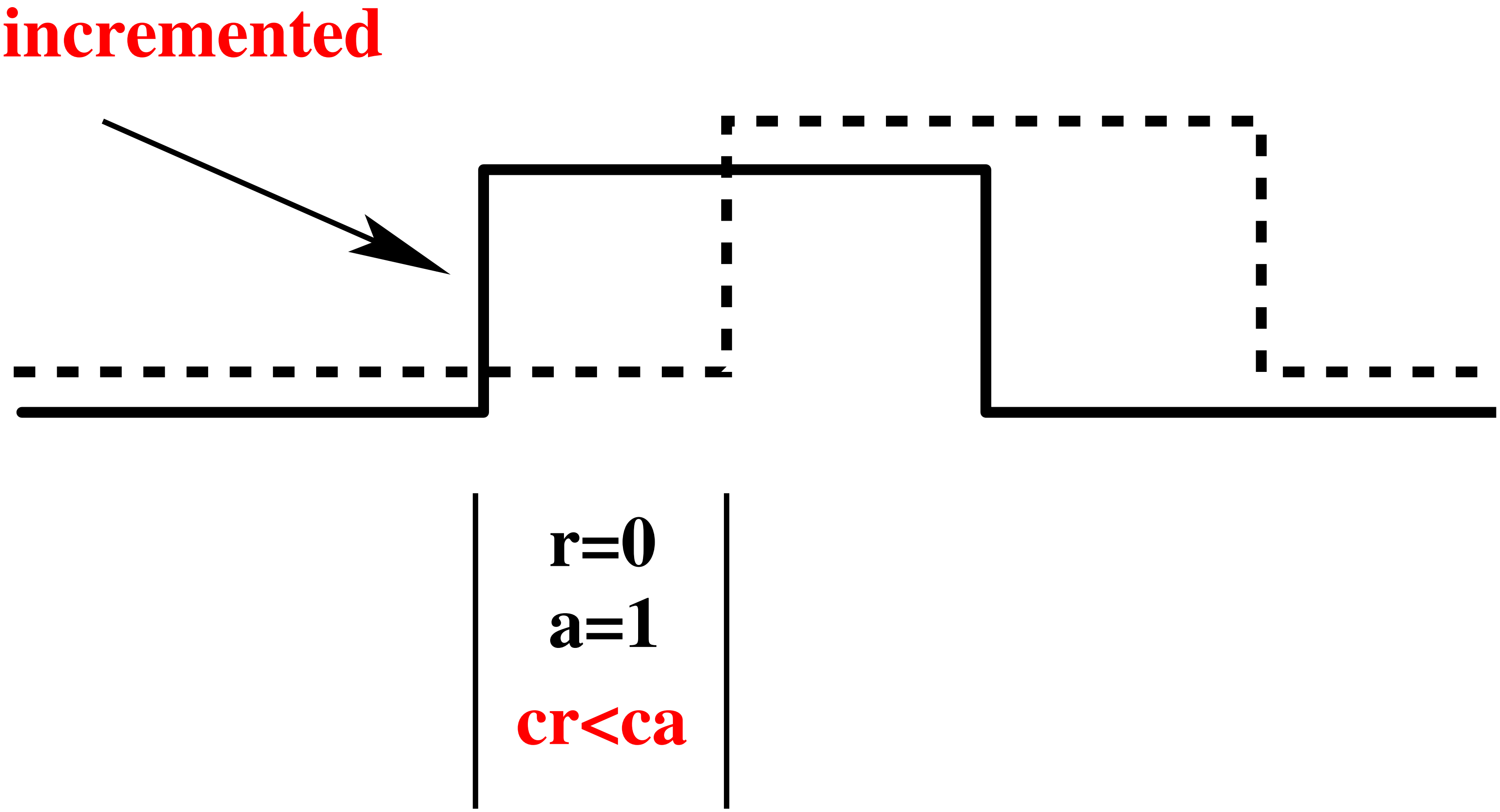
```
r_off
  when
    $r = 1$
    $a = 0$
  then
    $r := 0$
  end
```

Nothing guarantees that the invariants are preserved

D E M 0 (Showing a Problem and Finding a Solution)

Intuition Behind the new Invariant

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



Summary of the State of the weak Reaction

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: $cr \leq ca$

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

Summary of the Events of the weak Reaction

The counters have

been removed

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

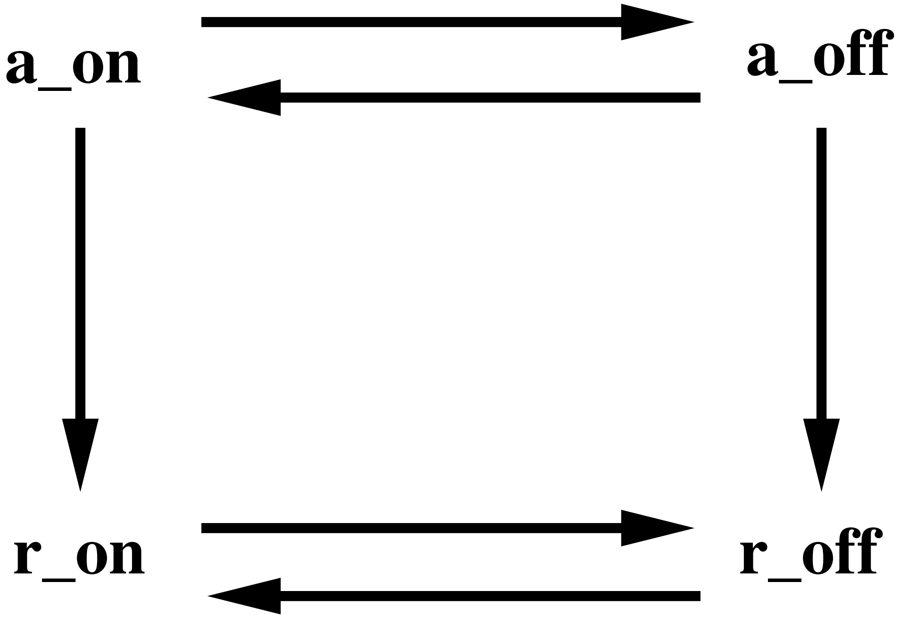
```
a_off
  when
    $a = 1$
  then
    $a := 0$
  end
```

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

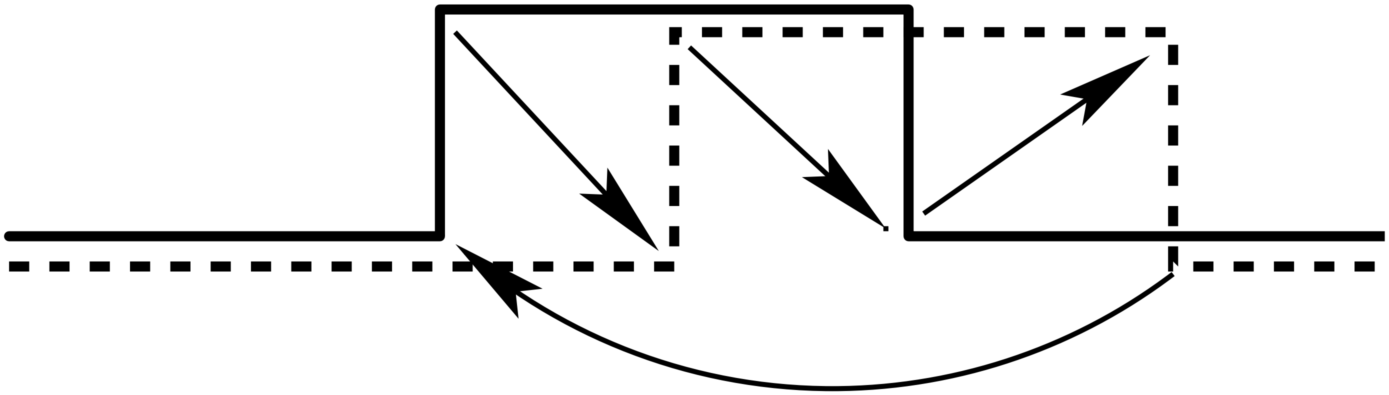
```
r_on
  when
    $r = 0$
    $a = 1$
  then
    $r := 1$
  end
```

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

Weak Synchronization of Events



Refinement: Strong action and reaction



- We add the following invariant

pat1_1: $ca \leq cr + 1$

- Remember invariant **pat0_5**

pat0_5: $cr \leq ca$

We have thus: $cr \leq ca \leq cr + 1$

Summary (so far) of the Strong Sncchronization

pat1_1: $ca \leq cr + 1$

```
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
```

Nothing guarantees that the invariant is preserved

D E M 0 (Showing Problems and Finding Solutions)

Merging the two invariants

- Putting together these two invariants

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

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

- leads to the following

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

Simplifying the Invariants

pat0_5: $cr \leq ca$

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

pat1_1: $ca \leq cr + 1$

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

This can be simplified to

pat2_1: $a = 1 \ \wedge \ r = 0 \ \Rightarrow \ ca = cr + 1$

pat2_2: $a = 0 \ \vee \ r = 1 \ \Rightarrow \ ca = cr$

Summary of the State for the Strong Reaction

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

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

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

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

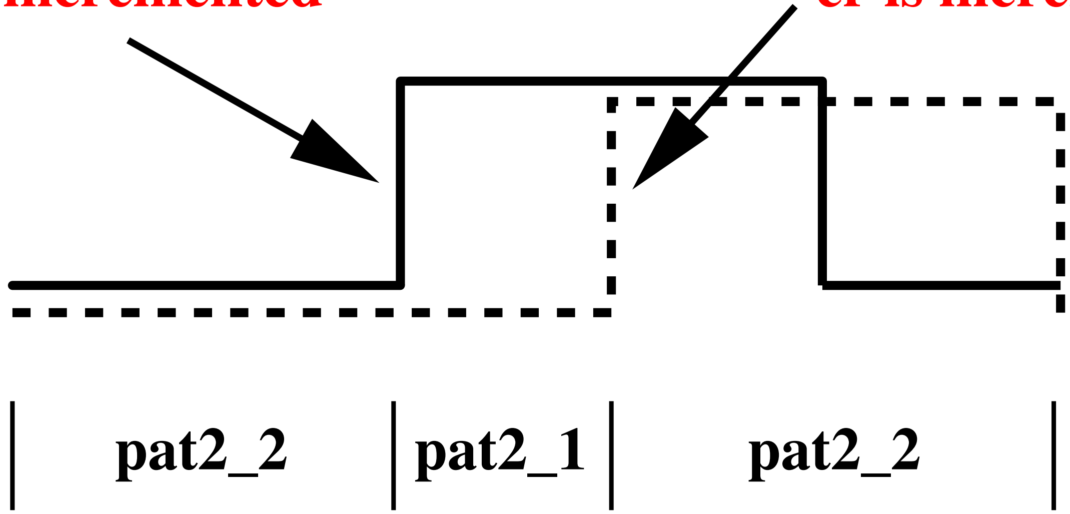
pat2_1: $a = 1 \wedge r = 0 \Rightarrow ca = cr + 1$

pat2_2: $a = 0 \vee r = 1 \Rightarrow ca = cr$

Intuition Behind the two Invariants

pat2_1: $a = 1 \land r = 0 \Rightarrow ca = cr + 1$

pat2_2: $a = 0 \lor r = 1 \Rightarrow ca = cr$



ca is incremented

cr is incremented

pat2_2	pat2_1	pat2_2
a=0	a=1	r=1
ca = cr	r=0	ca = cr
	ca=cr+1	

Summary of the Events for the Strong Reaction

The counters have

been removed

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

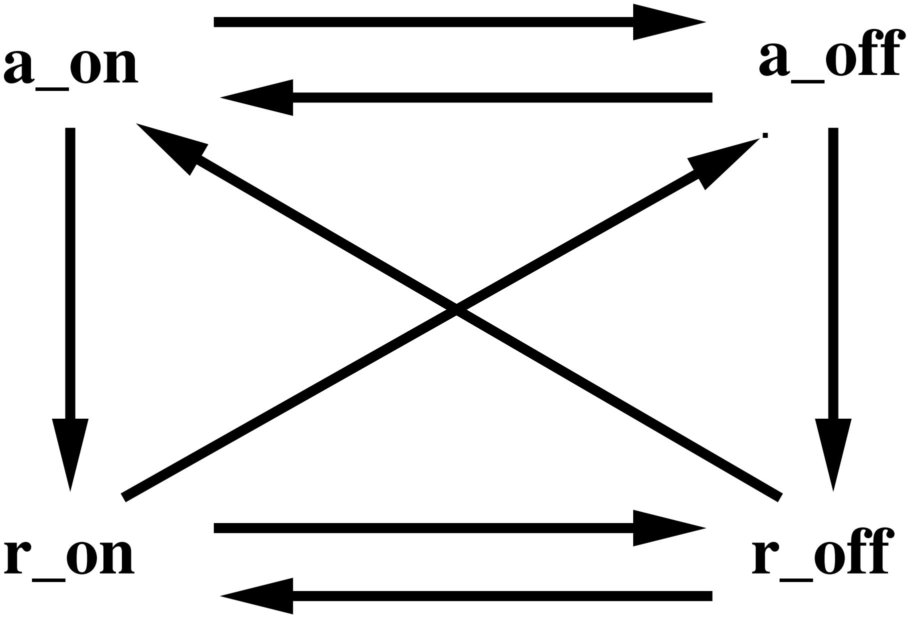
```
a_off
  when
    $a = 1$
    $r = 1$
  then
    $a := 0$
  end
```

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

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

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

Strong Synchronization of Events



What we Have Learned

- Proof failures helped us improving our models

- When an invariant preservation proof fails on an event,
  there are two solutions:
    - adding a new invariant
    - strengthening the guard

- Modelling considerations helped us choosing one or the other

- At the end, we reached a stable situation (fixpoint)

3. Writing the Requirement Document

Requirements: Describing Equipment

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

Requirements: Connection Constraints

Buttons and Controller are weakly synchronized	FUN_1

Controller are Equipment are strongly synchronized	FUN_2

Requirements: Relationship Between Motor and Clutch

When the clutch is engaged, the motor must work	SAF_1

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

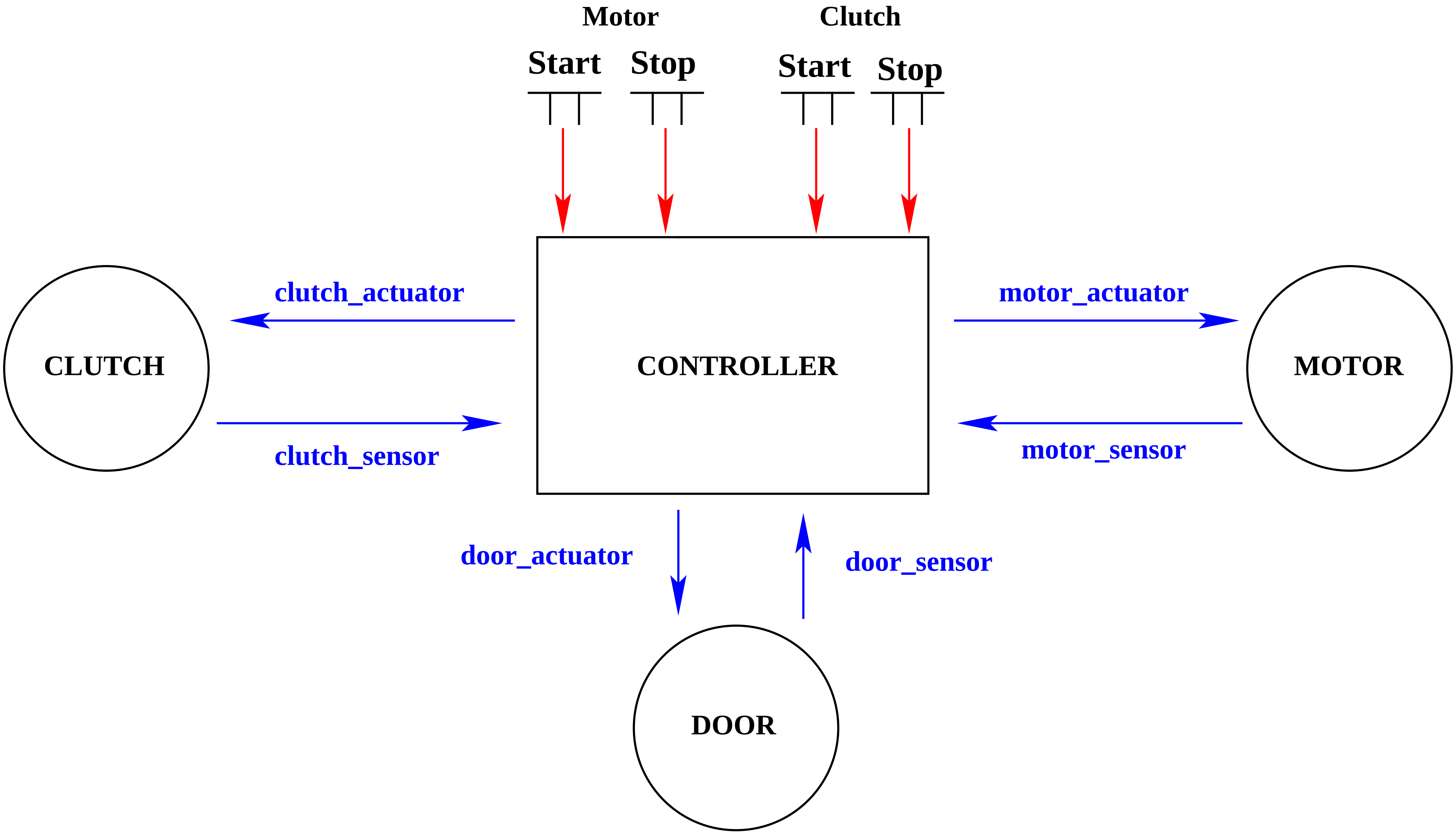
Requirements: Relationship Between Door and Clutch

When the clutch is engaged, 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 are not independent. It must be synchronized with disengaging and engaging the clutch	FUN_5

Overview


Motor
Start
Stop
Clutch
Start
Stop
CLUTCH
clutch_actuator
clutch_sensor
CONTROLLER
motor_actuator
motor_sensor
MOTOR
door_actuator
door_sensor
DOOR

4. Proposing a Refinement Strategy

Refinement Strategy

- 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

Refinement Strategy (cont’d)

- 4th refinement: Connecting the controller to the door

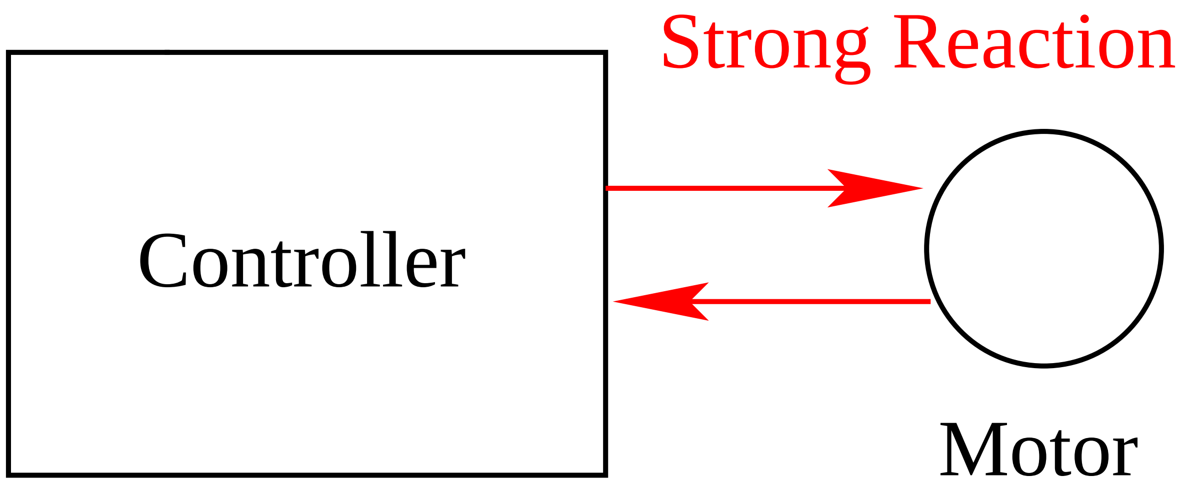
- 5th refinement: Constraining the clutch and the door

- 6th refinement: More constraints between clutch and door

- 7th refinement: Connecting the clutch buttons to the controller

5. Development of the Model using Refinements and Design Patterns

Initial Model: Connecting the Controller to the Motor



Controller are Equipment are strongly synchronized	FUN_2

Model for strong action and reaction: the Final Events

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
```

Initial Model: the Context

set: $STATUS$

constants: $stopped$
$working$

axm0_1: $STATUS = \{stopped, working\}$

axm0_2: $stopped \neq working$

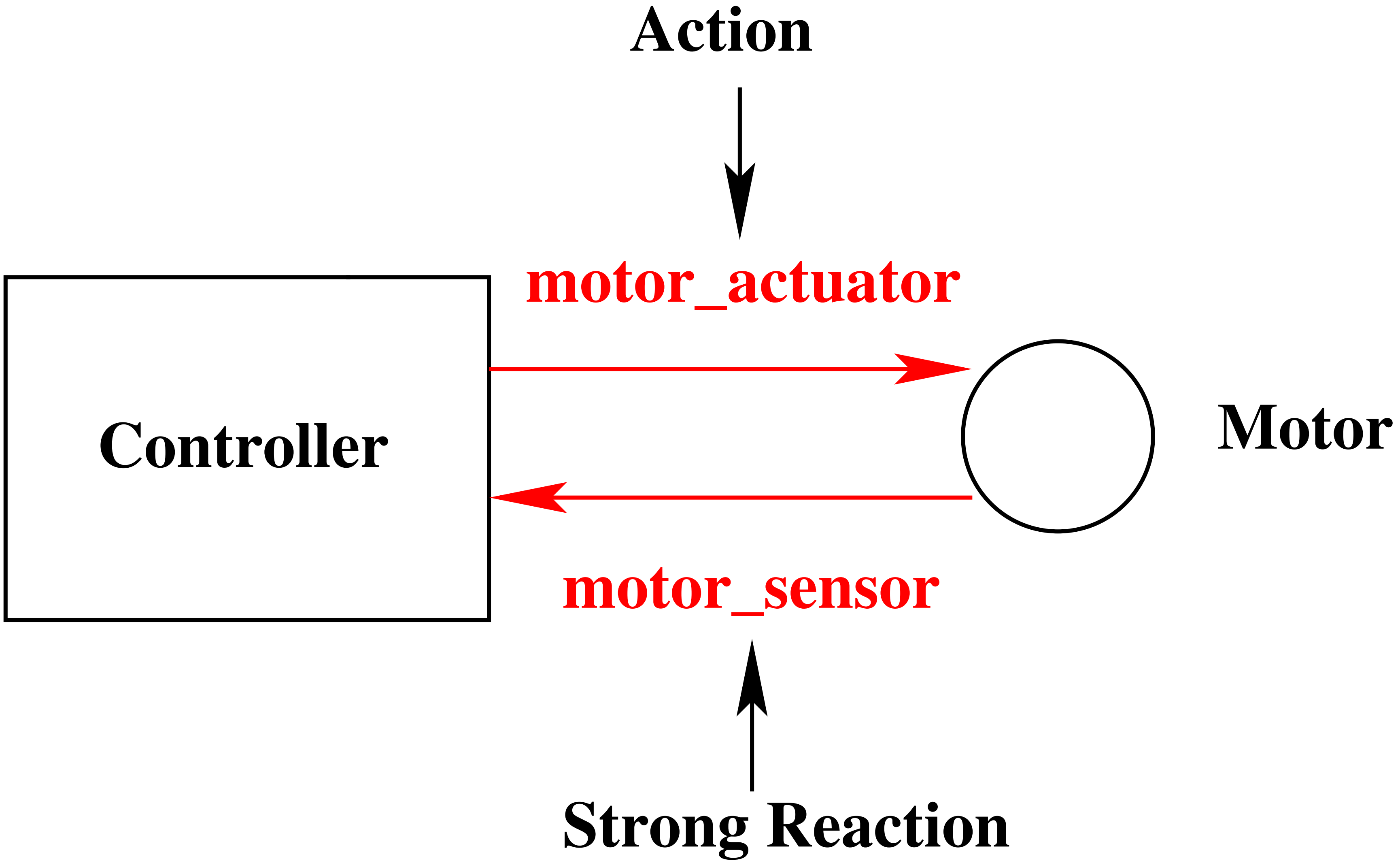
Initial Model: the State

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

inv0_1: $motor\_sensor \in STATUS$

inv0_2: $motor\_actuator \in STATUS$

Initial Model: the Synchronization



Pattern Instantiation

- We instantiate the weak pattern as follows:

$$\begin{array}{lcl} \boldsymbol{a} & \rightsquigarrow & \boldsymbol{motor\_actuator} \\ \boldsymbol{r} & \rightsquigarrow & \boldsymbol{motor\_sensor} \\ \mathbf{0} & \rightsquigarrow & \boldsymbol{stopped} \\ \mathbf{1} & \rightsquigarrow & \boldsymbol{working} \end{array}$$

a_on ⇝ treat_start_motor
a_off ⇝ treat_stop_motor
r_on ⇝ Motor_start
r_off ⇝ Motor_stop

- Convention: Controller events start with "treat_"

Initial Model: Initialization

init

$a := 0$

$r := 0$

init

$motor\_actuator := stopped$

$motor\_sensor := stopped$

Initial Model: Controller Events (1)

```
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
```

Initial Model: Environment Event (1)

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

Initial Model: Controller Events (2)

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

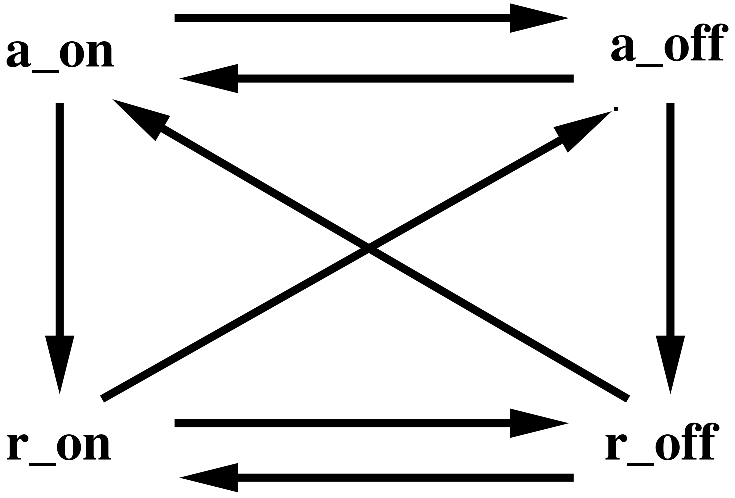
```
treat_stop_motor
    when
        $motor\_actuator = working$
        $motor\_sensor = working$
    then
        $motor\_actuator := stopped$
    end
```

Initial Model: Environment Event (2)

```
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
```

Synchronization

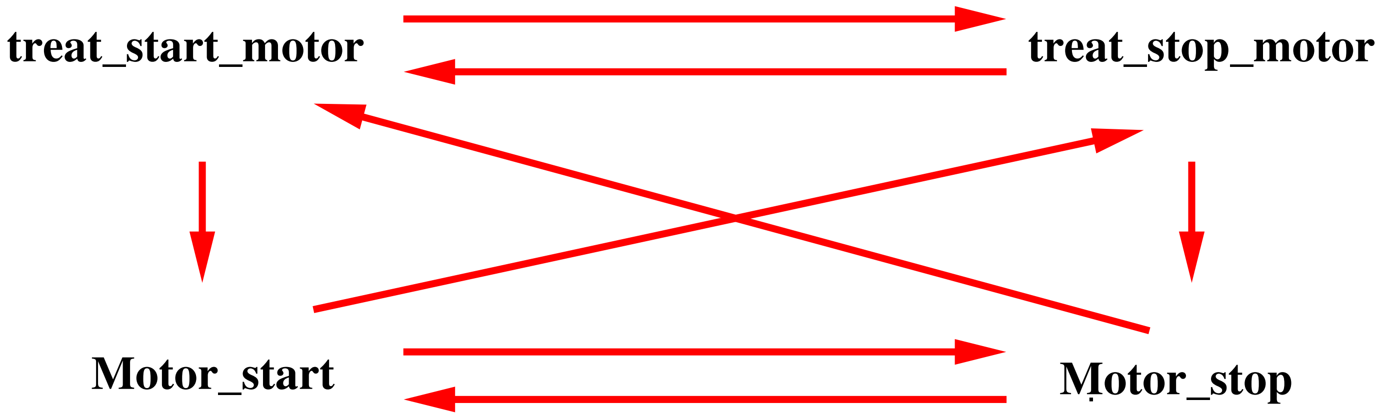


a_on

a_off

r_on

r_off



treat_start_motor

treat_stop_motor

Motor_start

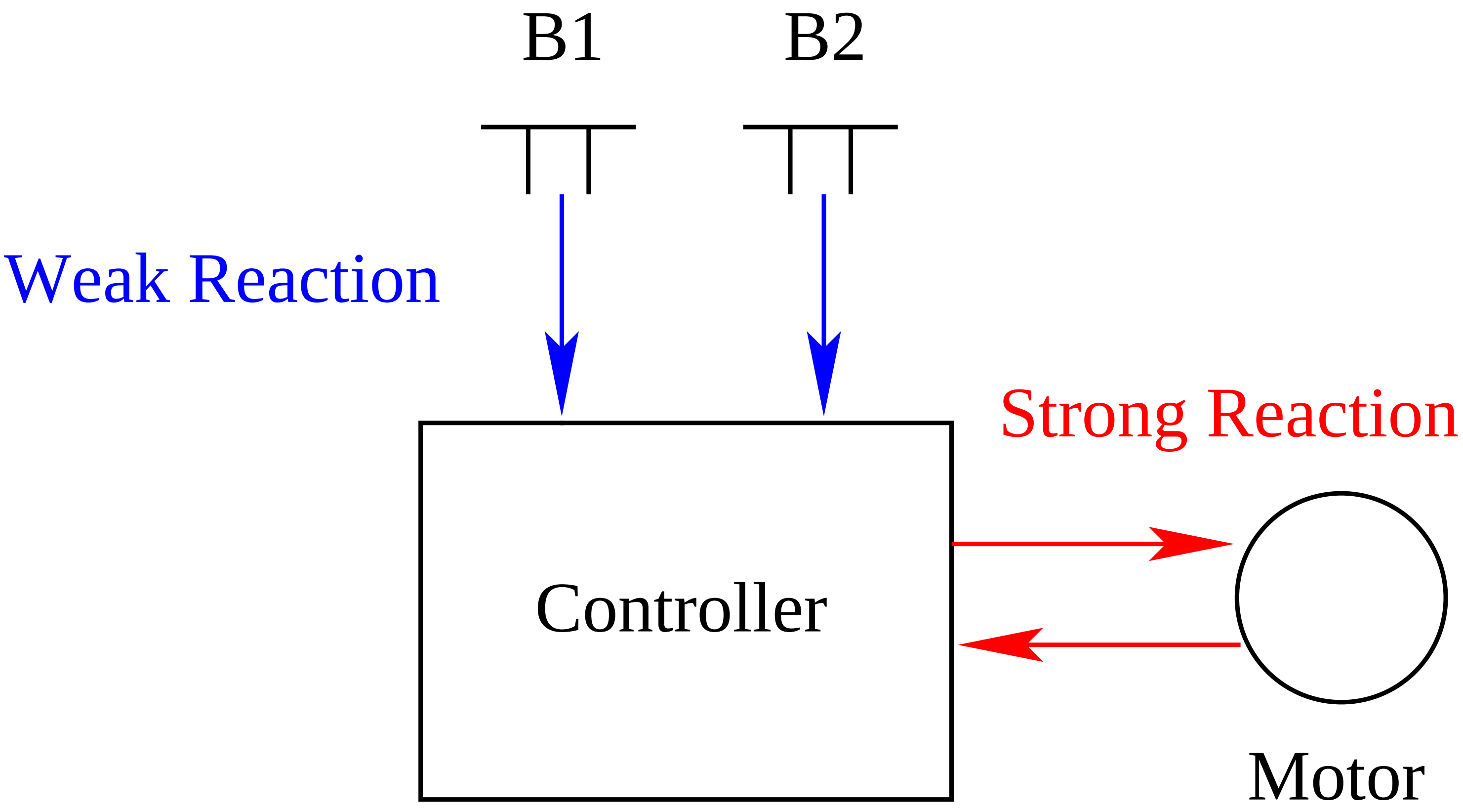
Motor_stop

Initial Model: Summary of the Events

- Environment
  - motor_start
  - motor_stop

- Controller
  - treat_start_motor
  - treat_stop_motor

1st Reft.: Connecting the Motor Buttons to the Controller



Buttons and Controller are weakly synchronized	FUN_1

Model for weak action and reaction: the Final Events

The counters have

been removed

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

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

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

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

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

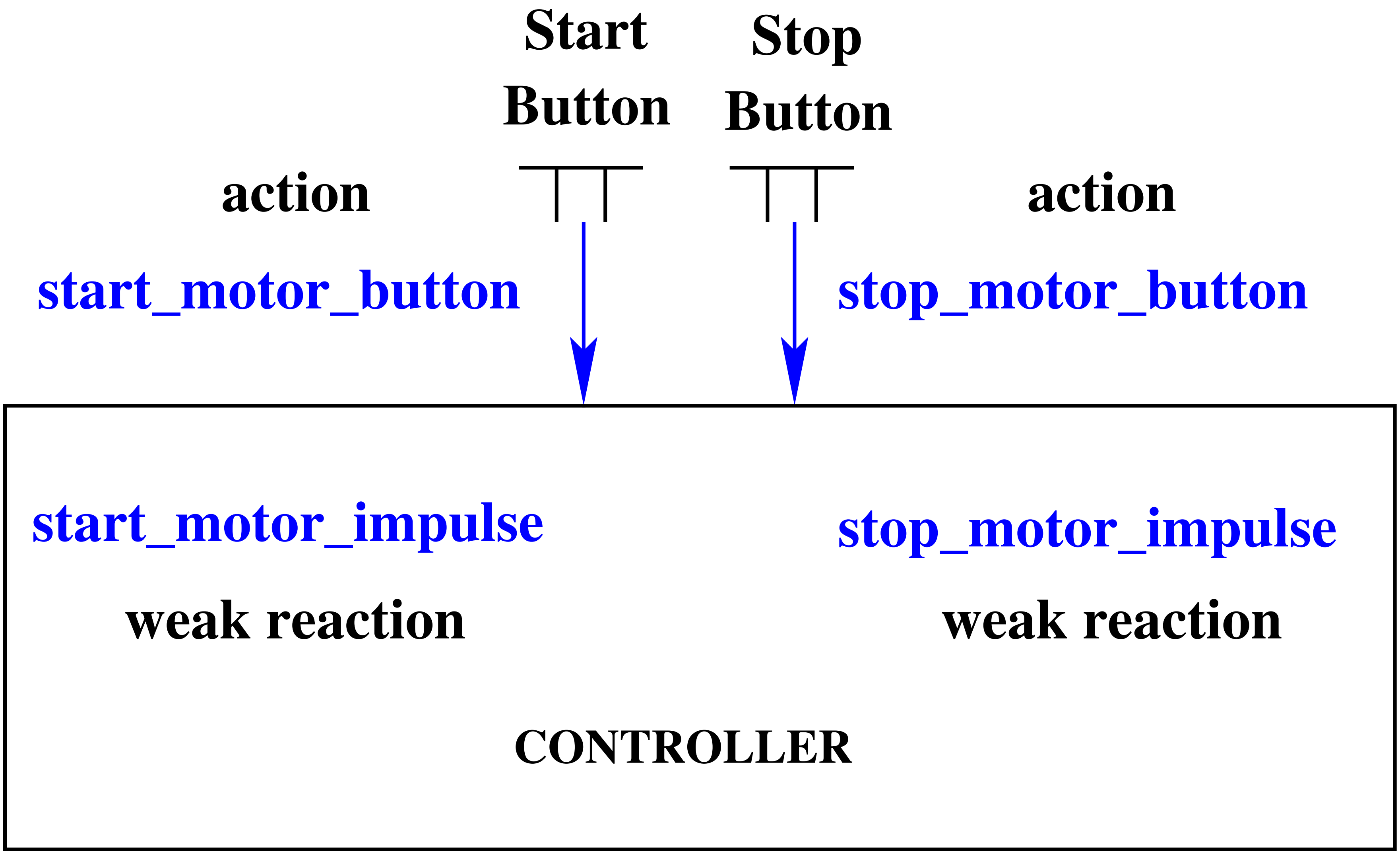
First Refinement: the State

variables:

$\dots$
$start\_motor\_button$
$stop\_motor\_button$
$start\_motor\_impulse$
$stop\_motor\_impulse$

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

First Refinement: the State



Start Button

Stop Button

action

action

start_motor_button

stop_motor_button

start_motor_impulse

stop_motor_impulse

weak reaction

weak reaction

CONTROLLER

Pattern Instantiation

- We instantiate the pattern as follows:

$$\begin{array}{lll} \boldsymbol{a} & \rightsquigarrow & \boldsymbol{start\_motor\_button} \\ \boldsymbol{r} & \rightsquigarrow & \boldsymbol{start\_motor\_impulse} \\ \mathbf{0} & \rightsquigarrow & \mathbf{FALSE} \\ \mathbf{1} & \rightsquigarrow & \mathbf{TRUE} \end{array}$$

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

1st Refinement: Refinement of Initialization

init

$a := 0$

$r := 0$

init

$motor\_actuator := stopped$

$motor\_sensor := stopped$

$start\_motor\_button := \mathrm{FALSE}$

$start\_motor\_impulse := \mathrm{FALSE}$

First Refinement: New Environment Events (1)

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

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

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

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

First Refinement: Refining Controller Events (1)

```
r_on

  when
    $r = 0$
    $a = 1$

  then
    $r := 1$

  end
```

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

- This is the most important slide of the talk

- We can see how patterns can be superposed

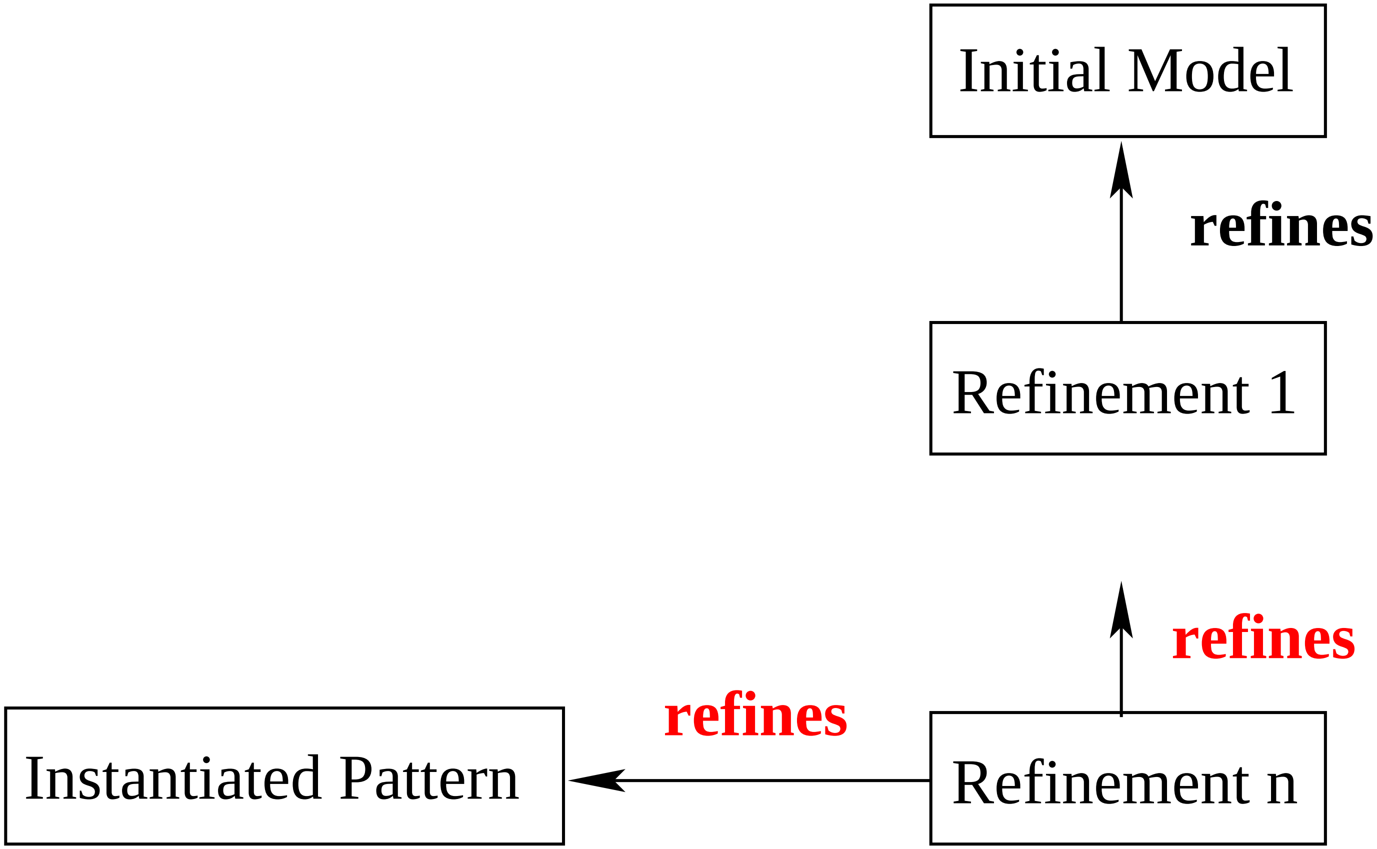
```
a_on
  when
    $a = 0$
    $r = 0$
  then
    $a := 1$
  end
```

```
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
    $start\_motor\_impulse = \mathrm{FALSE}$
    $start\_motor\_button = \mathrm{TRUE}$
    $motor\_actuator = stopped$
    $motor\_sensor = stopped$
  then
    $start\_motor\_impulse := \mathrm{TRUE}$
    $motor\_actuator := working$
  end
```

Design Pattern Integration within a Development



Initial Model

refines

Refinement 1

refines

refines

Instantiated Pattern

Refinement n

First Refinement: New Controller Events (1)

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

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

Pattern Instantiation

- We instantiate the pattern as follows:

$$\begin{array}{lcl}
\boldsymbol{a} & \rightsquigarrow & \boldsymbol{stop\_motor\_button} \\
\boldsymbol{r} & \rightsquigarrow & \boldsymbol{stop\_motor\_impulse} \\
\mathbf{0} & \rightsquigarrow & \mathbf{FALSE} \\
\mathbf{1} & \rightsquigarrow & \mathbf{TRUE}
\end{array}$$

a_on ⇝ push_stop_motor_button
a_off ⇝ release_stop_motor_button
r_on ⇝ treat_push_stop_motor_button
r_off ⇝ treat_release_stop_motor_button

1st Refinement: More Refinement of Initialization

init

$a := 0$

$r := 0$

init

$motor\_actuator := stopped$

$motor\_sensor := stopped$

$start\_motor\_button := \mathrm{FALSE}$

$start\_motor\_impulse := \mathrm{FALSE}$

$stop\_motor\_button := \mathrm{FALSE}$

$stop\_motor\_impulse := \mathrm{FALSE}$

First Refinement: New Environment Events

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

```
push_stop_motor_button
  when
    $stop\_motor\_button = \mathrm{FALSE}$
  then
    $stop\_motor\_button := \mathrm{TRUE}$
  end
```

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

```
release_stop_motor_button
  when
    $stop\_motor\_button = \mathrm{TRUE}$
  then
    $stop\_motor\_button := \mathrm{FALSE}$
  end
```

First Refinement: Refining Controller Events (2)

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

```
treat_push_stop_motor_button
    refines
        treat_stop_motor
    when
        $stop\_motor\_impulse = \mathrm{FALSE}$
        $stop\_motor\_button = \mathrm{TRUE}$
        $motor\_sensor = working$
        $motor\_actuator = working$
    then
        $stop\_motor\_impulse := \mathrm{TRUE}$
        $motor\_actuator := stopped$
    end
```

First Refinement: New Controller Events (2)

r_off

when

$r = 1$

$a = 0$

then

$r := 0$

end

treat_release_stop_motor_button

when

$stop\_motor\_impulse = \mathrm{TRUE}$

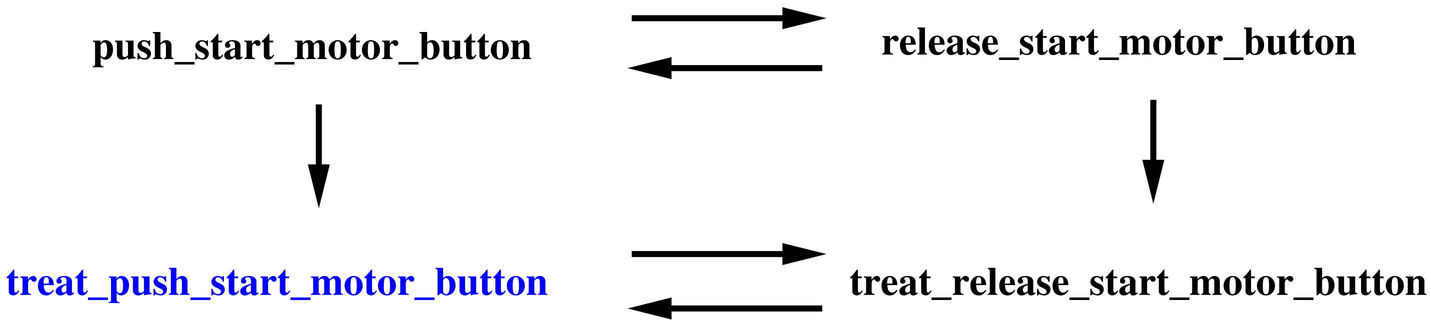
$stop\_motor\_button = \mathrm{FALSE}$

then

$stop\_motor\_impulse := \mathrm{FALSE}$

end

Independent Synchronizations



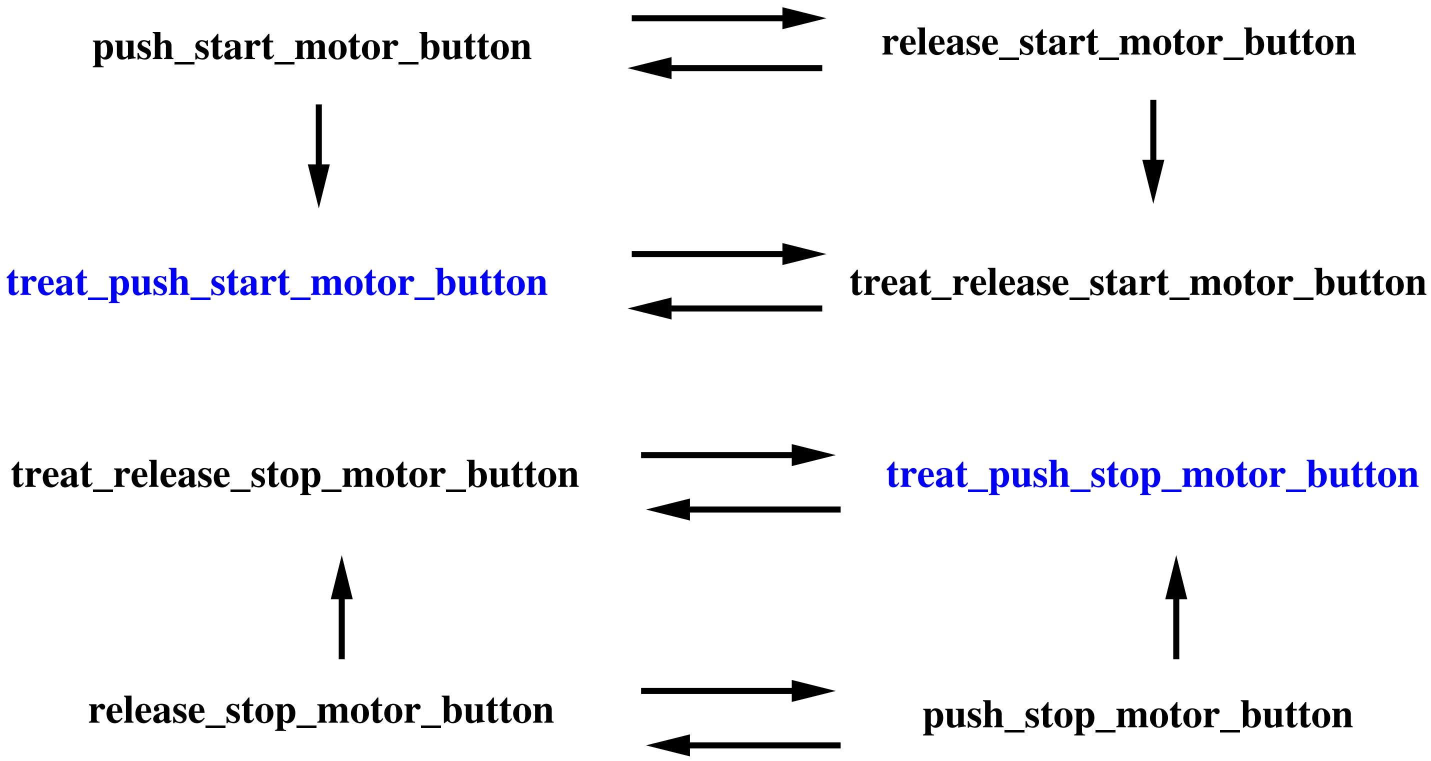
push_start_motor_button

release_start_motor_button

treat_push_start_motor_button

treat_release_start_motor_button

Independent Synchronizations



push_start_motor_button

release_start_motor_button

treat_push_start_motor_button

treat_release_start_motor_button

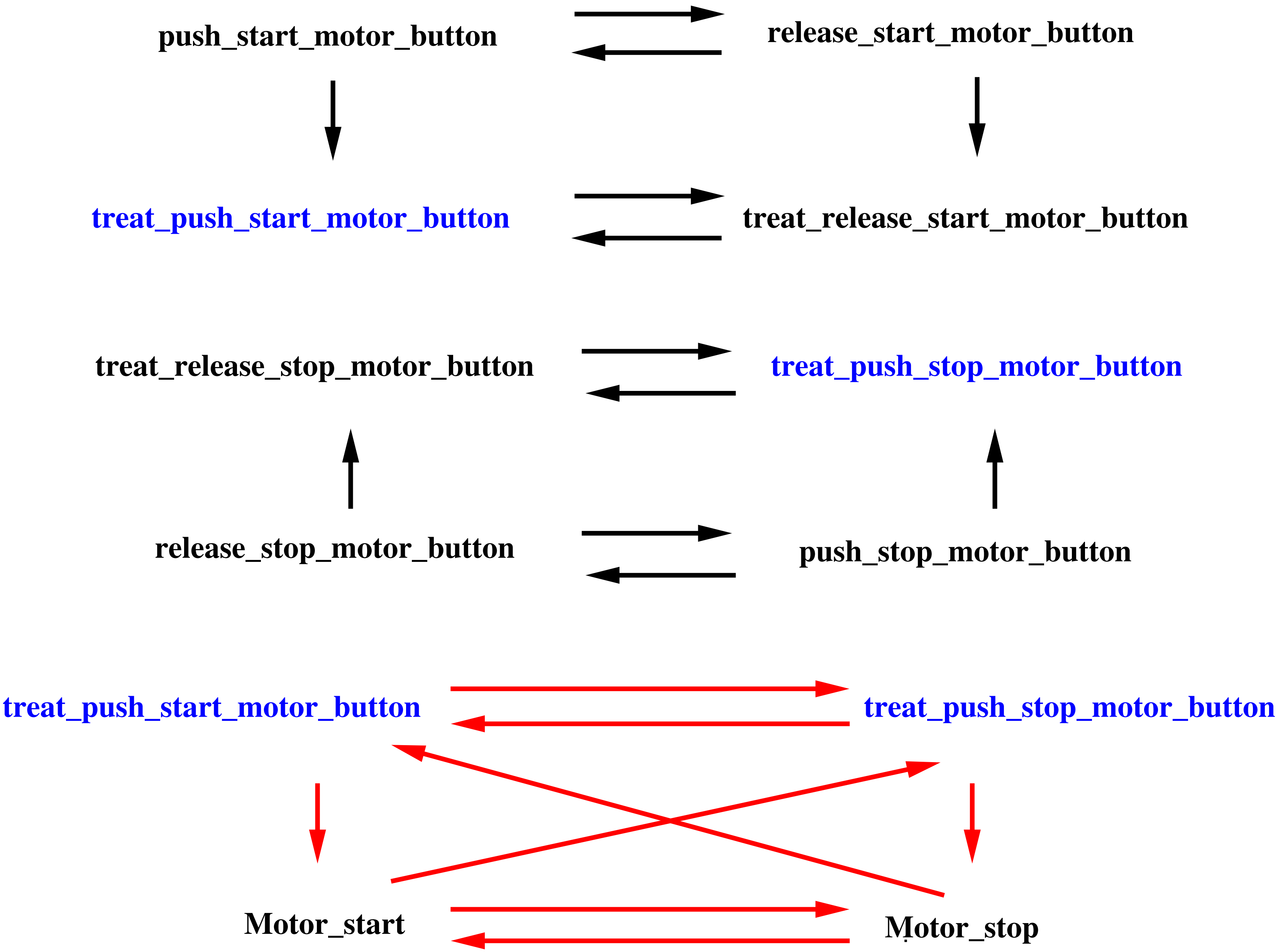
treat_release_stop_motor_button

treat_push_stop_motor_button

release_stop_motor_button

push_stop_motor_button

Independent Synchronizations



push_start_motor_button

release_start_motor_button

treat_push_start_motor_button

treat_release_start_motor_button

treat_release_stop_motor_button

treat_push_stop_motor_button

release_stop_motor_button

push_stop_motor_button

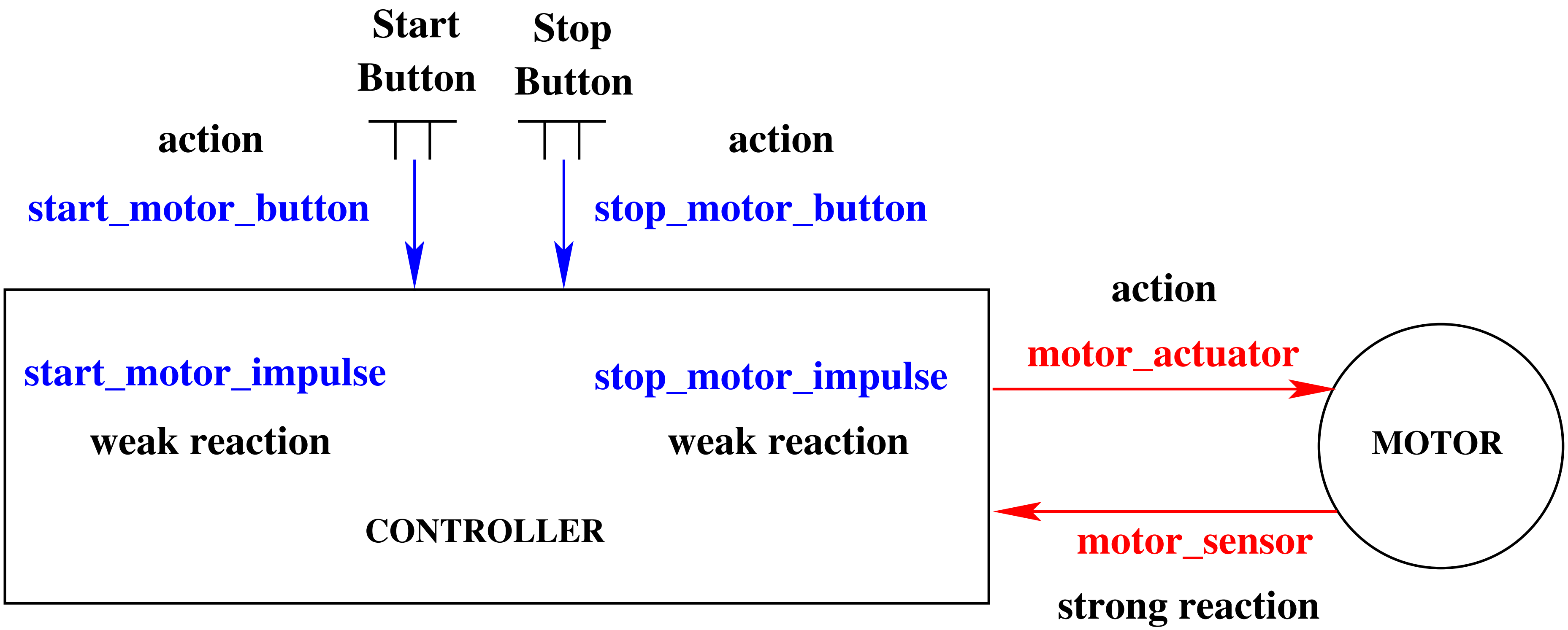
treat_push_start_motor_button

treat_push_stop_motor_button

Motor_start

Motor_stop

Weak and Strong Reactions Together



Start Button

Stop Button

action

start_motor_button

stop_motor_button

action

action

motor_actuator

start_motor_impulse

stop_motor_impulse

weak reaction

weak reaction

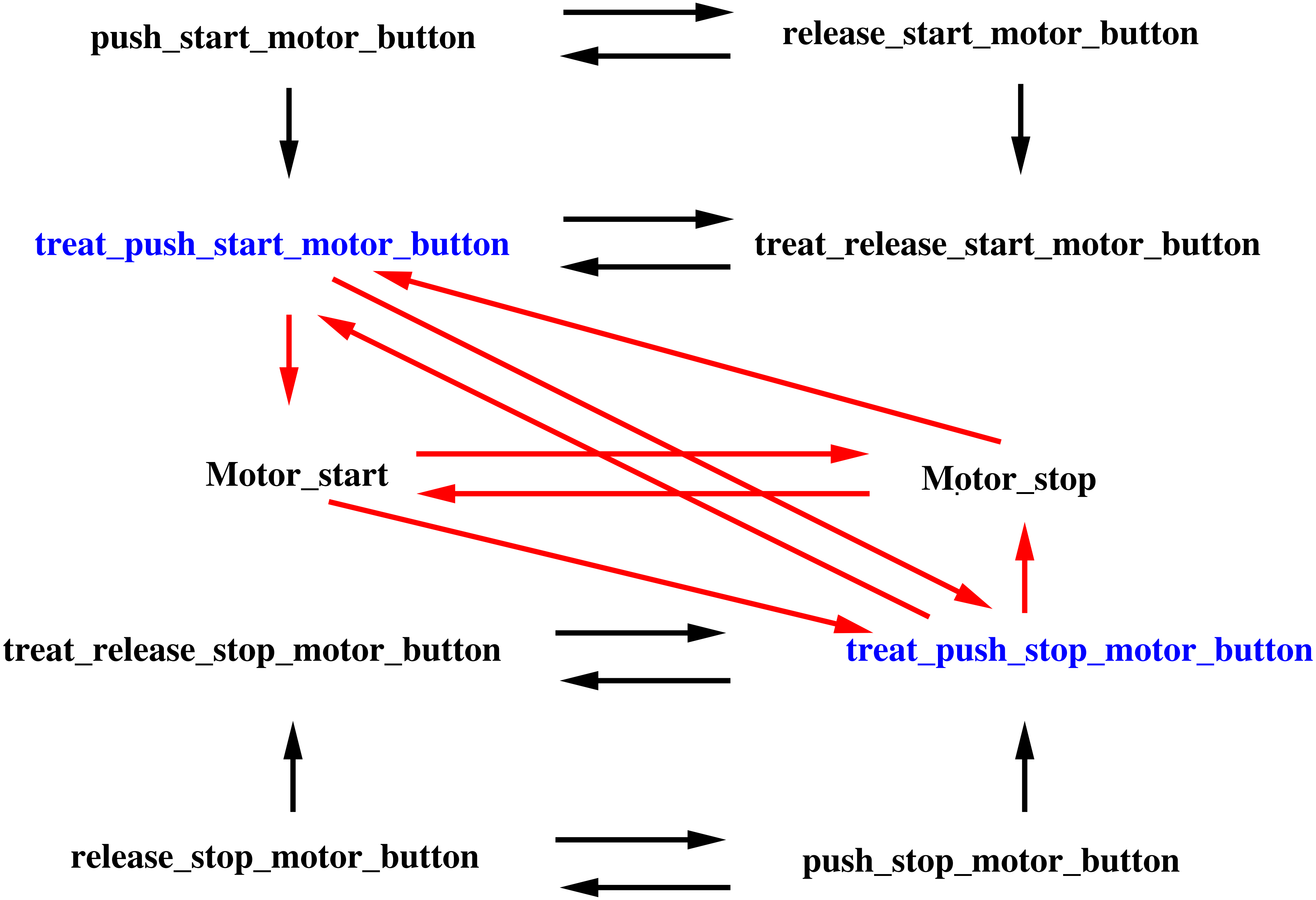
MOTOR

CONTROLLER

motor_sensor

strong reaction

Combined Synchronizations



push_start_motor_button

release_start_motor_button

treat_push_start_motor_button

treat_release_start_motor_button

Motor_start

Motor_stop

treat_release_stop_motor_button

treat_push_stop_motor_button

release_stop_motor_button

push_stop_motor_button

Problems with treat_push_start_motor_button

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

- What happens when the following hold

$$\neg\,(motor\_actuator = stopped \quad \wedge \quad motor\_sensor = stopped)$$

- We need another event

Problems with treat_push_start_motor_button

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

```
treat_push_start_motor_button_false

  when
    $start\_motor\_impulse = \mathrm{FALSE}$
    $start\_motor\_button = \mathrm{TRUE}$
    $\neg\,(motor\_actuator = stopped \;\wedge$
       $motor\_sensor = stopped)$
  then
    $start\_motor\_impulse := \mathrm{TRUE}$

  end
```

- In the second case, the button has been pushed but the internal conditions are not met

- However, we need to record that the button has been pushed:

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

Problems with treat_push_stop_motor_button

```
treat_push_stop_motor_button
  refines
    treat_stop_motor
  when
    $stop\_motor\_impulse = \mathrm{FALSE}$
    $stop\_motor\_button = \mathrm{TRUE}$
    $motor\_sensor = working$
    $motor\_actuator = working$
  then
    $stop\_motor\_impulse := \mathrm{TRUE}$
    $motor\_actuator := stopped$
  end
```

```
treat_push_stop_motor_button_false

  when
    $stop\_motor\_impulse = \mathrm{FALSE}$
    $stop\_motor\_button = \mathrm{TRUE}$
    $\neg\,(motor\_sensor = working \;\wedge$
      $motor\_actuator = working)$
  then
    $stop\_motor\_impulse := \mathrm{TRUE}$

  end
```

- In the second case, the button has been pushed but the internal conditions are not met

- However, we need to record that the button has been pushed:

$$stop\_motor\_impulse := \mathrm{TRUE}$$

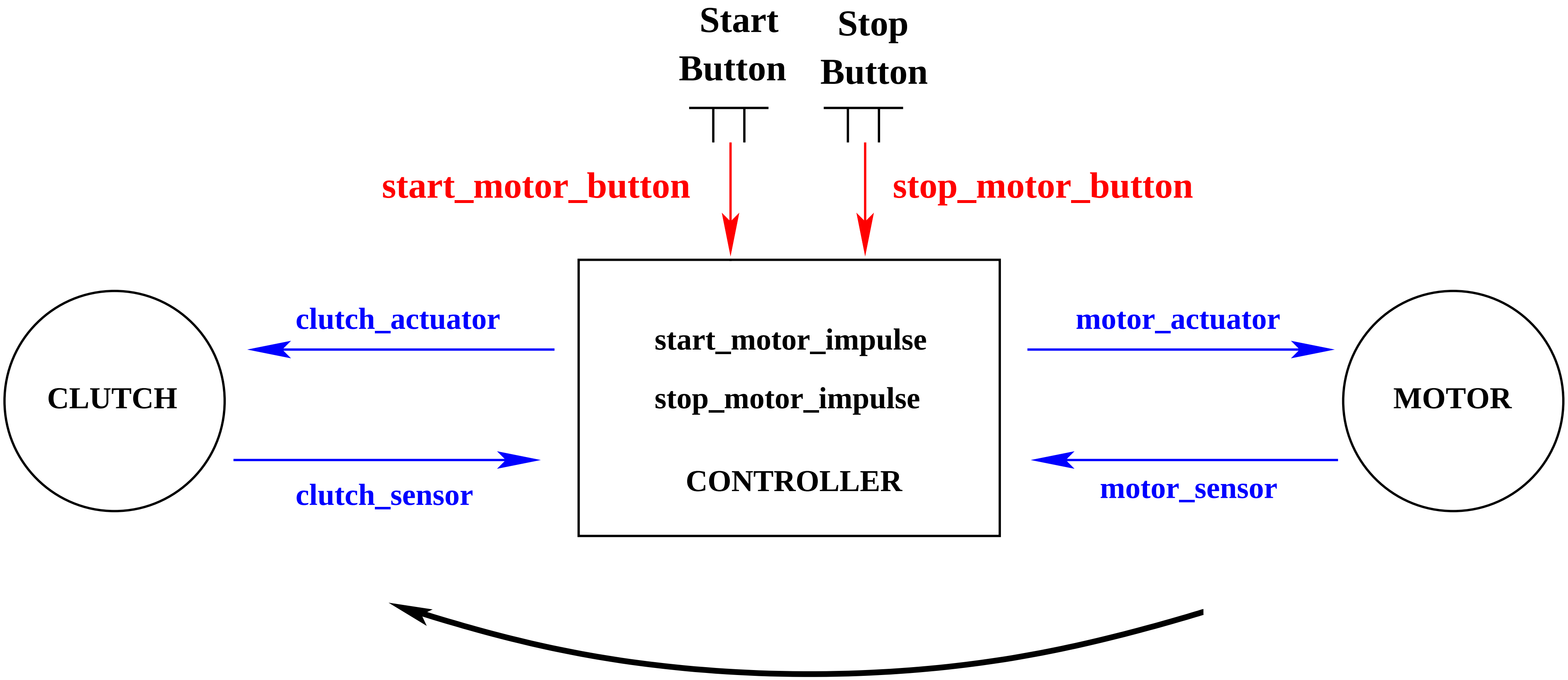
First Refinement: Summary of the Events (1)

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

First Refinement: Summary of the Events (2)

- 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

2nd Refinement: Connecting the Controller to the Clutch



2nd Refinement: Connecting the Controller to the Clutch

- We introduce the set in a new context:

$$CLUTCH = \{engaged, disengaged\}$$

- We copy the initial model where we instantiate:

$$\begin{array}{lcl} motor & \leadsto & clutch \\ STATUS & \leadsto & CLUTCH \\ working & \leadsto & engaged \\ stopped & \leadsto & disengaged \end{array}$$

Second Refinement: Summary of the Events (1)

- 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

Second Refinement: Summary of the Events (2)

- 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

Third Refinement: Constraining the Clutch and the Motor

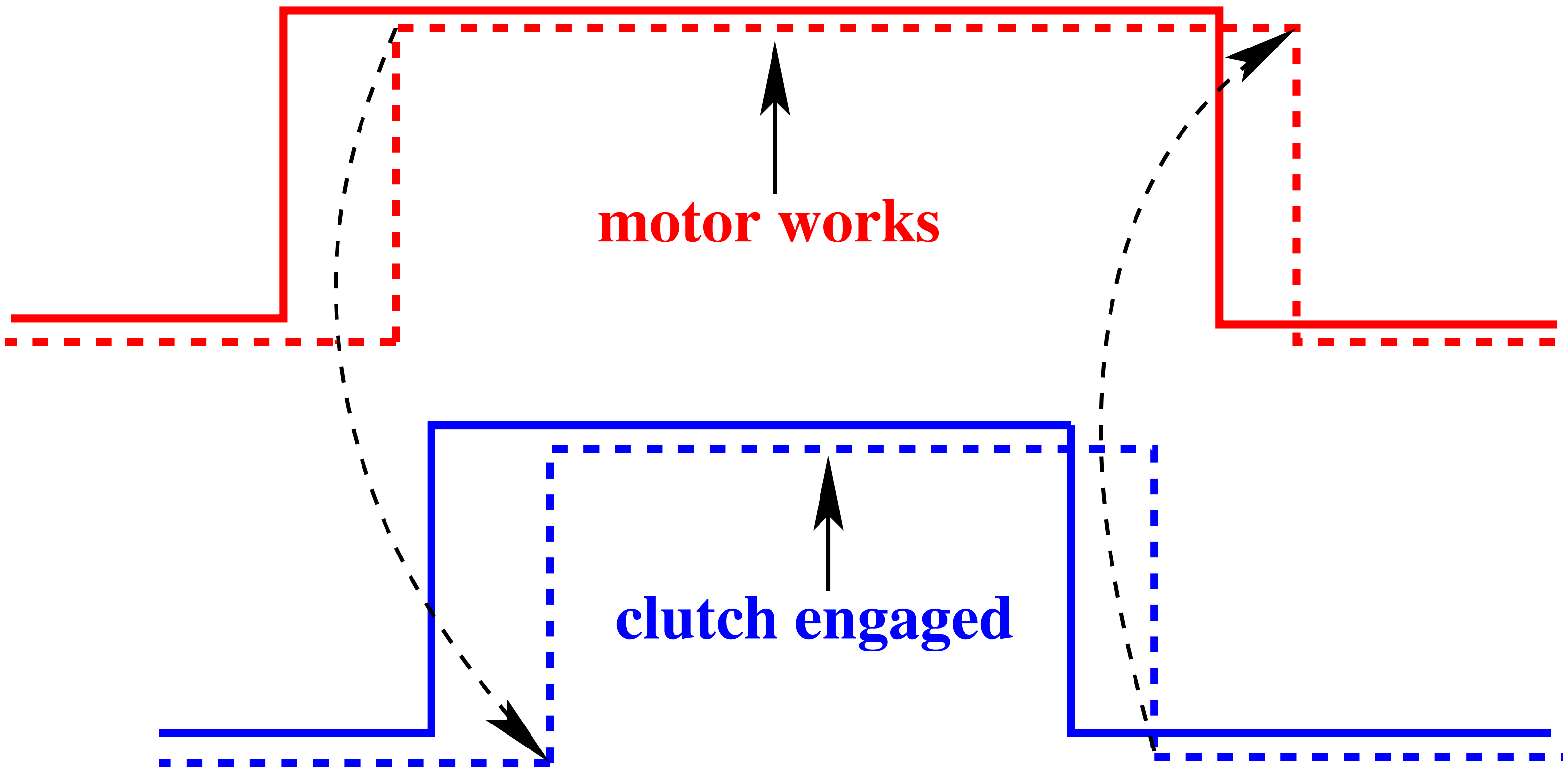
- An additional safety constraint

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

Pattern: Weak Synchronization of Strong Reactions

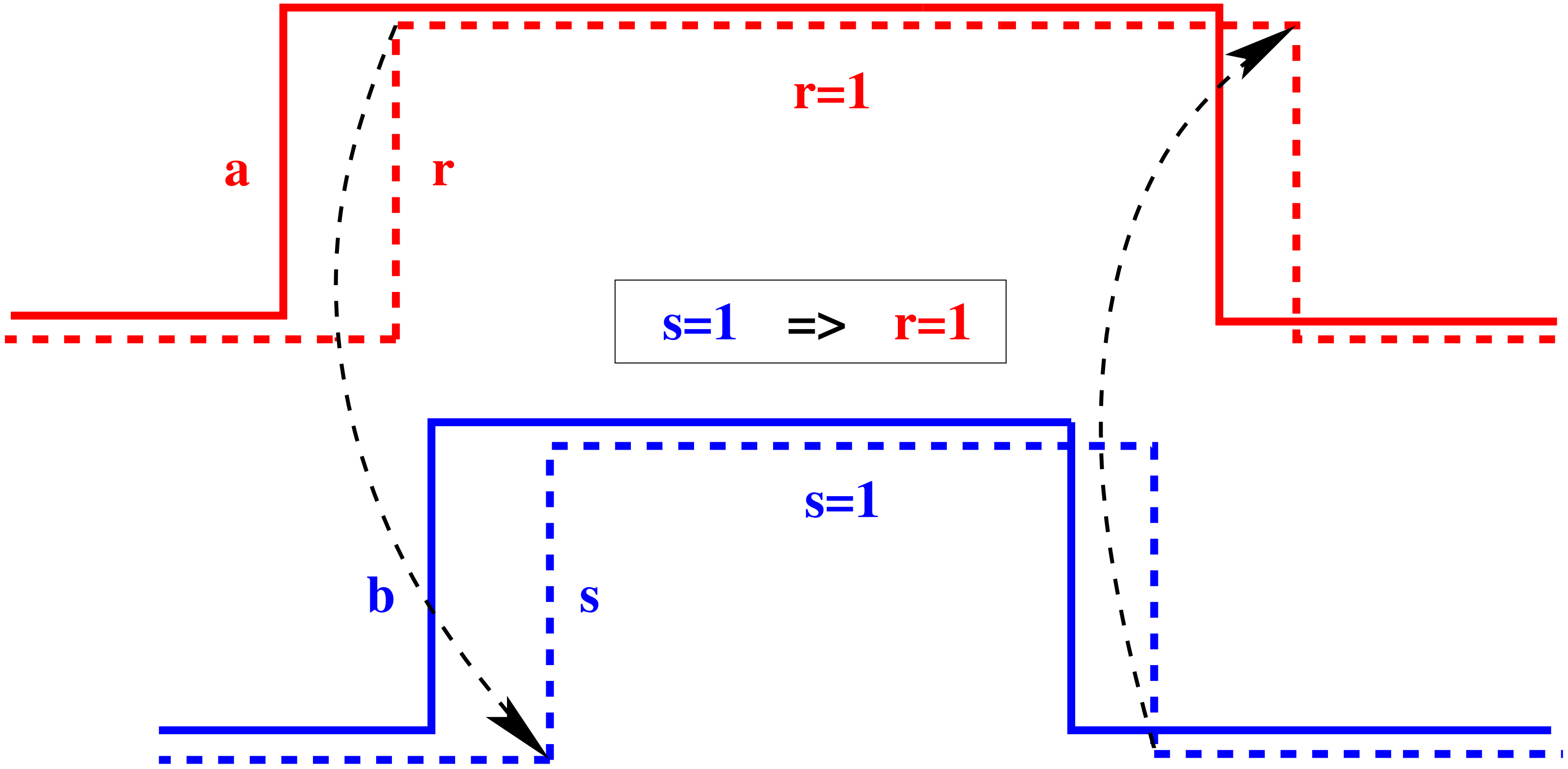


When the clutch is engaged

then

the motor must work

Pattern: Weak Synchronization of Strong Reactions

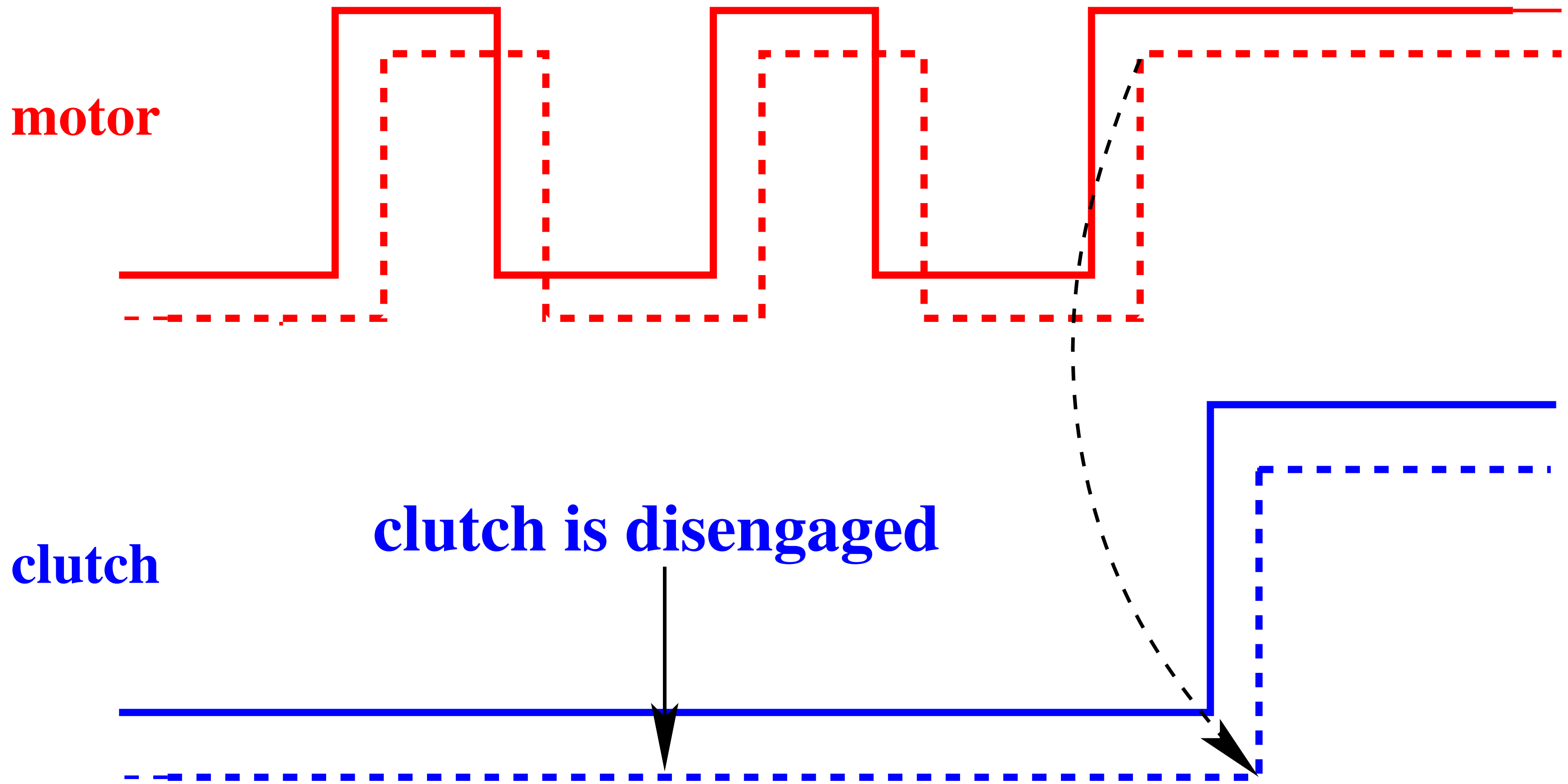


When the clutch is engaged

then

the motor must work

The Synchronization is Weak (1)

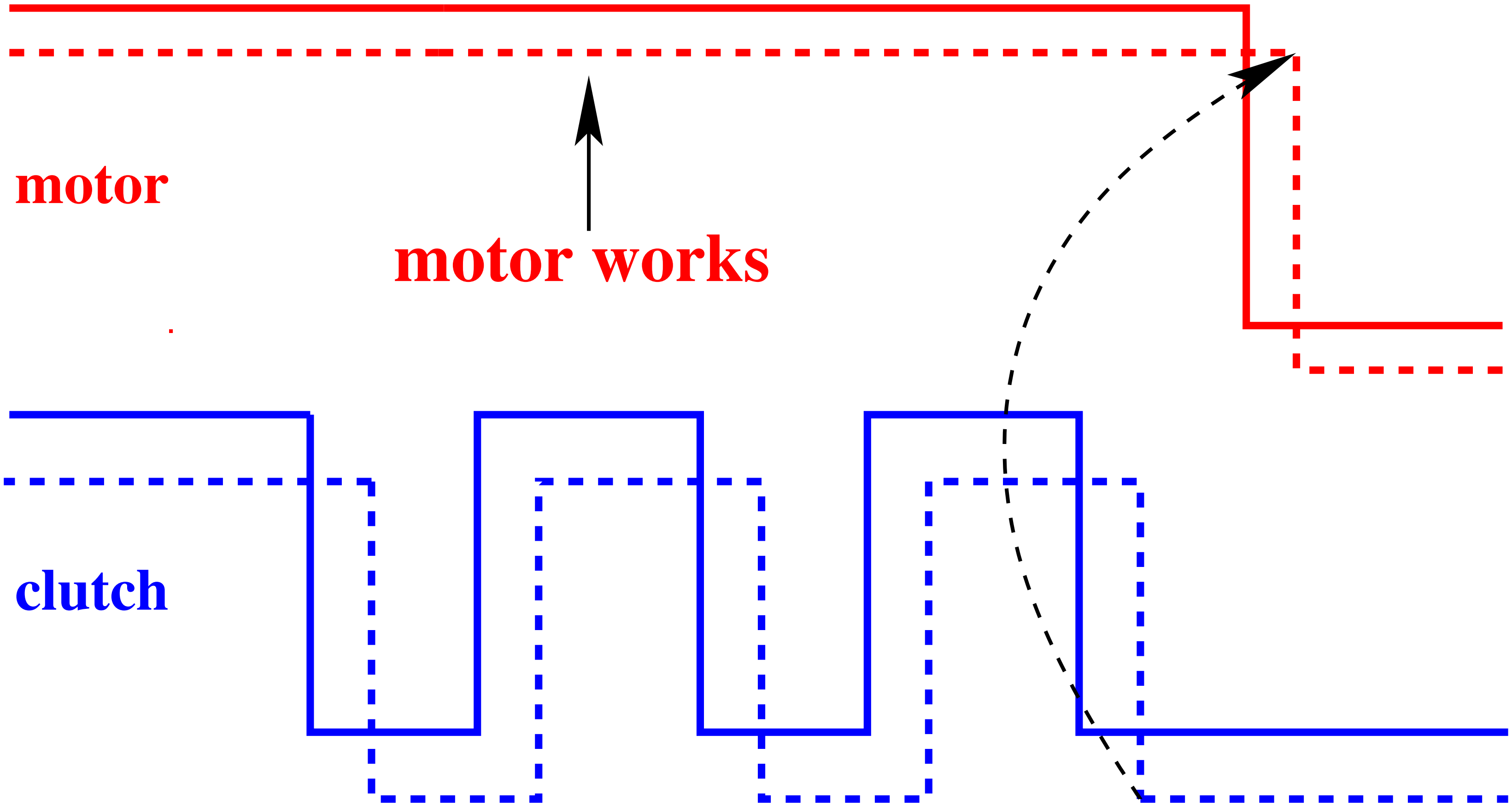


When the clutch is disengaged,

then

the motor can be started and stopped several times

The Synchronization is Weak (2)

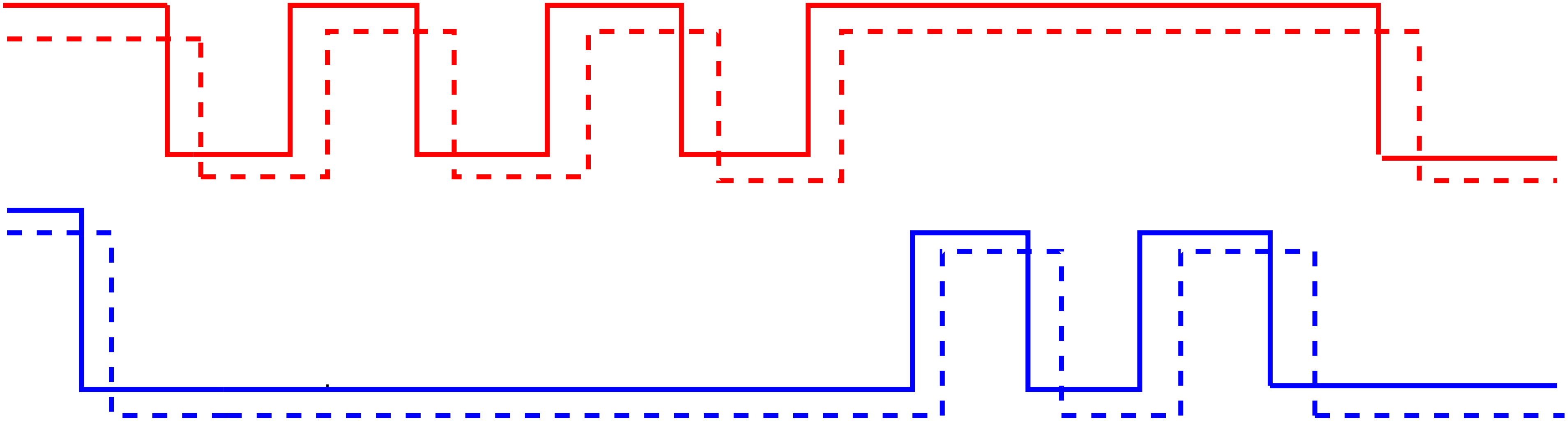


When the motor works,

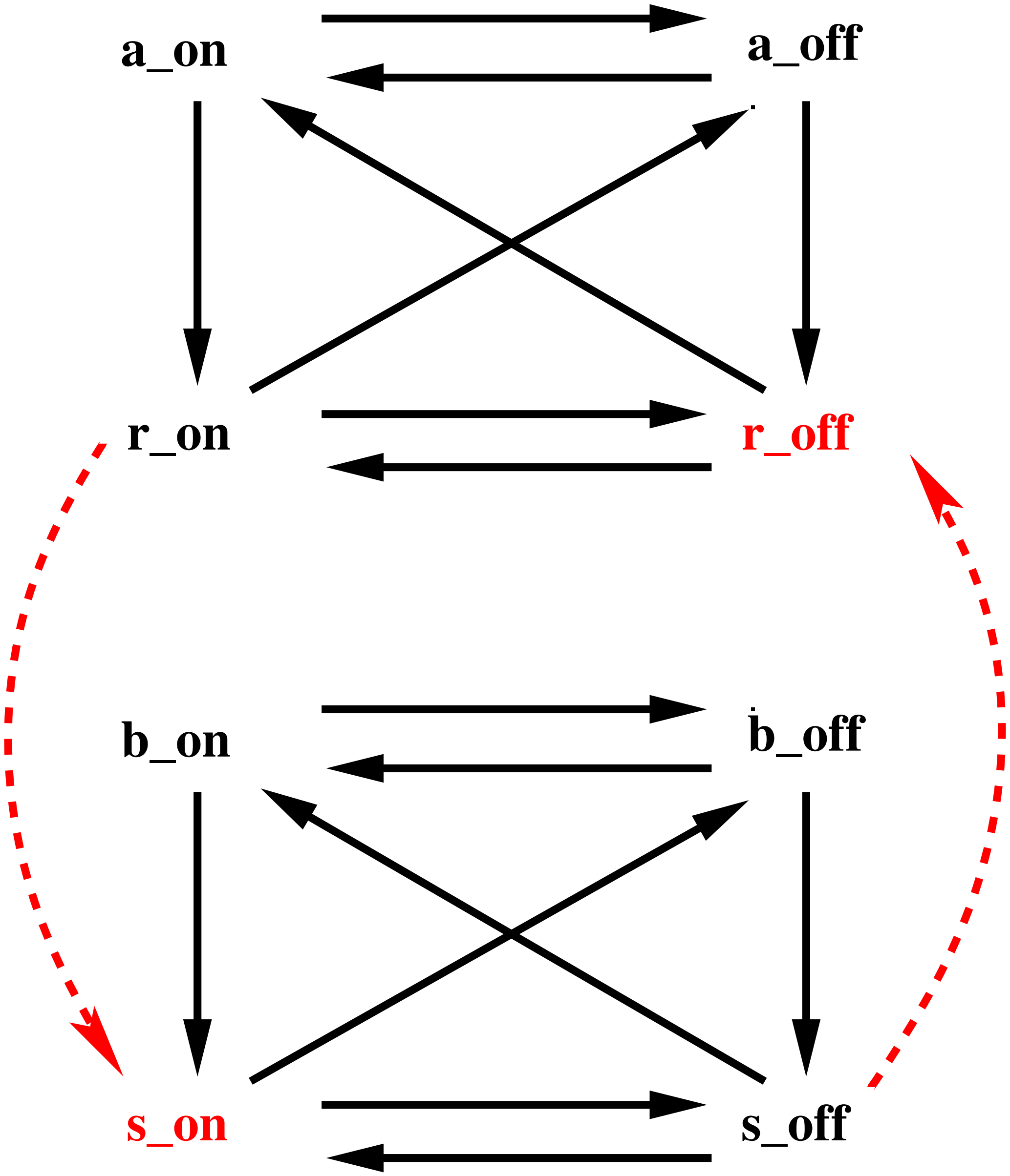
then

the clutch can be engaged and disengaged several times

Putting the Two Together



Synchronizing the Reactions Without Touching them



The Initial State Situation

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: $b = 0 \vee s = 1 \Rightarrow cb = cs$

The Initial Event Situation (1)

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

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

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

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

The Initial Event Situation (2)

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

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

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

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

The Synchronizing Invariant

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

- It seems sufficient to add the following guards

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

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

- But we do not want to touch these events

Introducing Additional Invariants to Remove the red guards

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

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

- We introduce the following additional invariants

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

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

Maintaining Invariant dbl1_2 (1)

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

In order to maintain this invariant, we have to refine b_on

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

$\leadsto$

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

Maintaining (Contraposition of) Invariant dbl1_2 (2)

dbl1_2: $b = 1 \Rightarrow r = 1 \qquad (r = 0 \Rightarrow b = 0)$

In order to maintain this invariant, we have to refine r_off

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

$\leadsto$

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

- But, again, we do not want to touch this event

Introducing a new invariant to Remove the Red Guard

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

- We introduce the following invariant

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

Maintaining Invariant dbl1_3 (1)

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

In order to maintain this invariant, we have to refine a_off

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

$\leadsto$

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

Maintaining (Contraposition of) Invariant dbl1_3 (2)

dbl1_3: $a = 0 \;\Rightarrow\; s = 0 \qquad (s = 1 \;\Rightarrow\; a = 1)$

In order to maintain this invariant, we have to refine s_on

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

$\leadsto$

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

- But, again, we do not want to touch this event

Introducing a new invariant to Remove the Red Guard

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

- We have to introduce the following invariant

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

- Fortunately, this is **dbl1_4** ($a = 0 \;\Rightarrow\; b = 0$) contraposed

Maintaining Invariant dbl1_4 (1)

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

In order to maintain this invariant, we have to refine a_off again

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

$\leadsto$

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

Maintaining (Contraposition of) Invariant dbl1_4 (2)

dbl1_4: $a = 0 \;\Rightarrow\; b = 0 \qquad (b = 1 \;\Rightarrow\; a = 1)$

In order to maintain this invariant, we have to refine b_on again

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

$\leadsto$

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

Summary of Refinement: Reactions have not been Touched

dbl1_1: $s = 1 \Rightarrow r = 1$
dbl1_2: $b = 1 \Rightarrow r = 1$
dbl1_3: $a = 0 \Rightarrow s = 0$
dbl1_4: $a = 0 \Rightarrow b = 0$

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

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

Intuition about the Invariants

dbl1_1: $s = 1 \quad \Rightarrow \quad r = 1$
dbl1_2: $b = 1 \quad \Rightarrow \quad r = 1$
dbl1_3: $a = 0 \quad \Rightarrow \quad s = 0 \qquad (s = 1 \quad \Rightarrow \quad a = 1)$
dbl1_4: $a = 0 \quad \Rightarrow \quad b = 0 \qquad (b = 1 \quad \Rightarrow \quad a = 1)$

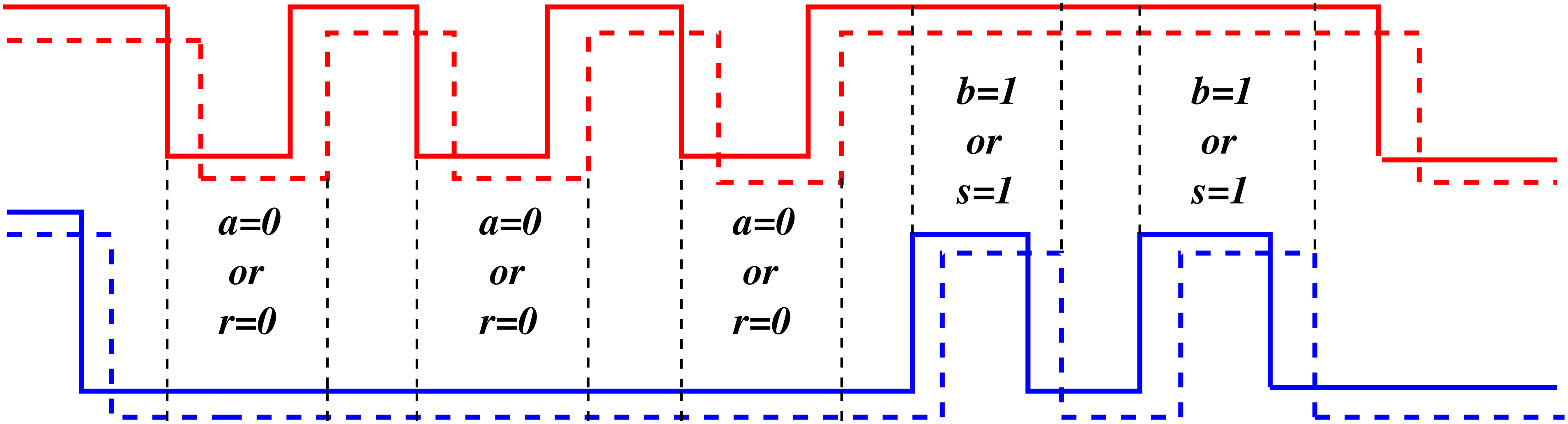
This can be put into a single invariant

dbl1_5: $b = 1 \ \lor \ s = 1 \quad \Rightarrow \quad a = 1 \ \land \ r = 1$

with the following contraposed form

dbl1_6: $a = 0 \ \lor \ r = 0 \quad \Rightarrow \quad b = 0 \ \land \ s = 0$

Intuition about the Invariants

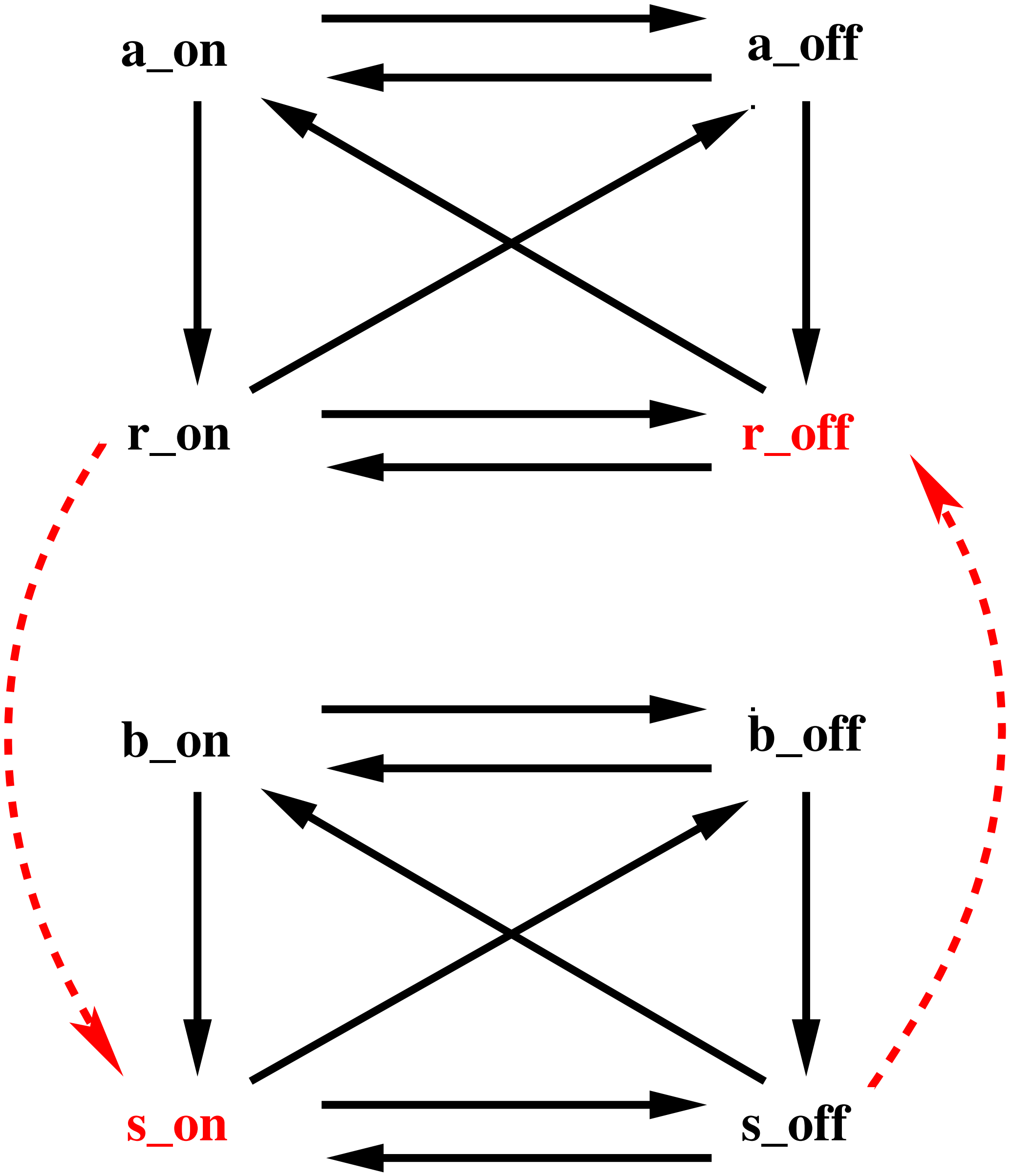


Reminder: - - - is the motor and - - - is the clutch

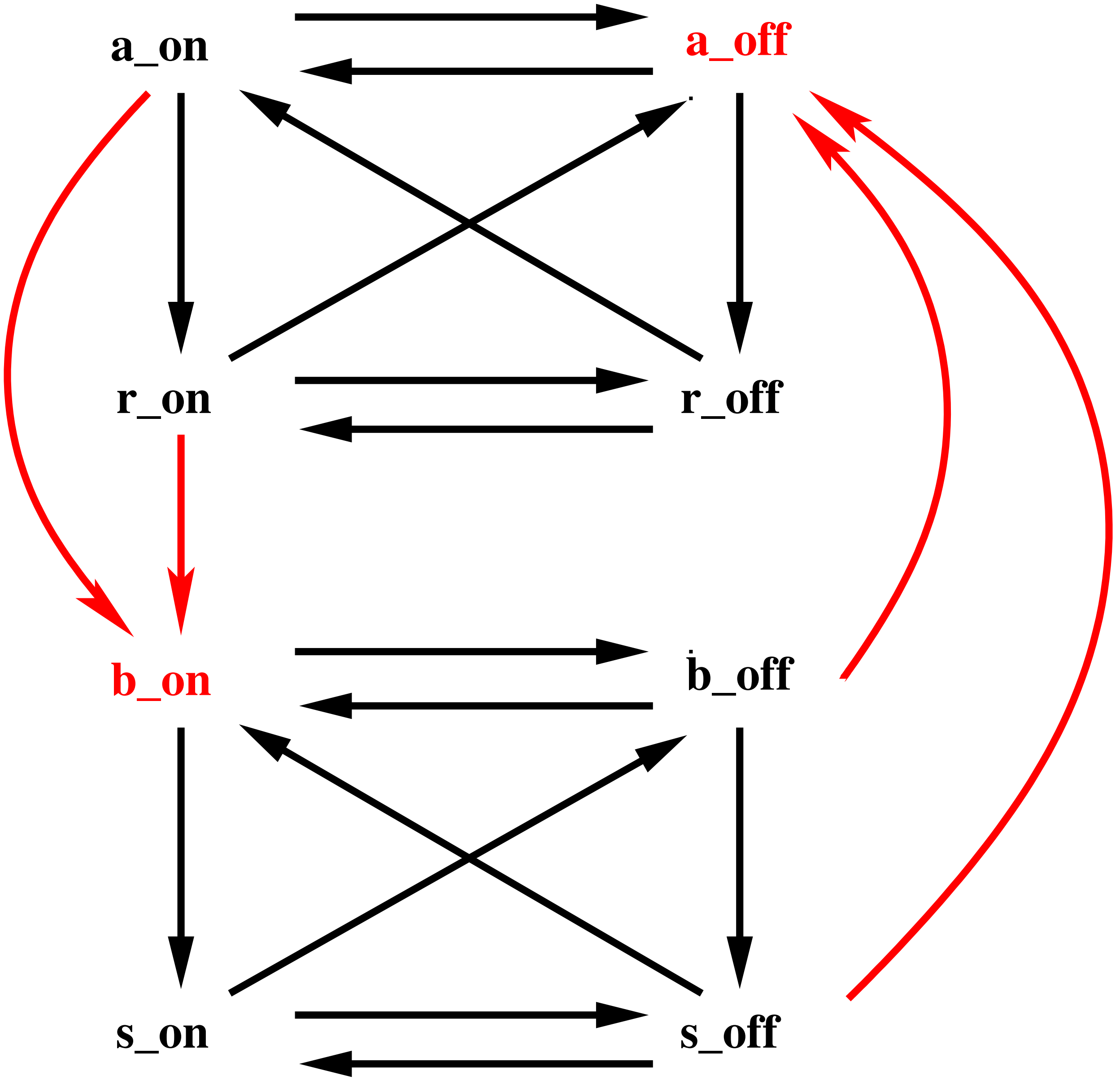
dbl1_5: $b = 1 \lor s = 1 \Rightarrow a = 1 \land r = 1$

dbl1_6: $a = 0 \lor r = 0 \Rightarrow b = 0 \land s = 0$

Weak Synchronization of Strong Reaction: the Problem



Weak Synchronization of Strong Reaction: the Solution



Back to 3rd Reft.: Constraining the Clutch and the Motor

When the clutch is engaged, the motor must work	SAF_1

inv3_1: $$\begin{array}{l} clutch\_sensor = engaged \\ \Rightarrow \\ motor\_sensor = working \end{array}$$

- This is an instance of the previous design pattern

Pattern Instantiation

- We instantiate the pattern as follows:

$\boldsymbol{a}$	$\rightsquigarrow$	$\boldsymbol{motor\_actuator}$	a_on	$\rightsquigarrow$	treat_push_start_motor_button
$\boldsymbol{r}$	$\rightsquigarrow$	$\boldsymbol{motor\_sensor}$	a_off	$\rightsquigarrow$	treat_push_stop_motor_button
$\mathbf{0}$	$\rightsquigarrow$	$\boldsymbol{stopped}$	r_on	$\rightsquigarrow$	Motor_start
$\mathbf{1}$	$\rightsquigarrow$	$\boldsymbol{working}$	r_off	$\rightsquigarrow$	Motor_stop

$\boldsymbol{b}$	$\rightsquigarrow$	$\boldsymbol{clutch\_actuator}$	b_on	$\rightsquigarrow$	treat_start_clutch
$\boldsymbol{s}$	$\rightsquigarrow$	$\boldsymbol{clutch\_sensor}$	b_off	$\rightsquigarrow$	treat_stop_clutch
$\mathbf{0}$	$\rightsquigarrow$	$\boldsymbol{disengaged}$	s_on	$\rightsquigarrow$	Clutch_start
$\mathbf{1}$	$\rightsquigarrow$	$\boldsymbol{engaged}$	s_off	$\rightsquigarrow$	Clutch_stop

Translating the pattern invariants (1)

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

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

inv3_1: $$\begin{array}{l} clutch\_sensor = engaged \\ \Rightarrow \\ motor\_sensor = working \end{array}$$

inv3_2: $$\begin{array}{l} clutch\_actuator = engaged \\ \Rightarrow \\ motor\_sensor = working \end{array}$$

Translating the pattern invariants (2)

dbl1_3: $a = 0 \;\Rightarrow\; s = 0$
dbl1_4: $a = 0 \;\Rightarrow\; b = 0$

inv3_3:	$motor\_actuator = stopped$ $\Rightarrow$ $clutch\_sensor = disengaged$
inv3_4:	$motor\_actuator = stopped$ $\Rightarrow$ $clutch\_actuator = disengaged$

Adapting the Events of the Pattern (1)

```
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
```

Adapting the events of the pattern (2)

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

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

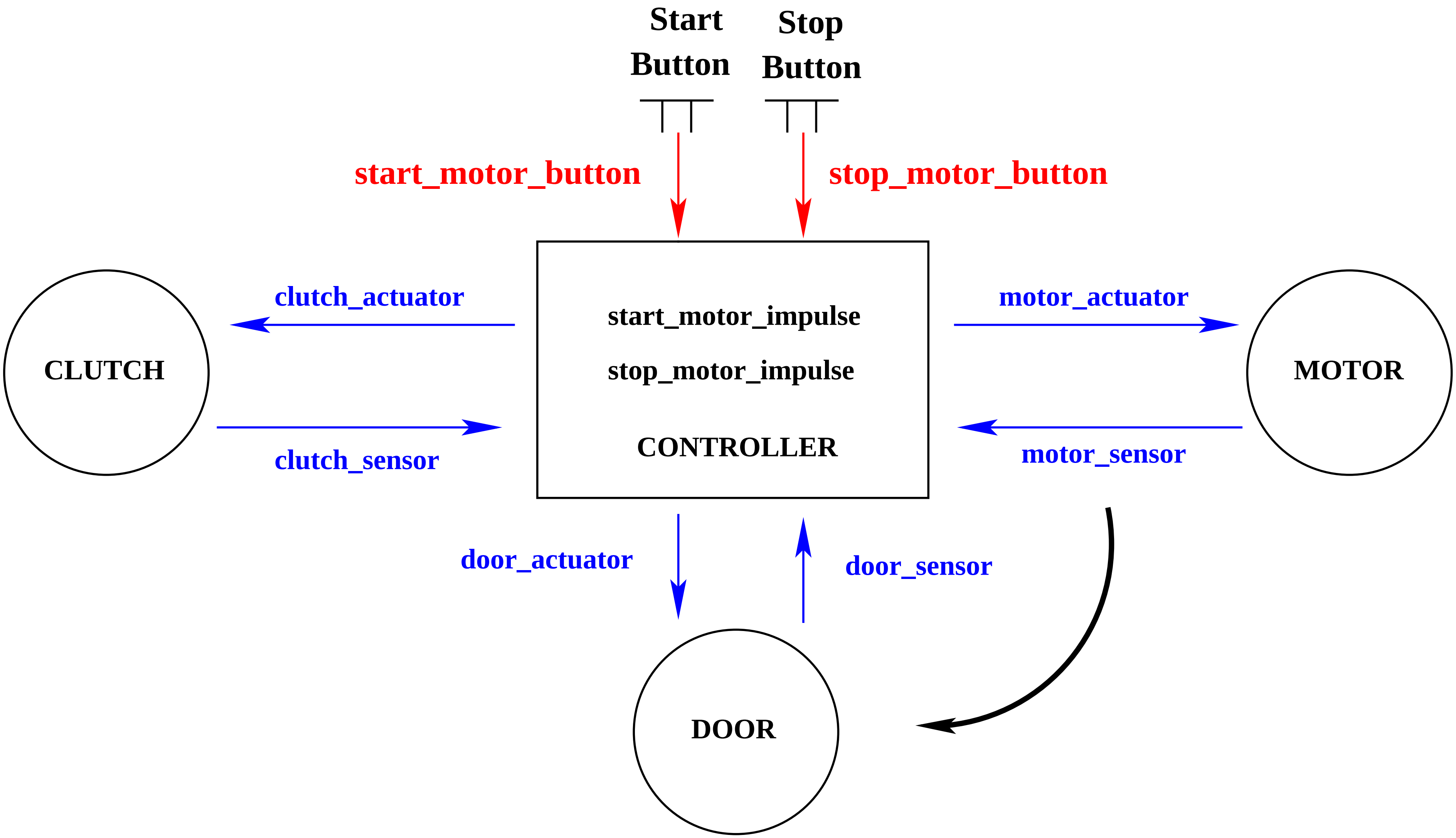
Third Refinement: Summary of the Events (1)

- Environment (no new events)
    - motor_start
    - motor_stop
    - clutch_start
    - clutch_stop
    - push_start_motor_button
    - release_start_motor_button
    - push_stop_motor_button
    - release_stop_motor_button

Third Refinement: Summary of the Events (2)

- Controller (no new events)
  - 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

Fourth Refinement: Connecting the Controller to the Door



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

Fourth Refinement: Connecting the Controller to the Door

- We introduce the set in a new context:

$$DOOR = \{open, closed\}$$

- We copy the initial model where we instantiate:

$$\begin{array}{lcl} motor & \leadsto & door \\ STATUS & \leadsto & DOOR \\ working & \leadsto & closed \\ stopped & \leadsto & open \end{array}$$

Fourth Refinement: Summary of the Events (1)

- 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

Fourth Refinement: Summary of the Events (2)

- 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

Fifth Refinement: Constraining the Clutch and the Door

- 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

Fifth Reft.: Something was forgotten Concerning the Door

Fifth Reft.: Something was forgotten Concerning the Door

- Can you guess it?

Fifth Reft.: Something was forgotten Concerning the Door

- Can you guess it?

- When the motor is not working, we must allow users:
  - to change the tool
  - to replace the part to be treated

Fifth Reft.: Something was forgotten Concerning the Door

- Can you guess it?

- When the motor is not working, we must allow users:
  - to change the tool
  - to replace the part to be treated

- Hence the following additional requirement (which was forgotten)

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

Fifth Reft.: Something was forgotten Concerning the Door

- Can you guess it?

- When the motor is not working, we must allow users:
  - to change the tool
  - to replace the part to be treated

- Hence the following additional requirement (which was forgotten)

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

- SAF_3’ is the contraposed form of SAF_3

Fifth Refinement: Taking Care of the New Constraint

- Additional safety constraint

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

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

When the clutch is engaged, the motor must work	SAF_1

Fifth Reft.: Summary of the Safety Requirements

When the clutch is engaged, the motor must work	SAF_1
When the clutch is engaged, the door must be closed	SAF_2
When the door is closed, the motor must work	SAF_3’

- Requirement SAF_1 is now redundant: SAF_2 ∧ SAF_3’ ⇒ SAF_1

Possible New Refinement 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 (4th) refinement: Connecting the controller to the door

Possible New Refinement Strategy (cont’d)

- 4th (5th) refinement: Constraining the clutch and the door

  Constraining the motor and the door

- 5th (6th) refinement: More constraints between clutch and door

- 6th (7th) refinement: Connecting the clutch button to the controller

Fifth Refinement: Summary of the Events (1)

- Environment (no new events)
  - 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

Fifth Refinement: Summary of the Events (2)

- Controller (no new events)
  - 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

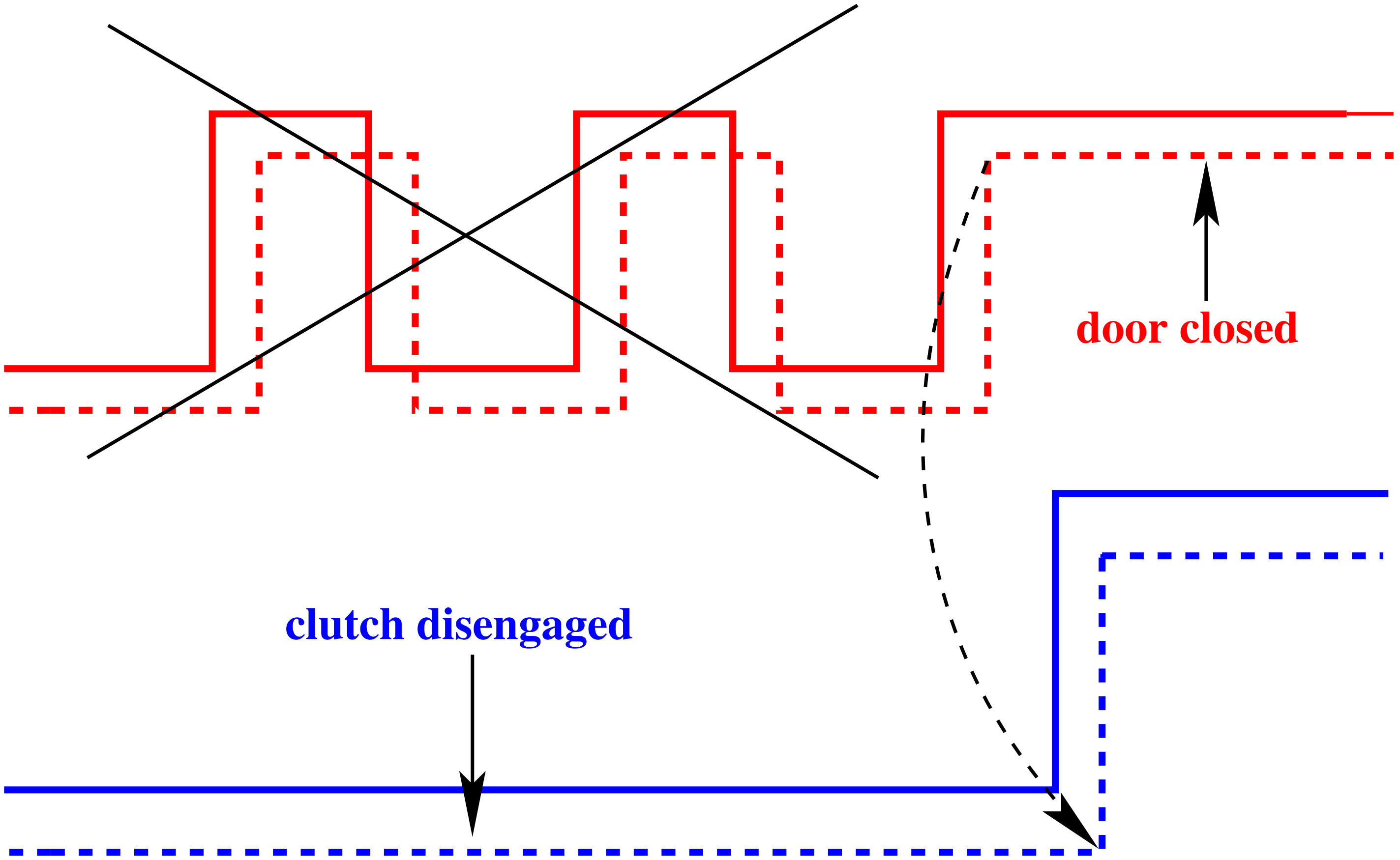
Sixth Reft.: More Constraints between Clutch and Door

- Adding two functional constraints

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

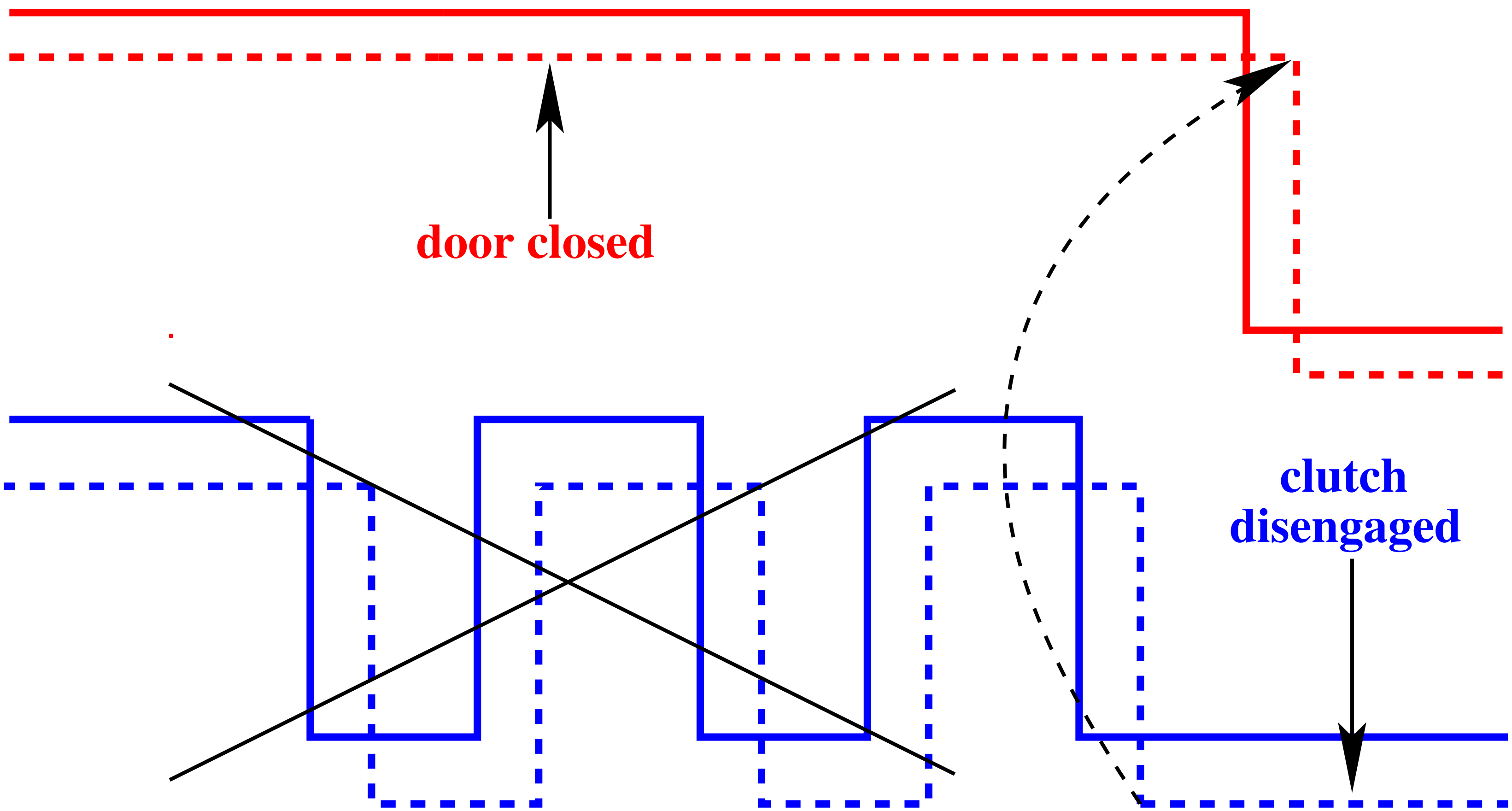
When the door is closed, the clutch cannot be disengaged several times, ONLY ONCE	FUN_4

Problem with the Weak Synchronization of Strong Reactions



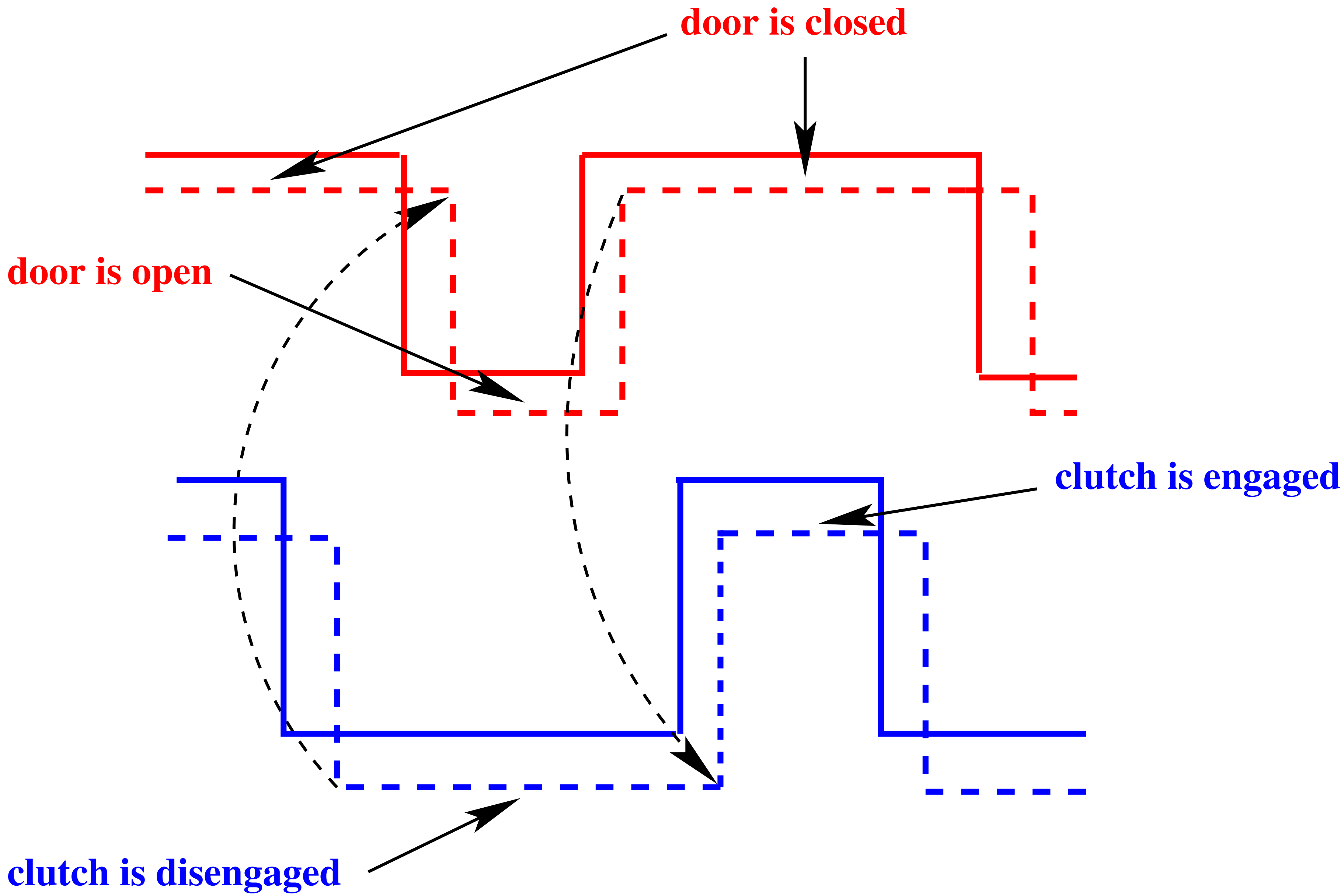
- When the clutch is disengaged, the door cannot be closed several times

Problem with the Weak Synchronization of Strong Reactions

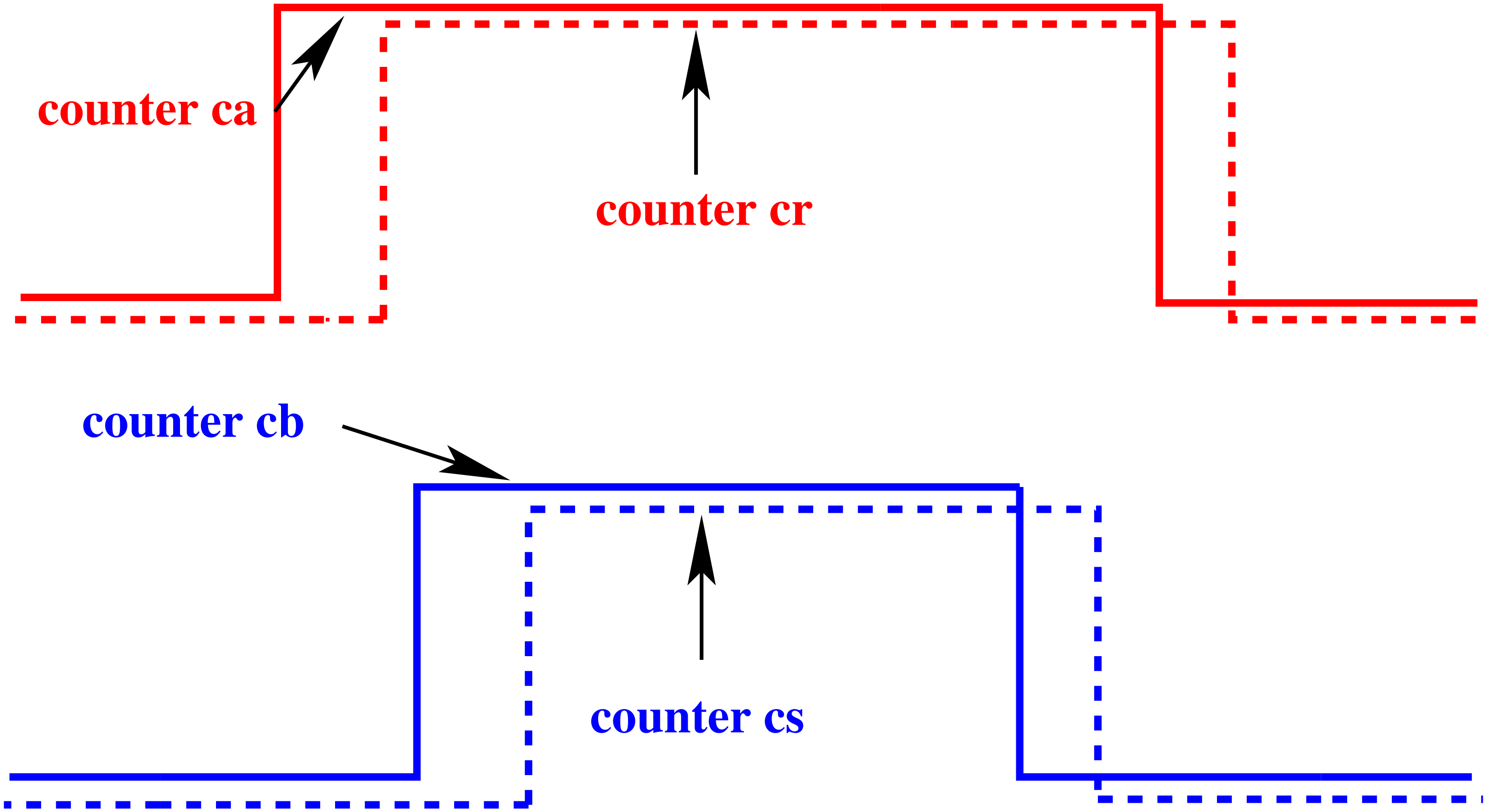


- When the door is closed, the clutch cannot be disengaged several times

The Full Picture



Strong Synchronization of two Strong Reactions

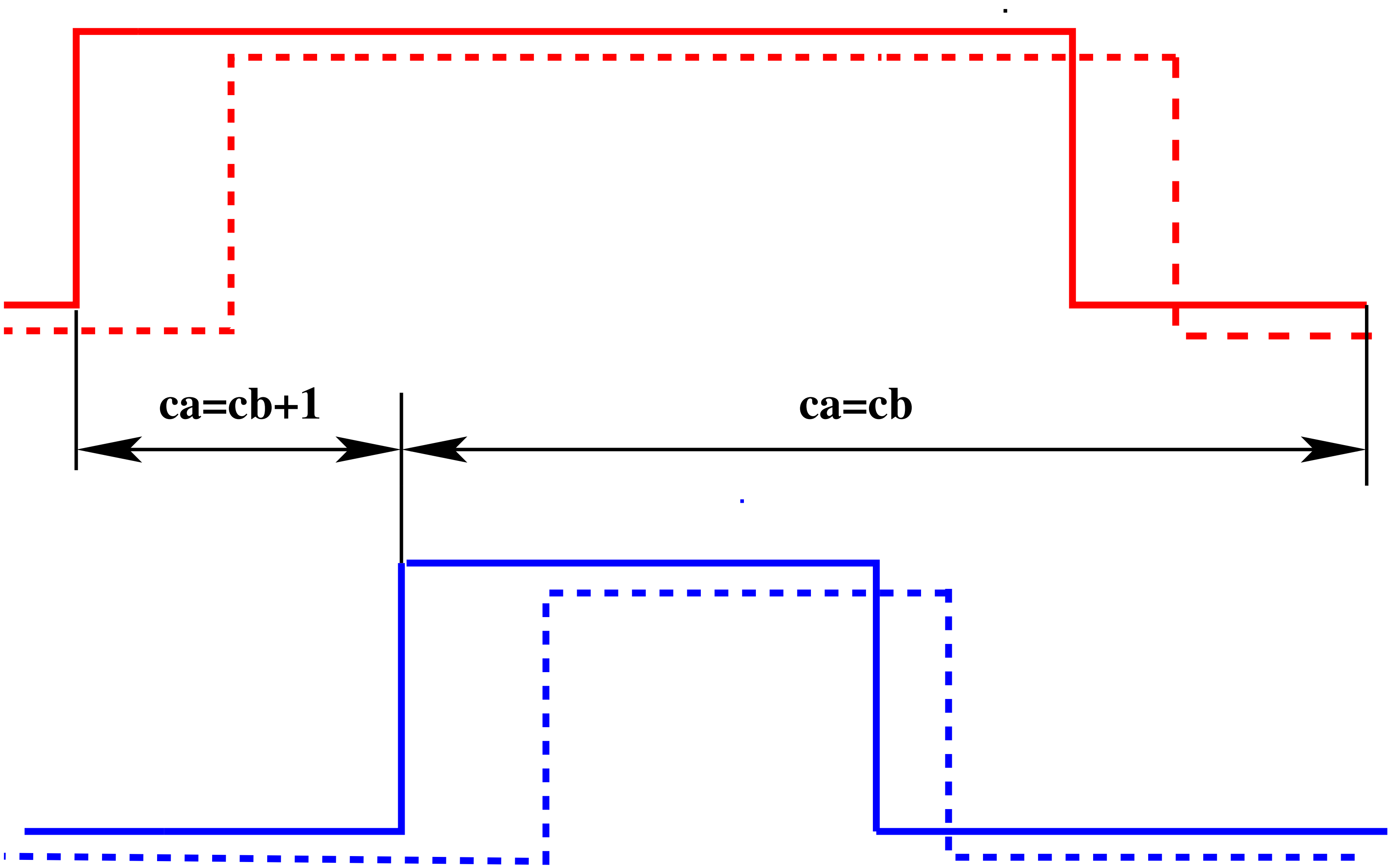


What we want:

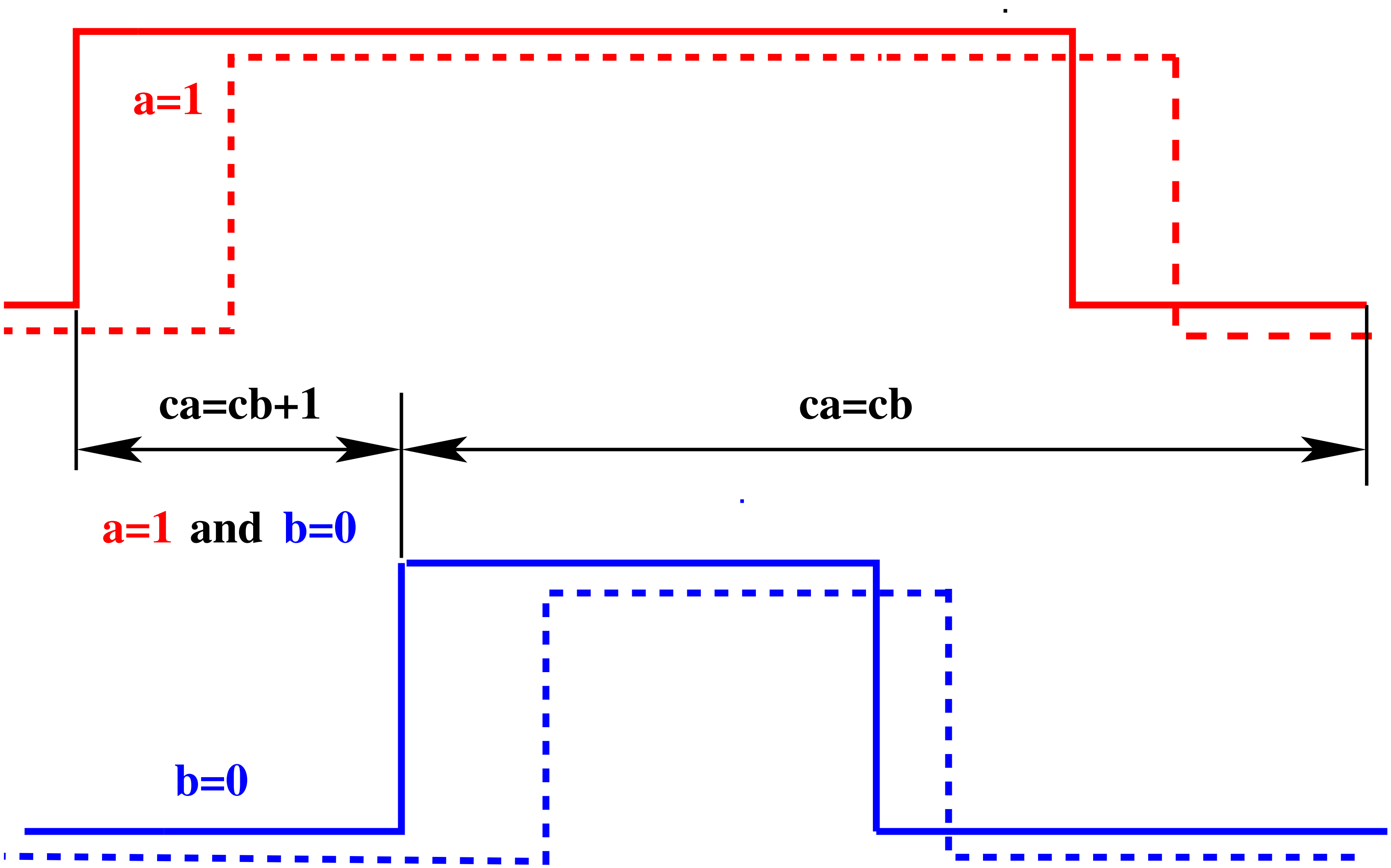
$$ca = cb \ \vee \ ca = cb + 1$$

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

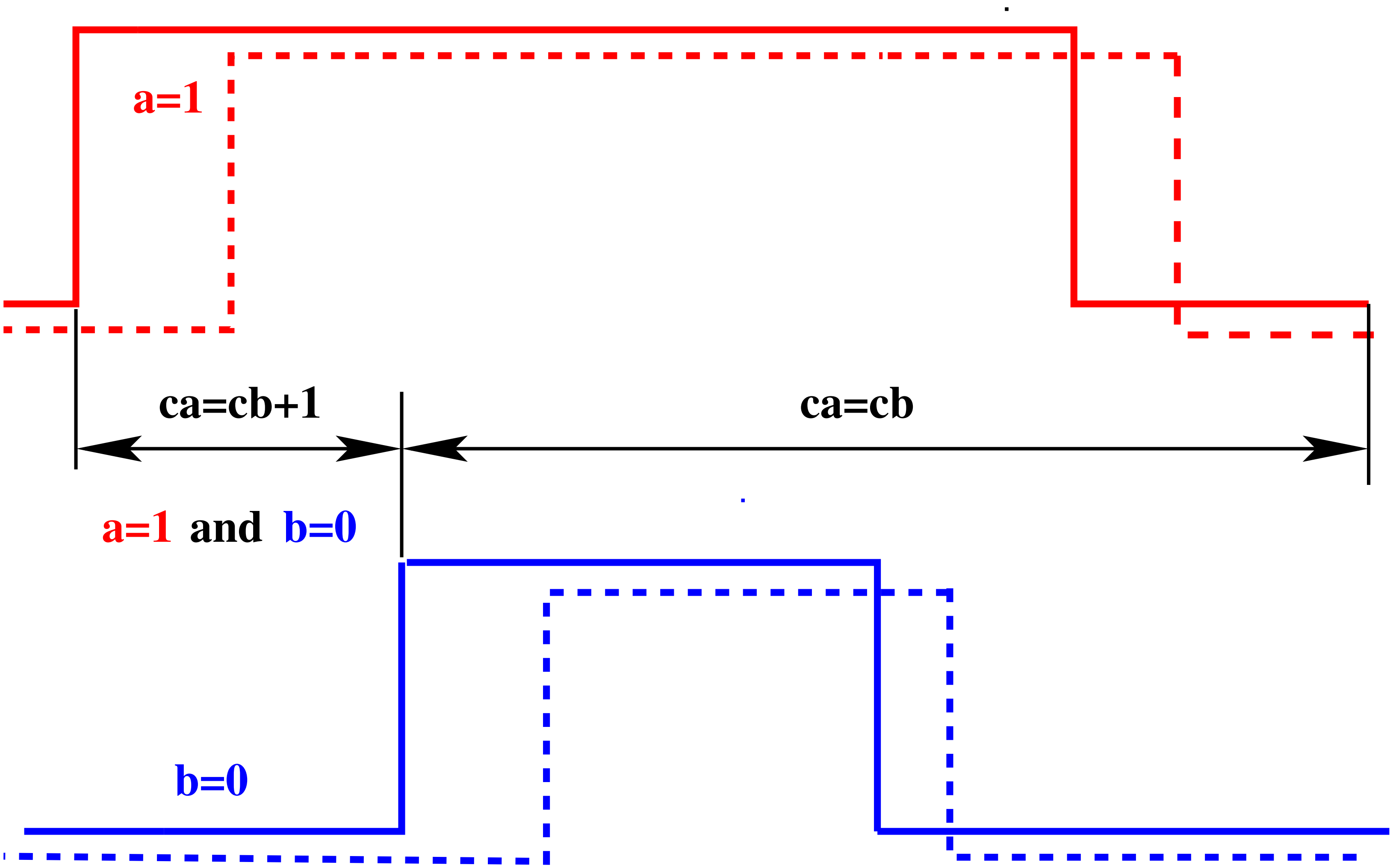
How about counters ca and cb?



In Search of a Solution

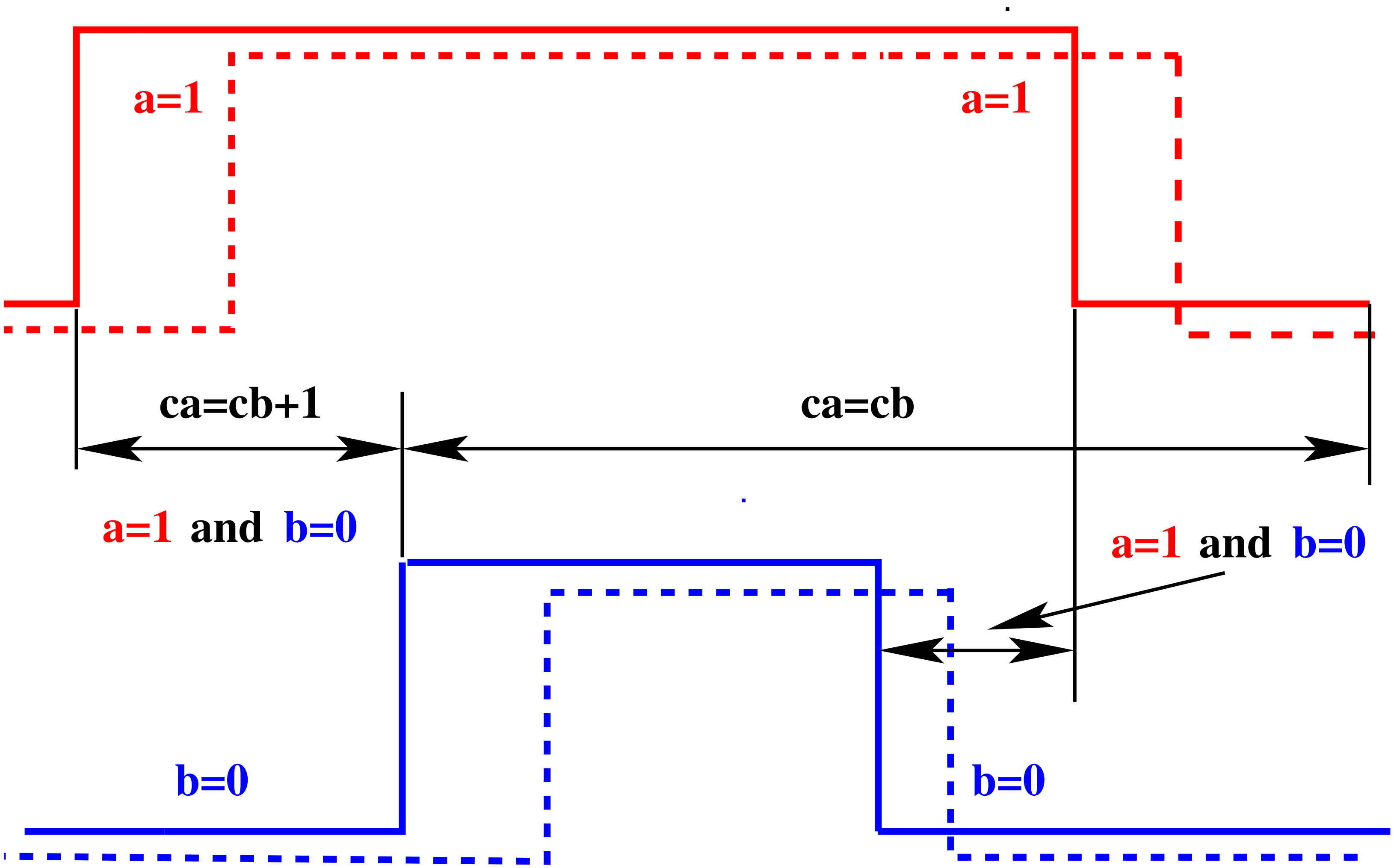


In Search of a Solution

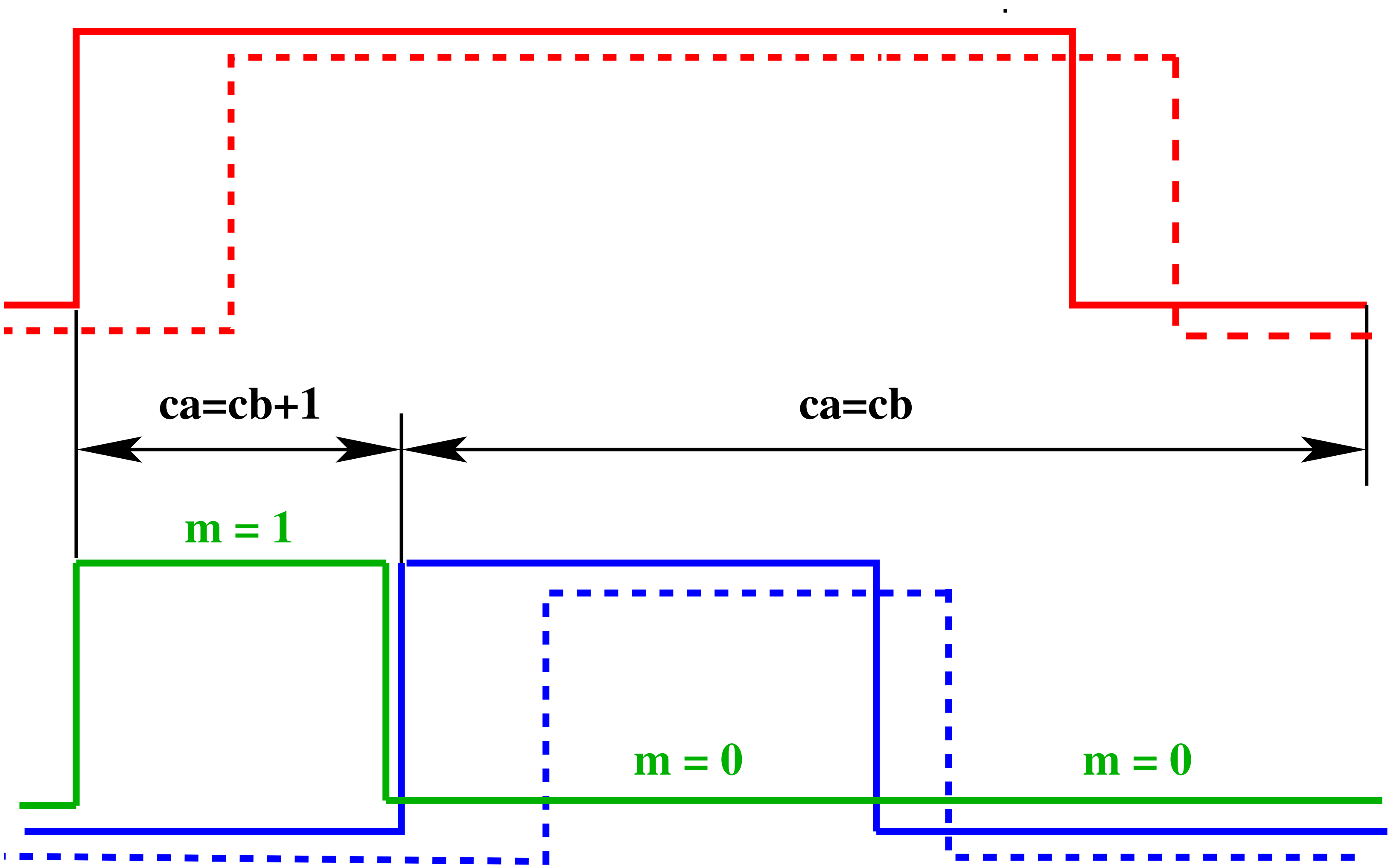


$$a = 1 \ \wedge \ b = 0 \quad \Rightarrow \quad ca = cb + 1 \qquad \textbf{?}$$

This Solution Does not Work

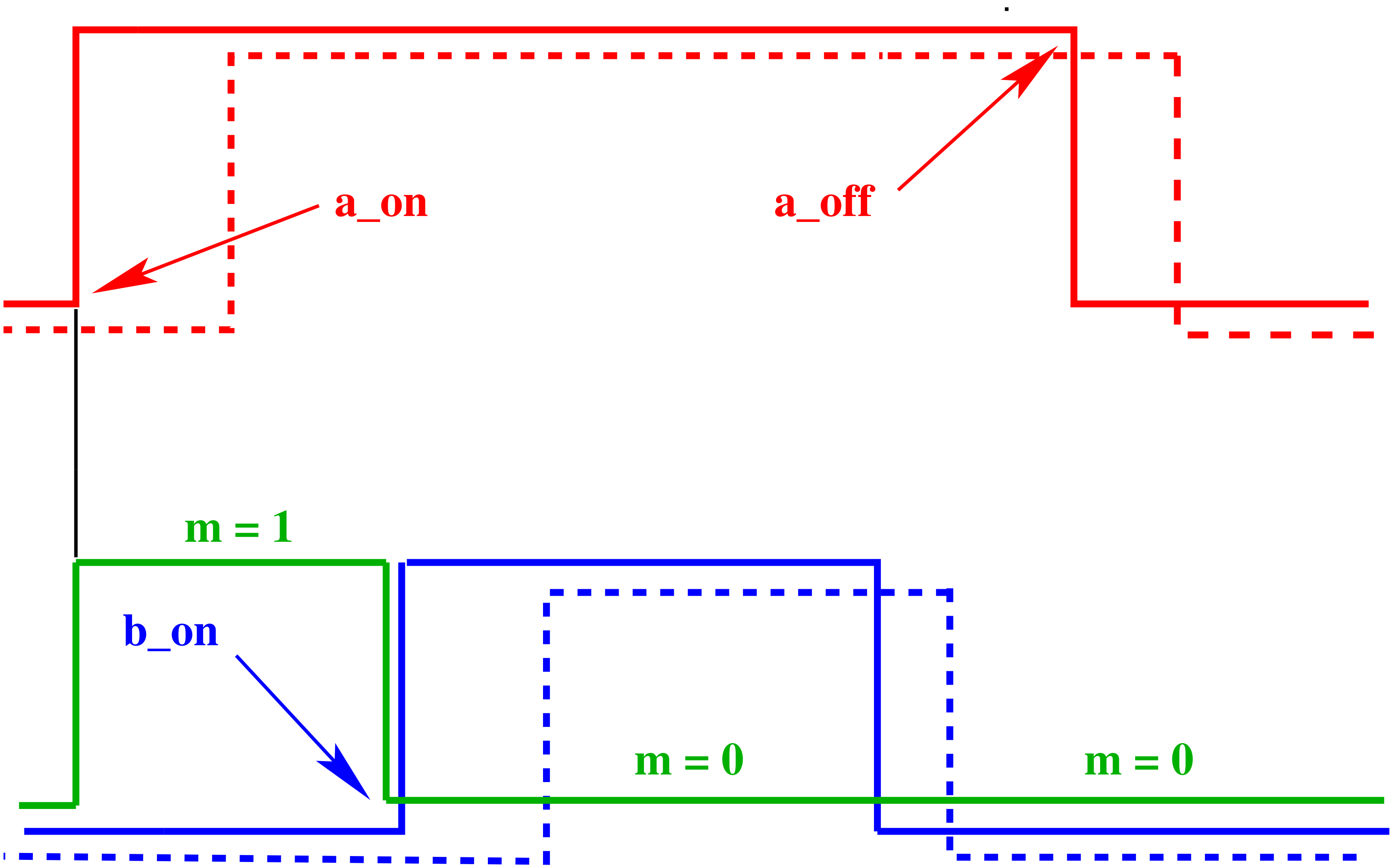


The Solution: an Additional Variable m



$$m = 1 \quad \Rightarrow \quad ca = cb + 1$$
$$m = 0 \quad \Rightarrow \quad ca = cb$$

The Events



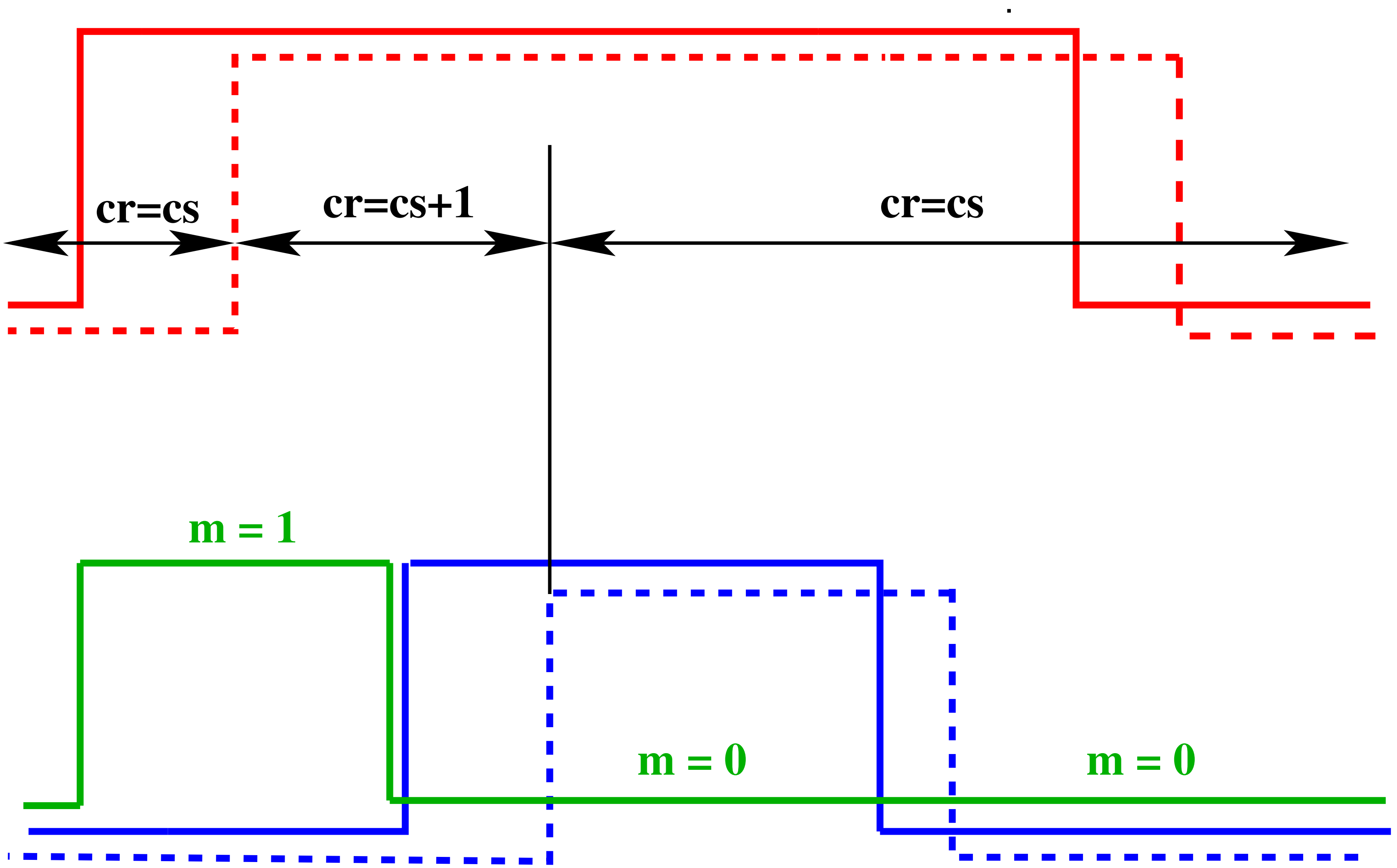
The Modified Events

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

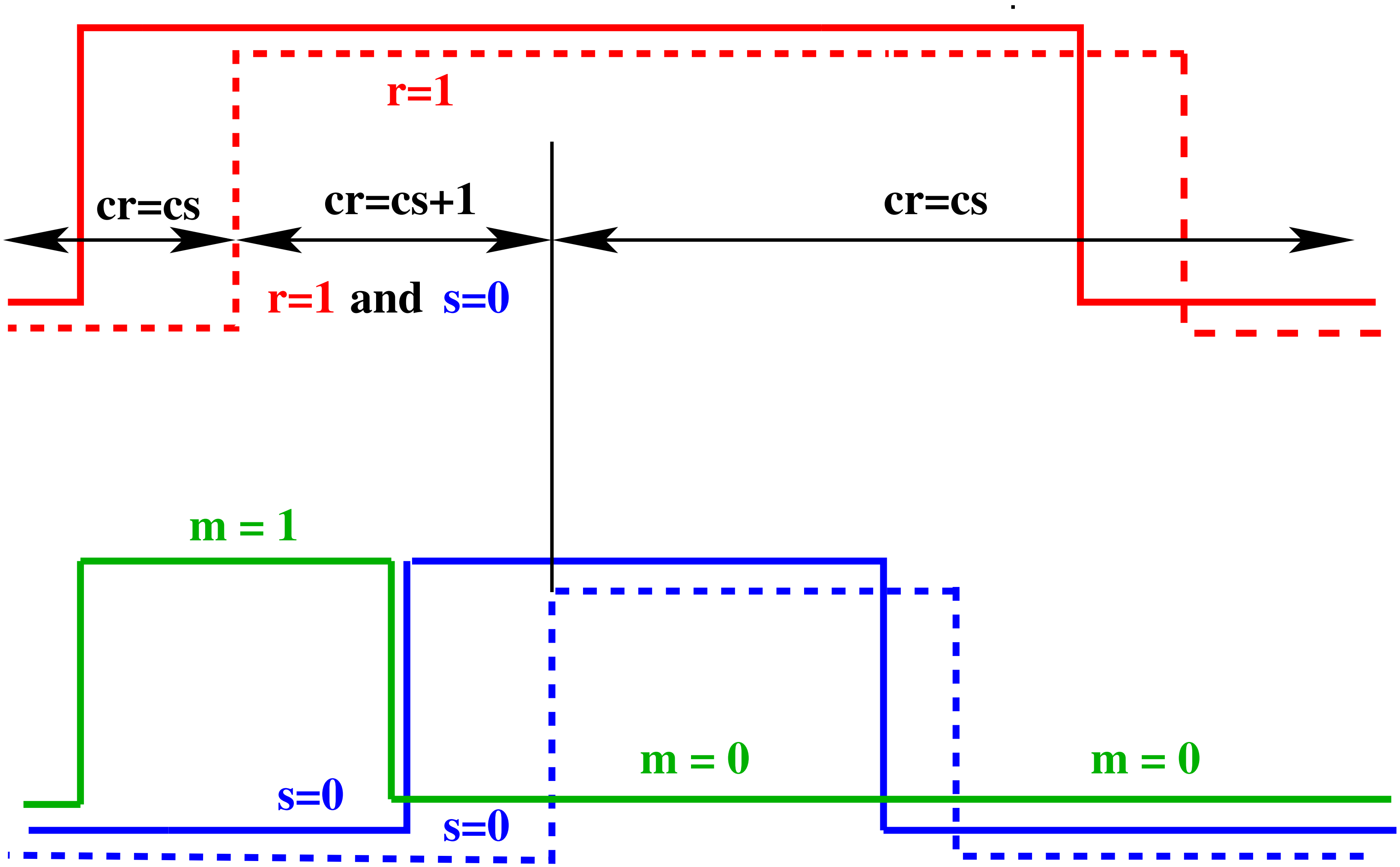
```
b_on
when
    $r = 1$
    $a = 1$
    $b = 0$
    $s = 0$
    $m = 1$
then
    $b := 1$
    $cb := cb + 1$
    $m := 0$
end
```

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

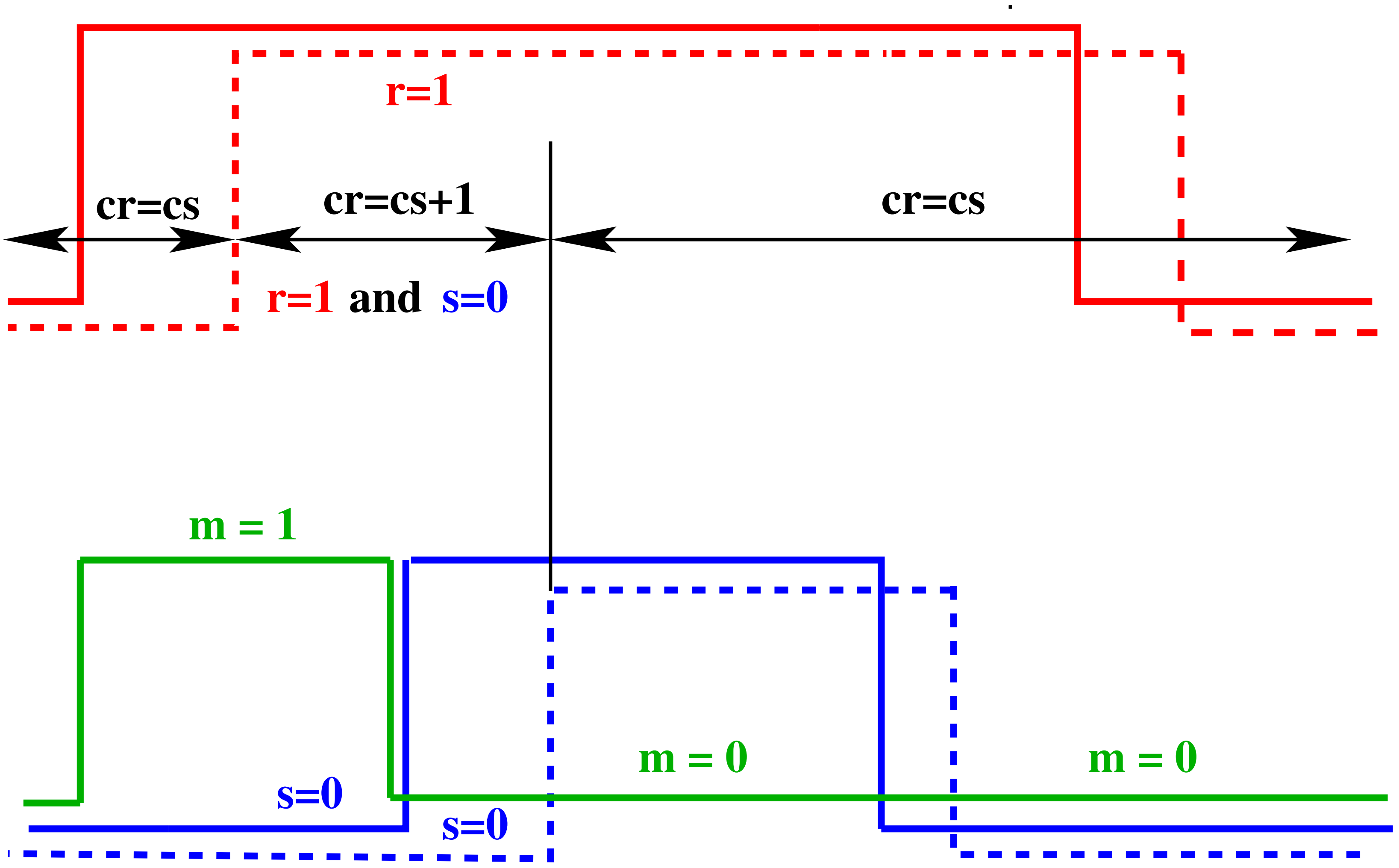
How about counters cr and cs



In Search of a Solution

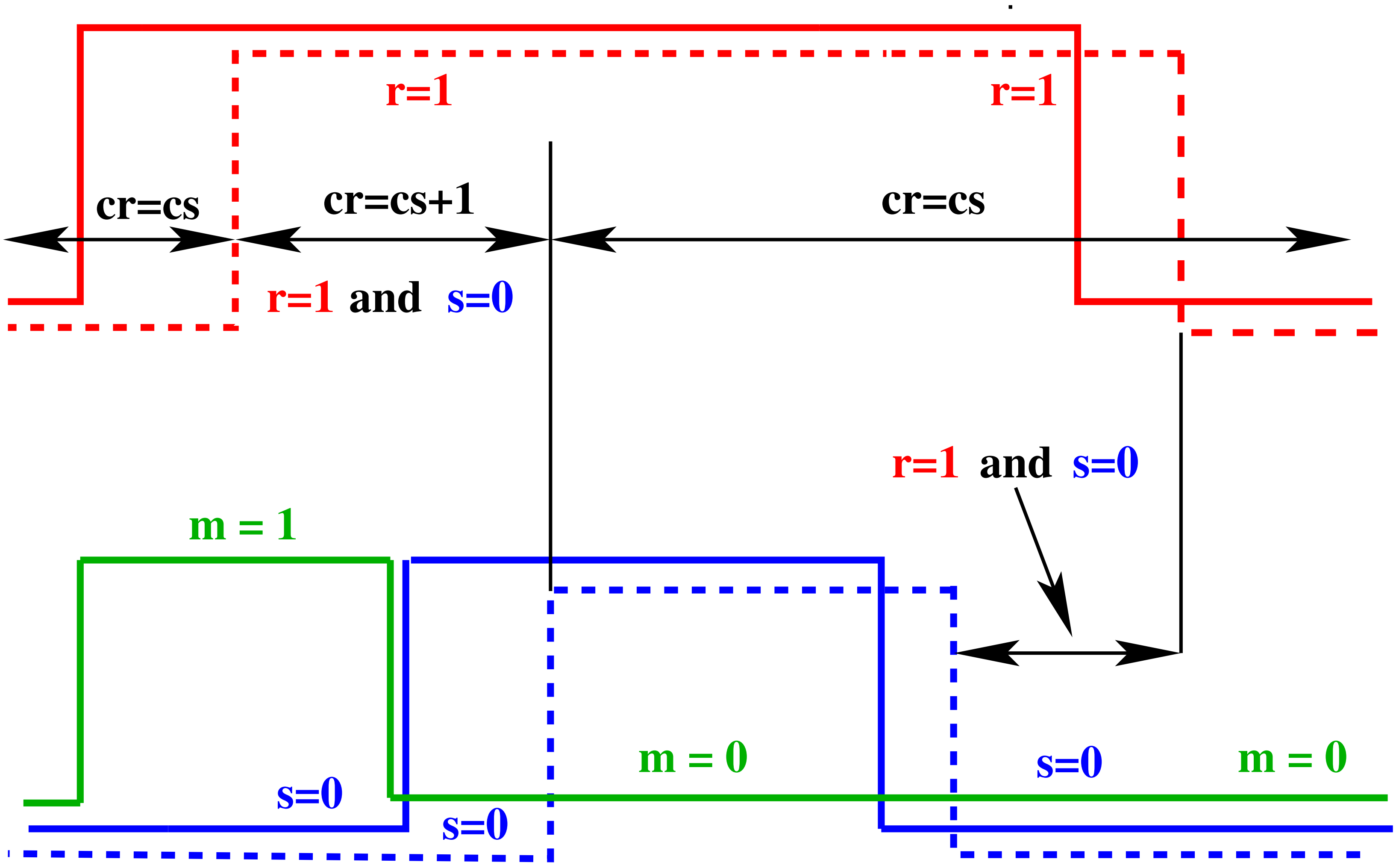


In Search of a Solution

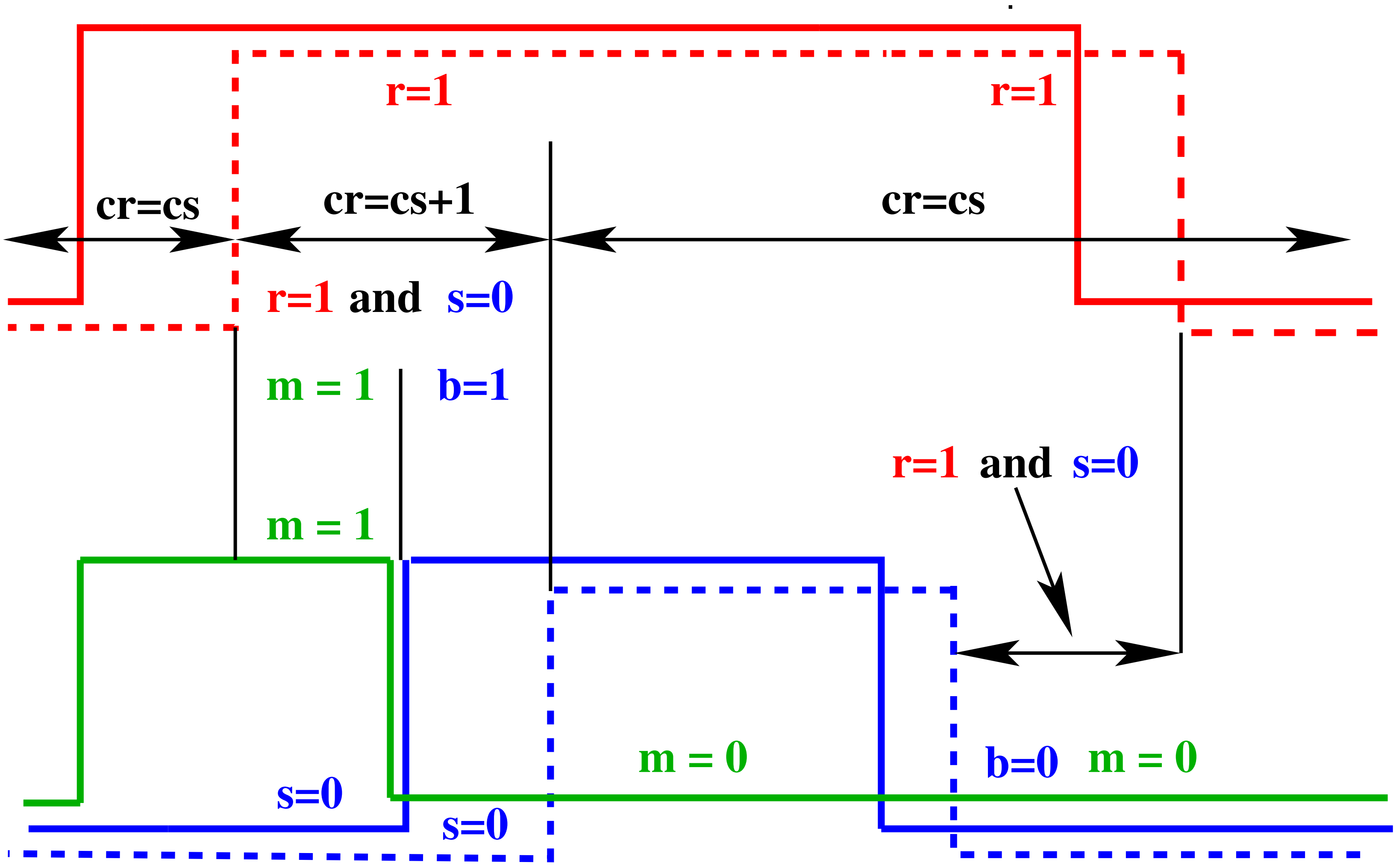


$$r = 1 \;\wedge\; s = 0 \;\;\Rightarrow\;\; cr = cs + 1 \qquad \text{?}$$

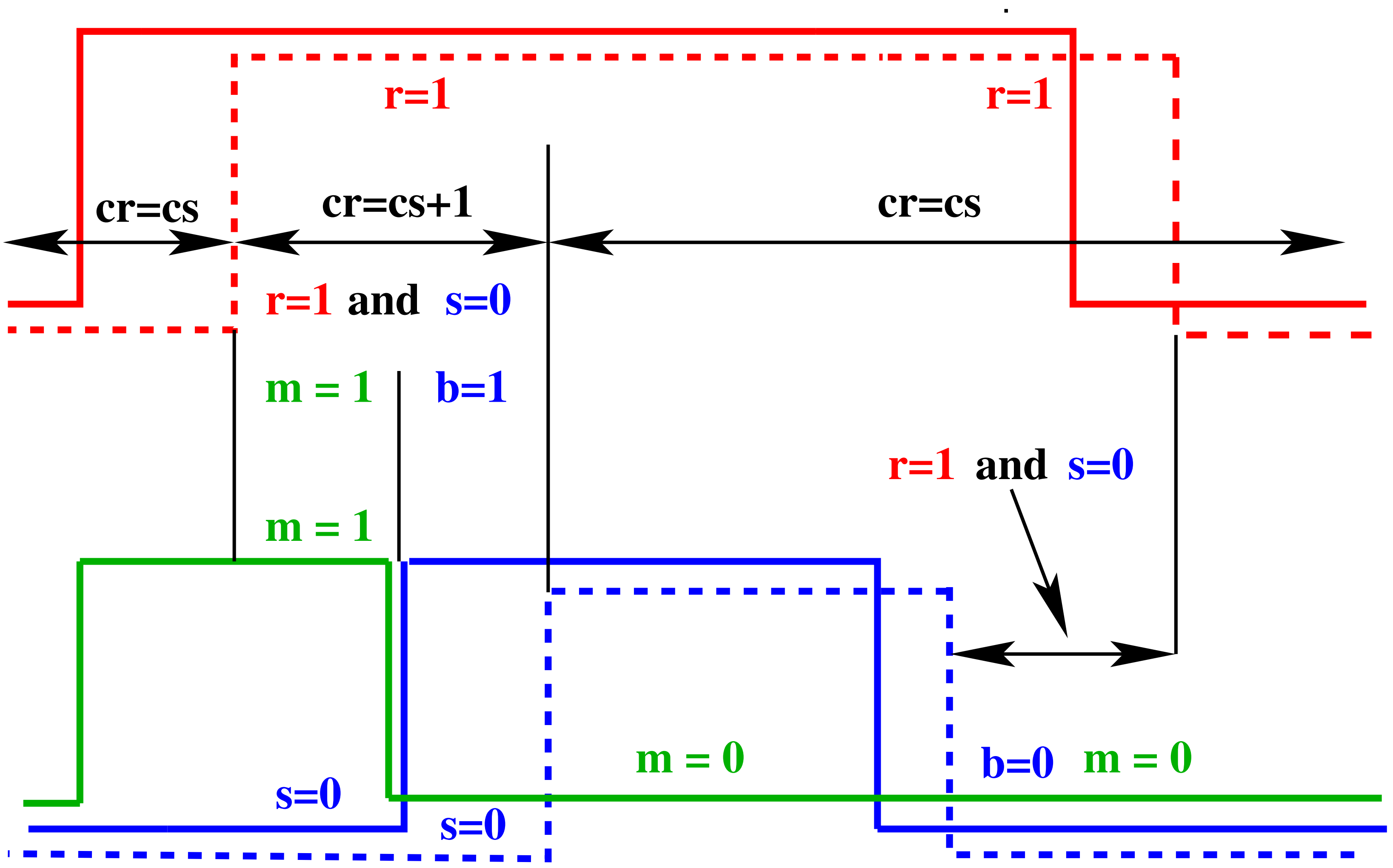
This Solution Does not Work



The Solution



The Solution



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

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

Summary of Refinement: the state

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

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

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

dbl2_4: $r = 1 \;\land\; s = 0 \;\land\; (m = 1 \;\lor\; b = 1) \;\Rightarrow\; cr = cs + 1$

dbl2_5: $r = 0 \;\lor\; s = 1 \;\lor\; (m = 0 \;\land\; b = 0) \;\Rightarrow\; cr = cs$

Summary of Refinement: the state

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

- The following theorems are easy to prove

thm2_1: $ca = cb \vee ca = cb + 1$

thm2_2: $cr = cs \vee cr = cs + 1$

More Invariants

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

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

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

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

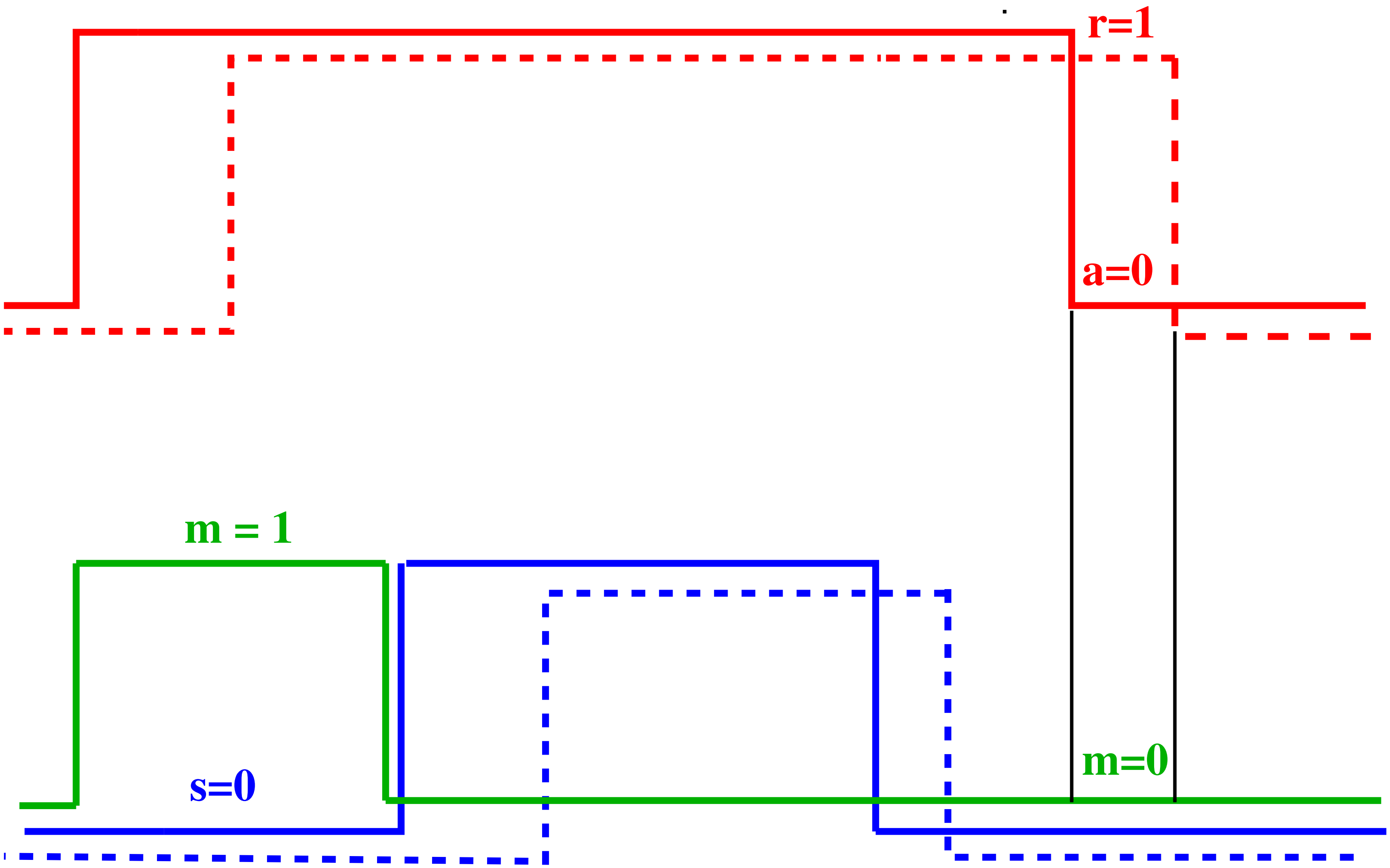
dbl2_5: $r = 0 \vee s = 1 \vee (m = 0 \wedge b = 0) \Rightarrow cr = cs$

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

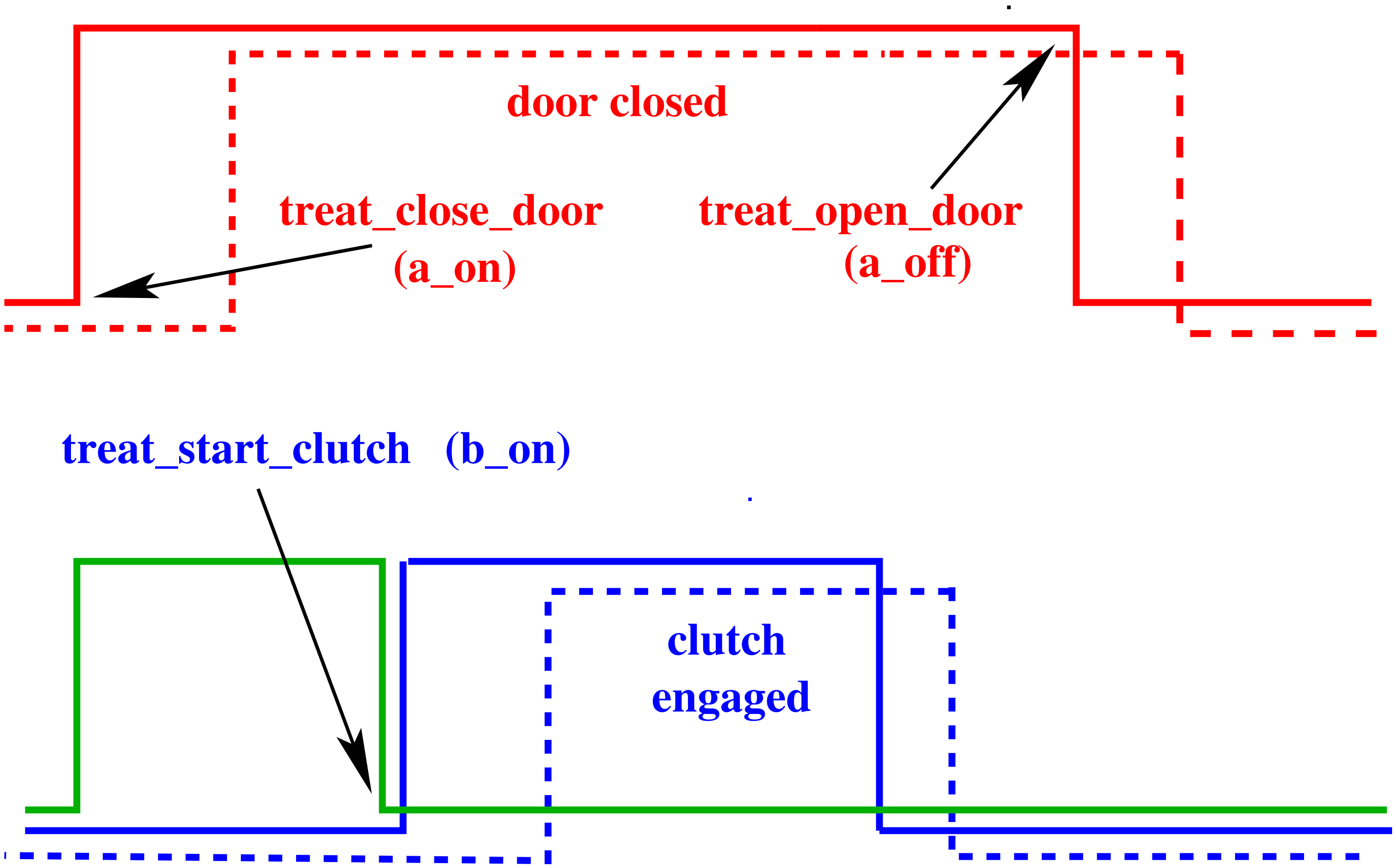
dbl2_7: $m = 1 \Rightarrow s = 0$

- The two new invariants were discovered while doing the proof

- The proofs are now completely automatic


r=1
a=0
m = 1
s=0
m=0

Instantiation



Pattern Instantiation for the 6th Refinement

- We instantiate the pattern as follows:

$$\begin{array}{lllllll}
a & \leadsto & door\_actuator & \quad & b & \leadsto & clutch\_actuator \\
r & \leadsto & door\_sensor & & s & \leadsto & clutch\_sensor \\
0 & \leadsto & open & & 0 & \leadsto & disengaged \\
1 & \leadsto & closed & & 1 & \leadsto & engaged
\end{array}$$

a_on ⇝ treat_close_door

a_off ⇝ treat_open_door

b_on ⇝ treat_start_clutch

6th Refinement: Adapting the events of the pattern (2)

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

6th Refinement: Adapting the events of the pattern (2)

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

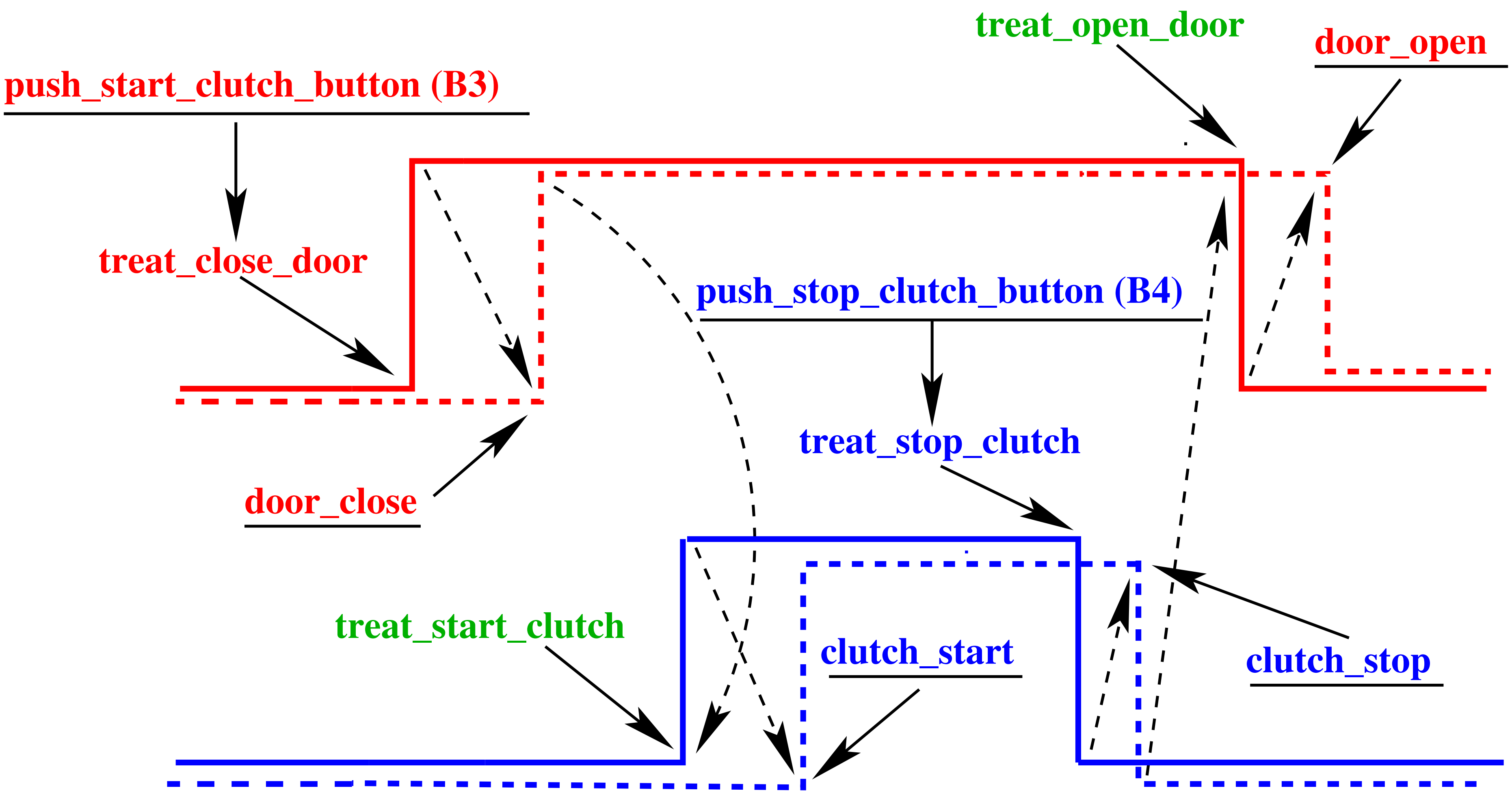
```
treat_start_clutch
  when
    $motor\_actuator = working$
    $motor\_sensor = working$
    $clutch\_actuator = disengaged$
    $clutch\_sensor = disengaged$
    $door\_sensor = closed$
    $door\_actuator = closed$
    $m = 1$
  then
    $clutch\_actuator := engaged$
    $m := 0$
  end
```

6th Refinement: Adapting the events of the pattern (3)

```
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 Complete Synchronization of Door and Clutch



- treat_close_door is the result of depressing button B3

- treat_stop_clutch is the result of depressing button B4

- treat_start_clutch and treat_open_door are automatic

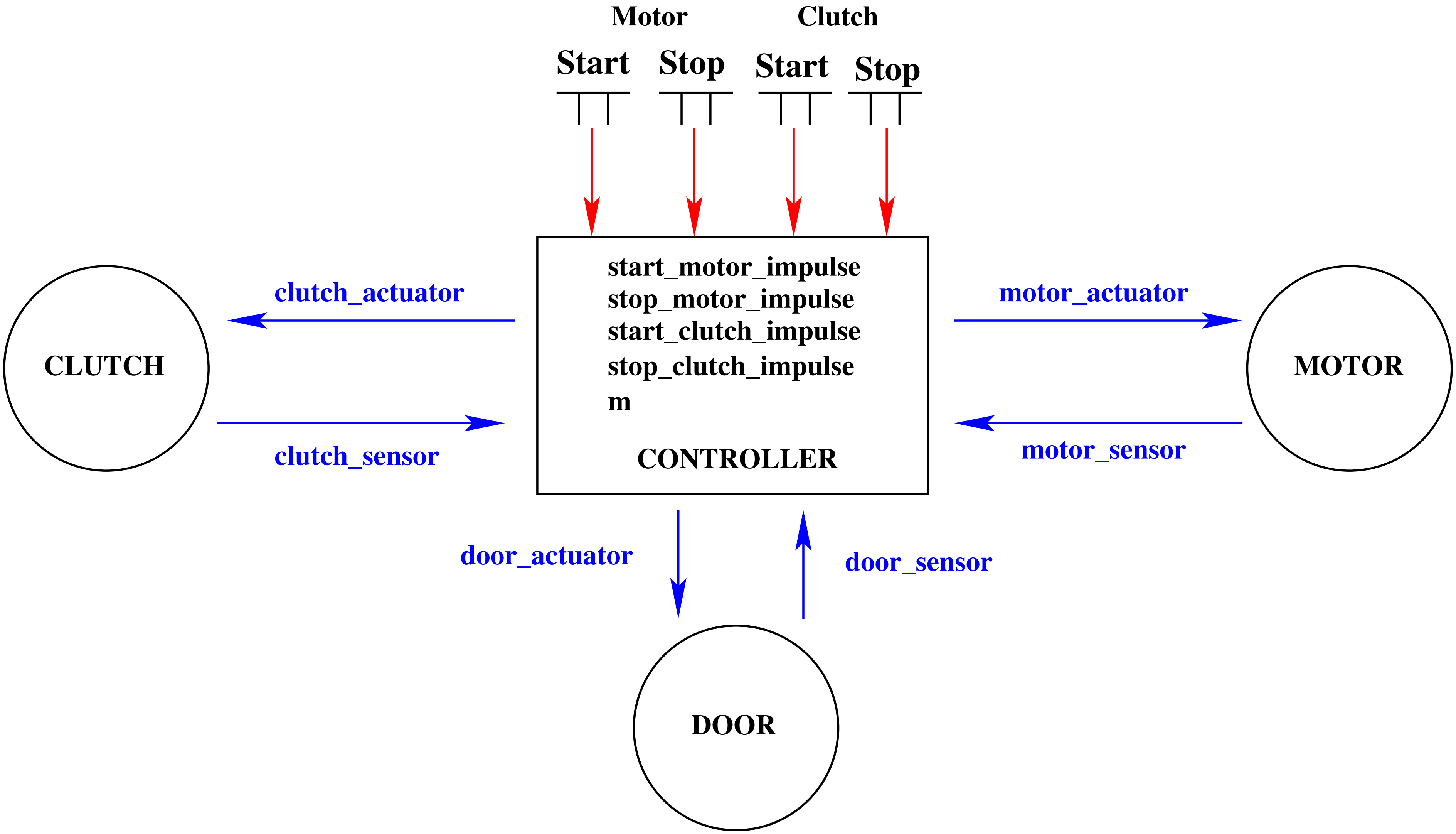
Sixth Refinement: Summary of the Events (1)

- Environment (no new events)
    - 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

Sixth Refinement: Summary of the Events (2)

- Controller (no new events)
  - 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

7th Reft.: Connecting the Controller to the Clutch Buttons



Reminder: Synchronizing Door and Clutch

- There are no door buttons

- The door must be closed before engaging the clutch

- The door must be opened after disengaging the clutch

- It is sufficient to connect:
    - button B3 to the door (closing the door)
    - button B4 to the clutch (disengaging the clutch)

Seventh Refinement: 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
- push_start_clutch_button
- release_start_clutch_button
- push_stop_clutch_button
- release_stop_clutch_button

Seventh Refinement: Summary of the Events (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
- treat_stop_clutch_false
- treat_release_start_clutch_button
- treat_release_stop_clutch_button

Decomposing the Final Model: Environment

- The environment events

- The environment variables modified by environment events

- The sensor variables modified by environment events

- The actuator variables read by environment events

- The controller variables not seen by environment events

- No environment variables in this model

Decomposing the Final Model: Controller

- The controller events

- The controller variables modified by controller events

- The sensor variables read by controller events

- The actuator variables modified by controller events

- The environment variables not seen by controller events

- No environment variables in this model

Summary: Variables of the Last Refinement (1)

- 7 sensor variables:
    - $motor\_sensor$
    - $clutch\_sensor$
    - $door\_sensor$
    - $start\_motor\_button$
    - $stop\_motor\_button$
    - $start\_clutch\_button$
    - $stop\_clutch\_button$

Summary: Variables of the Last Refinement (2)

- 3 actuator variables:
    - $motor\_actuator$
    - $clutch\_actuator$
    - $door\_actuator$
- 5 controller variables (without the counter variables):
    - $start\_motor\_impulse$
    - $stop\_motor\_impulse$
    - $start\_clutch\_impulse$
    - $stop\_clutch\_impulse$
    - m

Summary: Events of the Last Refinement

- 14 environment events,

- 14 controller events,

- 130 lines for environment events,

- 180 lines for controller events.

Summary: Usage of the Design Patterns

- 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

Summary: Number of Invariants

- Weak reaction: 6
- Strong reaction: 3
- Strong-weak reaction: 16
- Strong-strong reaction: 7
- Total: 32

- Press (typing): 15
- Total: 15

Summary: Number of Proof Obligations

- Weak reaction: 18
- Strong reaction: 12
- Strong-weak reaction: 60
- Strong-strong reaction: 40
- Total: 130

- Press: 0

- PO saving: 4x18 + 3x12 + 3x60 + 40 = 328

Summary: Proofs

- Design patterns: 2 easy interactive, out of 130

- Press: 0

Summary: Simulation

- 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)

Conclusion

- This design pattern approach seems to be fruitful

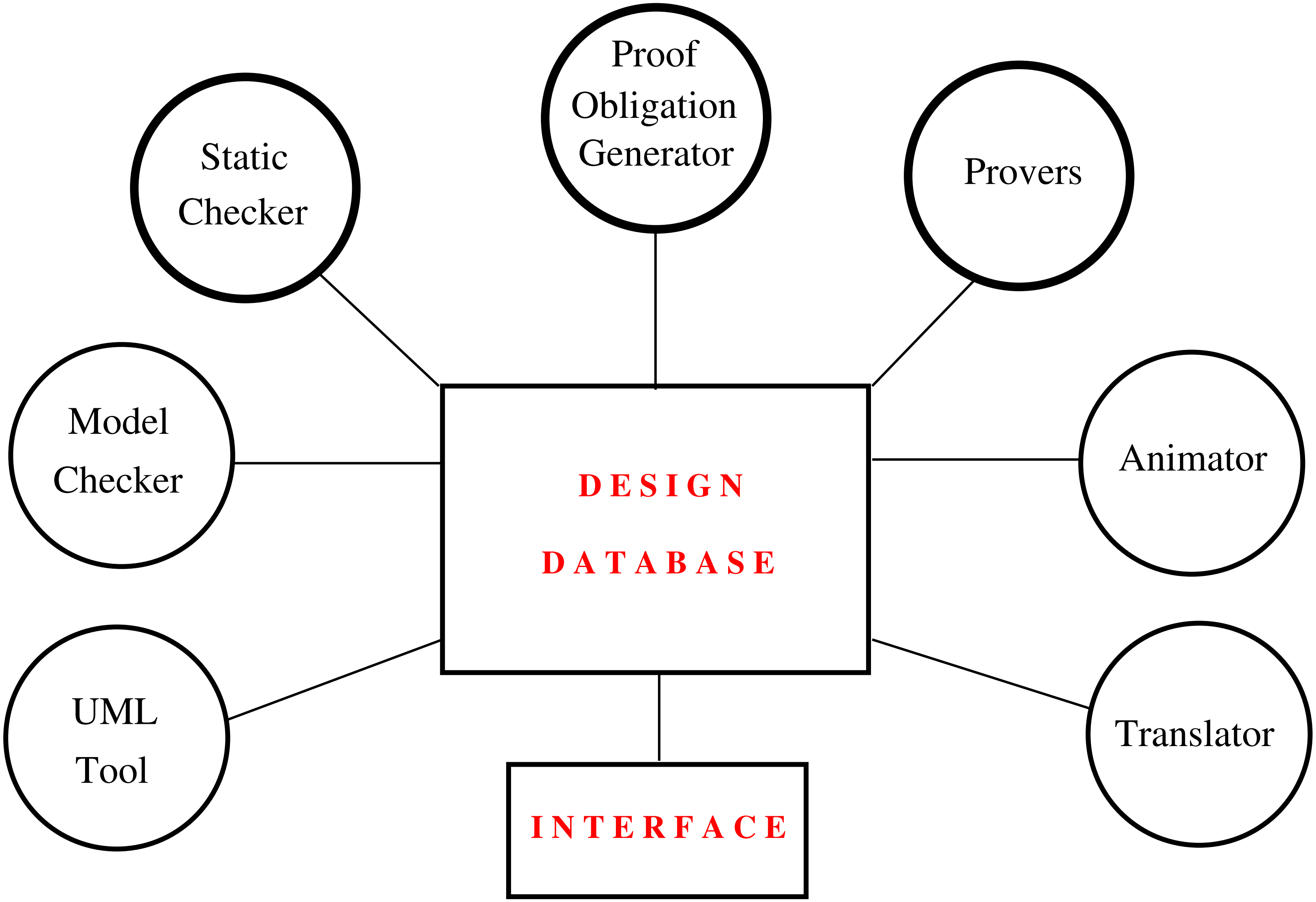
- It results in a very systematic formal development

- Many other patterns have to be developed

- More automation has to be provided (plug-in)

A Possible "Compiler" for Tomorrow

DESIGN AND VERIFICATION PLUG-INS



Thanks for Listening