Formal methods (with particular attention to concurrent and real-time systems)

Teacher: Dino Mandrioli

Course given in English

Shared by joint UIC-PdM master and normal PdM's Ingegneria Informatica curriculum

Course presentation

- Outline
- Organization

Contents

- Introduction and motivation (lot of motivation needed):
  - what are formal methods?
  - Objectives, problems, criticisms, …promises and reality, hopes and hypes, academia and industry, …
  - FMs in the context of sw (system!) life cycle
  - The international FM community (organizes conferences, etc)
- The basics of FMs: Hoare's approach to program specification and proof
- Coping with concurrent -distributed- and real-time systems: problems and approaches
- A short survey of main FMs for concurrent -and real-time- systems: Petri nets, CSP, timed automata, (timed) temporal logic, …
- Case study: the TRIO method
  - the TRIO language(s)
  - the tools
  - applying TRIO to industrial problems: experiences and case studies
- The dual language approach (hints)
- The refinement approach to system design and verification (hints)

Organization.1

- Class lectures
- Class exercises (with a TA)
- Class case studies and tool demos (with TAs)
- Homework(s)
  - Individual or team work
  - developing case studies
  - use of tools
  - producing documentation
- Final test:
  - paper and pencil
  - Based on the “conceptual parts” (Hoare, FMs for concurrent systems, …)
- Follow up for theses (both at UIC and PdM)

Organization.2

- We have a problem:
- We (UIC) must be finished by June 15!
- (unlike non-UIC courses)
- To allow for completing the homework in time advanced theoretical topics (dual-language approach and refinement) have been moved to the end
- With final test

- Minor organizational details:
  - Time table (classes: Tue 1.15 PM– 2.45 PM; Wed. 8.40 AM – 10.10 AM; later on meetings with the TA – Paola Spoletini and/or me)
  - Use of tools and lab: agree with Paola
  - Interact! Through e-mail for non-technical issues; face to face preferred for technical issues
  - Anything else? OK, this way?

Teaching material

- *Reference* books
  - Mandrioli/Ghezzi, Theoretical foundations …, J.Wiley
  - Heitmeyer/Mandrioli (eds), Formal methods for real time computing, Wiley
  - A.K. Cheng, Real-Time Systems; scheduling, …, J. Wiley, 2002
- Transparencies (.pdf) available on the web (my personal –italian- page)
- Scientific papers and reports (to be specified; possibly electronically available)
- (Public domain) tools

A check on your background

- UML
- Statechart
- Petri nets
- Model checking
- The Z language
- TRIO
- Theoretical Computer Science
- Mathematical logics

What are Formal methods?

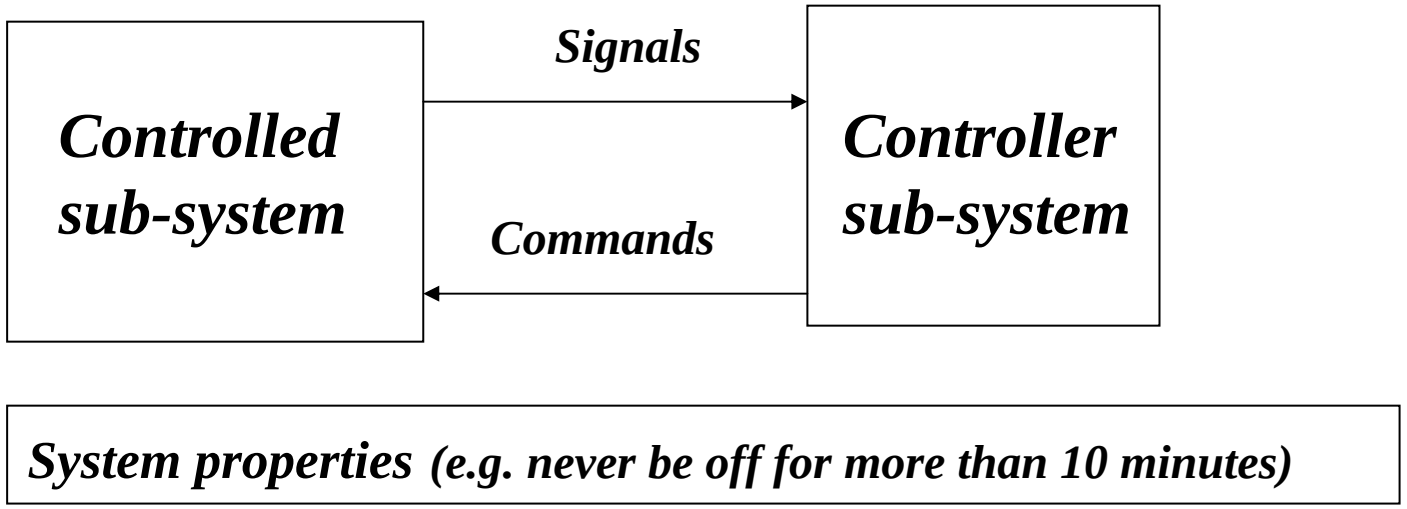
- Exploiting mathematical formalisms in the analysis and synthesis of -computer- systems
- Not a big deal in traditional engineering
- What is new/different in computer science/ SW engineering?
- Traditional mathematics is not well suited (lack of continuity, lack of linearity, …)
- Booelan algebra and other "basic discrete formalisms" are good only for low level hardware design; ---->
- In "normal" computer system design the programming language is the only -if any!- formalized medium:
- Most design documentation is "semiformal" -whatever such a term means …-
- Be aware of the difference between *(design) notation* and *design/development method*.

The problems with informal methods

- Lack of precision:
  - ambiguous definitions/specifications
  - erroneous interpretations
    - user/producer
    - specifier/designer
  - shaky verification (the well known problems with testing)
- ----->
- Unreliability
  - lack of safety/security
  - economic loss
- Lack of generality/reusability/portability
- ----->
- Poor quality (Even MS is realizing that …)

Formal methods: the ideal picture

- Everything is formalized: a mathematical model of everything is built ----->
- Every reasoning is based on mathematical analysis
- What is “everything”?
  - The system to be built
  - The system to be controlled
  - The (non-computer) to-be-controlled system / (computer-based) controller
  - The wished/feared properties (requirements)
  - …



- --------->
- “Everything” is mathematically certified
- Mathematical reasoning can be (partially) automatized
- ---->
- Increased reliability, ….
- Increased quality

What is wrong with the ideal picture?
Why FMs did not gain -so far- wide industrial adoption?

- Formal models are not reality:
  - of course; this is true everywhere;
  - certainly we need verify adequacy of formalization, no 100% guarantee in real cases, certainty is only mathematical but not real, …
  - nevertheless they have been often powerful design tools
  - mathematics is not guarantee of error freedom, but is much better than informal reasoning
- (Proposed) FMs require too much mathematical skill on the side of industrial people
  - yes, but this claim is pretty much "context-dependent" …
  - (we will go back to this issue several times)
- (Present) tools are not well-engineered/user-friendly, ….
  - True, but the situation may -and should- evolve

- If we look at most formal methods we actually find that they are much more of *formal models* than real *formal methods*.
  - A fashionable slogan (due to John Rushby?):
    "the problem with formal methods is that they are just *formal* not *methods*"
- This criticism is pretty much true and must be taken into account seriously
- Studying formal models is a basic prerequisite to apply formal methods but is not enough
- A common misunderstanding on FMs:
  - FMs = formal verification: absolutely not! They are much more. Formal verification is a -non necessary- part of a FM
- -------->

FMs in the context of (SW or system?) life cycle

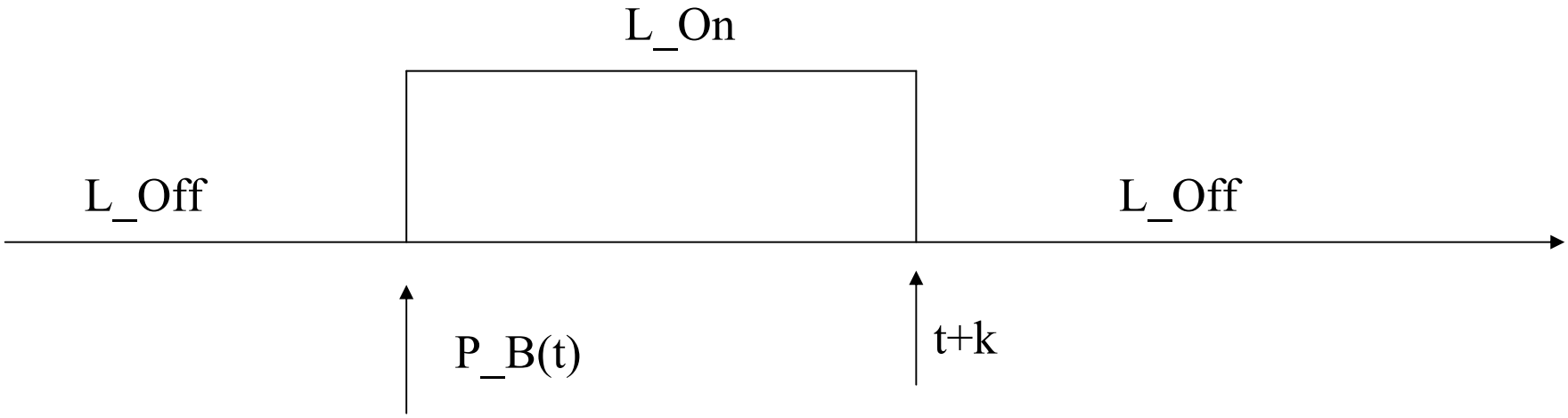
- All *systems* -including but not exclusively!- SW systems have their own life cycle
  - feasibility study,
  - requirements analysis and specification
  - design
  - verification,
  - …
  - maintenance
- their development exploits suitable *methods* (UML [not a real method …], IEC, VDM, Structured vs OO analysis and design, …)
- not all methods are exclusive … in some way or another they can be *integrated*

- FMs can be “pure” and/or complete (B, Z, VDM with their OO versions, …)
- Or they can be used as a “formal add-on” to be matched and integrated with pre-existing (and adopted) methods: “formal UML” TRIO-XXX/UML”, …
- Incrementality should play a major role in developing and introducing (Formally-based) Methods in industrial environments.
- Much more on this critical topic should be said -and a little more will be.
- Reference readings:
  - Ciapessoni et al. From formal models to .. Methods .. ACM TOSEM, January 1999.

Let us now (re)begin from the very basics ..

- Formalism categories:
  - operational formalisms or abstract machines (FSM, Petri nets, …)
  - descriptive formalisms (logic formulae, algebraic formalisms, …)
- Let us restart from the use of logic formulas to specify systems (properties) ...

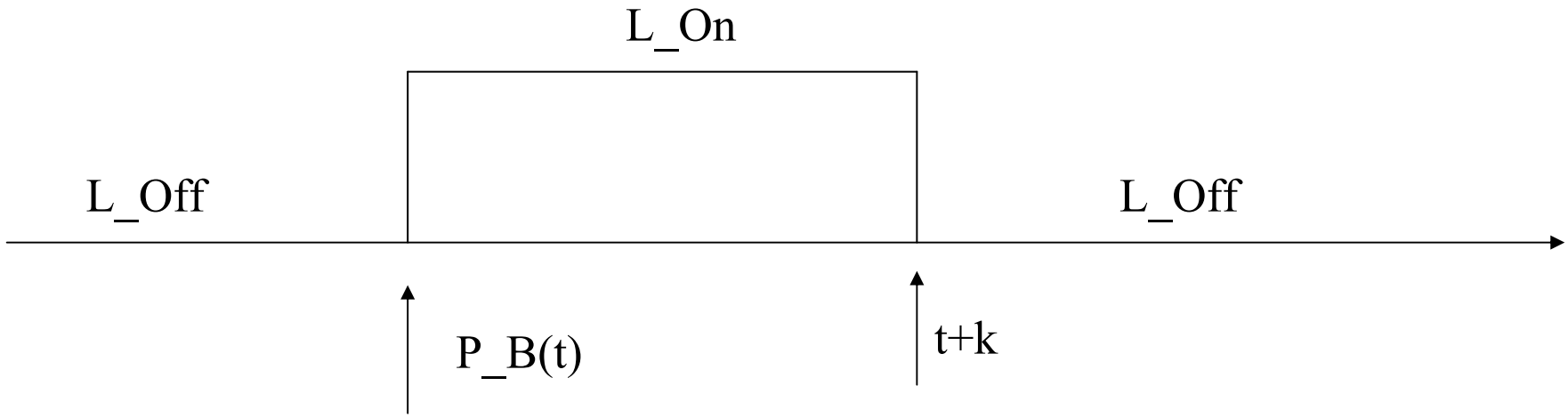
- Let us go back to an old (for the PdM students) example: the light timer:
- L_On: Light is on; P_B: push the button,



$$\forall t(P\_B(t) \rightarrow \forall t_1((t \le t_1 < t + k) \rightarrow L\_On(t_1)) \wedge$$
$$\forall t_2((t + k \le t_2) \rightarrow L\_Off(t_2))$$

- Actually this formalization leaves much to be desired …
  let's apply some error chasing ...

What about this one:

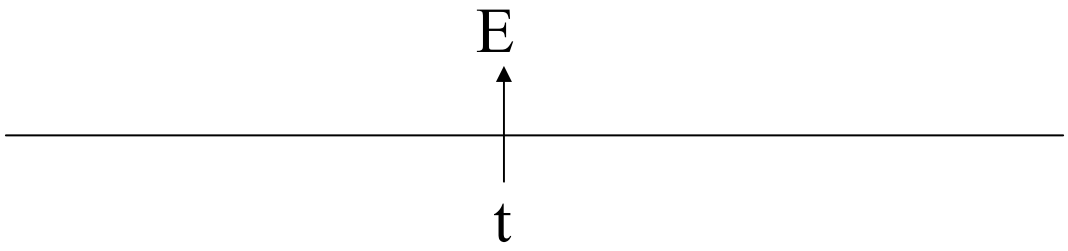


$$\forall t((P\_B(t) \wedge L\_Off(t)) \rightarrow$$
$$\forall t_1((t \le t_1 < t+k) \rightarrow L\_On(t_1)) \wedge L\_Off(t+k))$$
$$\wedge$$
$$\forall t_3, t_4(L\_Off(t_3) \wedge \forall t_5((t_3 \le t_5 \le t_4) \rightarrow \neg P\_B(t_5))) \rightarrow L\_Off(t_4)$$

A little more systematic approach (mainly for continuous time modeling)

- Event_E: short notation for the following:

$$\forall t(E(t) \rightarrow \exists \delta(\forall t_1(t-\delta < t_1 < t \vee t < t_1 < t+\delta) \rightarrow \neg E(t_1)))$$



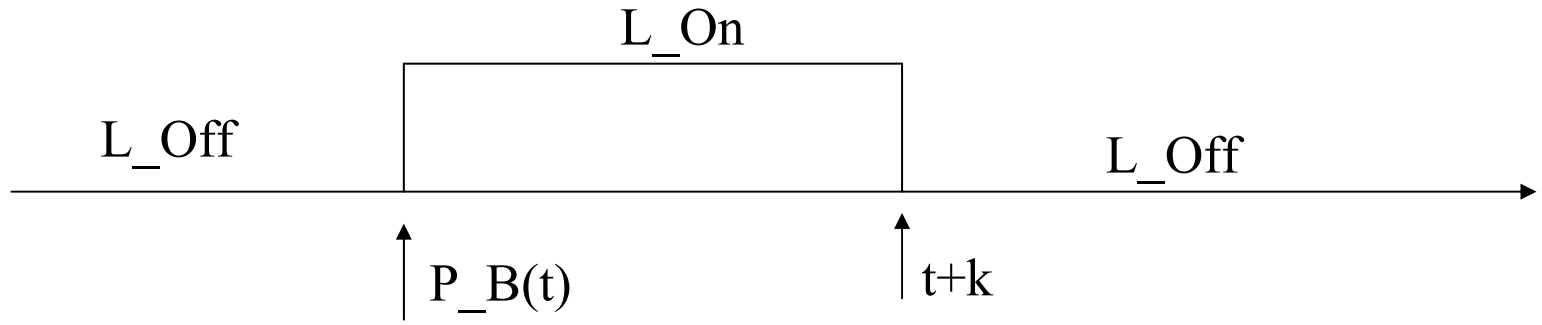
- Up_to_now_S(t): short notation for:

$$\exists \delta(\forall t_1(t-\delta < t_1 < t) \rightarrow S(t_1)) \qquad \text{(subsumed } : \delta > 0)$$

- From_now_on_S(t): short notation for:

$$\exists \delta(\forall t_1(t \leq t_1 < t+\delta) \rightarrow S(t_1))$$

Going back to our timer:



$$\forall t(L\_On(t) \leftrightarrow \neg L\_Off(t)) \quad \text{(we can argue about that)} \quad \wedge$$

$$Event\_P\_B \quad \wedge$$

$$\forall t((P\_B(t) \wedge Up\_to\_now\_L\_Off(t)) \rightarrow$$

$$\forall t_1((t \le t_1 < t+k) \rightarrow L\_On(t_1)) \wedge L\_Off(t+k)) \quad \wedge$$

$$\forall t_3, t_4(L\_Off(t_3) \wedge \forall t_5((t_3 \le t_5 \le t_4) \rightarrow \neg P\_B(t_5))) \rightarrow L\_Off(t_4)$$

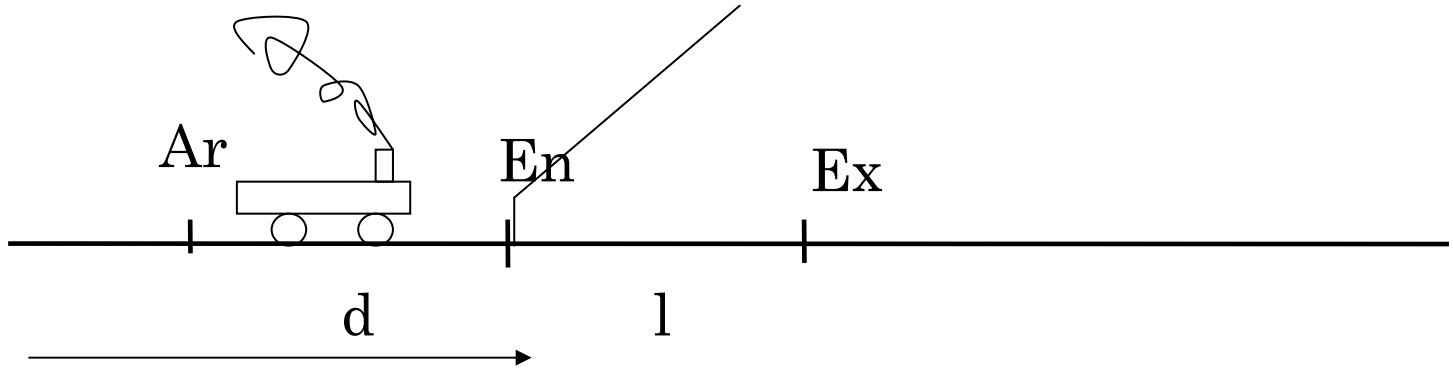
Several lessons can already be derived from this first little example

- Pretty much work is already necessary -and far from trivial- just for *specifying* -i.e., *defining*- system properties
- Using formulas is no guarantee to avoid mistakes ...
- But it helps understanding and fixing errors
- We need *methods* (guidelines) to learn how to write good specifications whether formal or not
- Formulas are the *result* of our activity
- All above work -whether formal or not:
- *Requirements elicitation*, analysis and specification: a typical and critical phase of system development
- The activity is in the domain of SW (system) engineering;
- But exploiting formalism may help a lot (even more with the help of tools):
  - informal statement;
  - sample behavior
  - tentative formula
  - checking for consistency
  - modifying and retrying ...

Logic formulas and deductions to *verify* system design

Let us consider a railroad crossing system. Such a system is widely used in the literature as an interesting benchmark for rigorous (formal) analysis methods

For the time being we consider a highly simplified version thereof:



Only one track and only one train.
Instantaneous opening and closing of the gate.

- Event_Ar
- Event_En
- Event_Ex
- State_In

First, we formalize the "controlled system" behavior (α):

$$(\delta_1 = d/V_{\min} \wedge \delta_2 = \frac{d+l}{V_{\max}}) \wedge (\delta_3 = l/V_{\min}) \wedge (\delta_4 = l/V_{\max}) \wedge (\delta_{\min} = d/V_{\max}) \wedge (\delta_{\max} = \frac{d+l}{V_{\min}}) \wedge$$

$$\forall t(Ar(t) \rightarrow \exists t_1(En(t_1) \wedge (t+\delta_{\min} \leq t_1 \leq t+\delta_1)) \wedge$$

$$\forall t(En(t) \rightarrow \exists t_1(Ar(t_1) \wedge (t-\delta_1 \leq t_1 \leq t-\delta_{\min}))) \wedge$$

$$\forall t(En(t) \rightarrow \exists t_1(Ex(t_1) \wedge (t+\delta_4 \leq t_1 \leq t+\delta_3))) \wedge$$

$$\forall t(Ex(t) \rightarrow \exists t_1(Ar(t_1) \wedge (t-\delta_{\max} \leq t_1 \leq t-\delta_2))) \wedge$$

$$\forall t(In(t) \leftrightarrow \exists t_1(En(t_1) \wedge (t_1 \leq t)) \wedge \neg\exists t_2(Ex(t_2) \wedge (t_2 \leq t))$$

Then, we "design the controller", i.e., the gate behavior (β):

$$\forall t(Ar(t) \rightarrow \forall t_1((t+\delta_{\min} \leq t_1 \leq t+\delta_{\max}) \rightarrow Down(t_1))) \wedge$$

$$\forall t(Down(t) \rightarrow \exists t_1((t-\delta_{\max} \leq t_1 \leq t-\delta_{\min}) \wedge Ar(t_1))) \wedge$$

$$\forall t(Down(t) \leftrightarrow \neg Up(t))$$

Third, we specify the safety requirement (γ):

$$\forall t(In(t) \rightarrow Down(t))$$

Remark: such a (safety) requirement is not a complete enough spec of what we want/need form a railroad crossing system (why?). But for the moment we focus on this fundamental requirement.

At this point we are ready to *prove the correctness* of our design, i.e.:

First we obtain the following from the fourth and sixth row of α:

$$In(t) \rightarrow \exists t_1 (En(t_1) \wedge (t - \delta_3 < t_1 \leq t)$$

In fact In(t) implies

$$In(t) \rightarrow \exists t_1 (En(t_1) \wedge (t_1 \leq t))$$

$$En(t_1) \rightarrow \exists t_2 (Ex(t_2) \wedge (t_1 + \delta_4 \leq t_2 \leq t_1 + \delta_3))$$

Thus, $t < t_2$ and therefore $t - \delta_3 < t_1$

(From the third row of α), the consequent of the above formula, in turn, implies:

$$\exists t_2 \left(Ar(t_2) \wedge (t - \delta_1 - \delta_3 \leq t_2 \leq t - \delta_{\min}) \right)$$

Then, thanks to the inequalities

$$\delta_{\min} \leq \delta_1 + \delta_3 = \delta_{\max}$$

and from the first row of β, we obtain the thesis

Again, we can derive several lessons from this first little example

- We have obtained -thanks to mathematical formalization- a correctness proof as opposed to system testing: a successful test (running just one train at a given speed) is no guarantee of complete correctness ---->
- Typical folklore about “testing can only show the presence of errors, not their absence” as opposed to mathematical certainty coming form a mathematical theorem
- However:
  - not so trivial proof for a very trivial property of a very trivial system
  - what about real, complex cases?
  - (we will see that just a bit of generalization makes things much less trivial)
  - how much mathematical skill needed? (probably more than what is needed just for specifying)
  - the more details, the more chances for errors,
  - ….
- We must keep in mind that we are just laying the foundations of -hopefully- practically useful formal methods: this first example should be intended as the first programming examples of, say, Euclid algorithm, possibly even in machine language.
- Much support can -and should- be obtained from support tools (despite we are dealing with undecidable problems, … at least in most cases)

Let us now focus on *program correctness*: the classical Hoare's method

- Let us recall *program specification*:
- {Precondition: *Pre*}
  Program - or program *fragment- P*
  {Post-condition: *Post*}
  If *Pre* holds before *P's* execution then, after *P's* execution *Post* must hold:
- Example: search in an ordered array:

$$\{\forall i(1 \le i \le n) \rightarrow a[i] \le a[i+1]\}$$

$$P$$

$$\{\ found \ \leftrightarrow \exists i(1 \le i \le n \wedge a[i] = x)\}$$

In analogy with the railroad crossing example:

- We build a specification *Spec*:
- $\{Pre\}$
  P
  $\{Post\}$
- We design an implementation (program) P
- We want to prove that P's semantics implies *Spec*.
- To reduce the whole process to a logic deduction we need:
  - A syntax that embodies program and their properties (as first-order formulas)
  - An axiom system that embodies program semantics, i.e., allows to deduce properties on program variable (values) that are consequence of program statements execution (language semantics axiomatization)
- We will do this for a typical "MiniPascal" language

- The method is a typical syntax-directed, compositional method, as it is often the case with structured Algol-like languages (the same applies to operational and denotational semantics, to translation schemas, to compiler testing, etc.)
- Thus, let us begin with the kernel of any -von Neumann- language: the assignment statement

The axiomatic semantics of the assignment statement

- Let us go backwards:
  - If we execute x := 0 we are sure that after the execution the condition x = 0 holds
  - This allows us to derive the axiom
    {T} x := 0 {x = 0}
- A little more generally:
  - if we want that after execution of x := a, the postcondition Post(x) ≡ x > y + z holds a necessary and sufficient precondition to achieve such a goal is that Pre(a) ≡ a > y + z holds before statement execution

- Even more generally:
  - Let x := exp be a generic assignment statement
  - Let Post be a generic (first-order) formula
  - Let $Post_x^{exp}$ denote the result of substituting any (free…) occurrence of x in Post by exp
  - then
  - $Post_x^{exp}$ is a necessary and sufficient precondition for guaranteeing the condition Post.
  - We write the backwards substitution *axiom*
  - A1) $\{Post_x^{exp}\}\ x := exp\ \{Post\}$
  - Thanks to A1, whenever we can rely on that a condition such as $\{Post_x^{exp}\}$ holds before executing x := exp, we can also rely on the holding of Post after its execution.

- Let us now consider the sequence of two statements such as x := y; z := x
  - By applying A1 twice we have, say,
    - {x > 0} z := x {z > 0} and {y > 0} x := y {x > 0}, i.e.,
    - if y is > 0 before execution on x := y then we can rely on x > 0 after its execution; but also if x is > 0 before execution on z := x then we can rely on z > 0 after its execution; thus
    - if y is > 0 before execution on x := y followed by z := x then we can rely on z > 0 after execution of the sequence of the two statements
  - More generally, we can apply the following *inference rule (IR1)*:
  - Let S1 and S2 be two program fragments, then

$$\frac{\{P\}S_1\{R\},\ \{R\}S_2\{Q\}}{\{P\}S_1;S_2\{Q\}}$$

Where the notation $\frac{F_1,\ F_2, F_3, \ldots F_n}{F}$

Means that if we have proven F_1, F_2, … and F_n, then we can deduce F

- The following rule (IR2) can also be useful

$$\frac{P_1 \to P,\ \{P\}\ S\ \{Q\}, Q \to Q_1}{\{P_1\}\ S\ \{Q_1\}}$$

Notice that P_1 is a sufficient (but not necessary) precondition to guarantee Q_1 as a postcondition of S

•From what we have up to now we are able to prove facts such as:

$$\{f(g(x)) > 5\}\ x := g(x);\ x := f(x)\ \{x > 0\}$$

The semantics of sequencing two pieces of programs is the composition of the two respective semantics

The axiomatization of the **if-then[-else]** statement

(*IRCa*)
$$\frac{\{P \wedge c\}S_1\{Q\}, \{P \wedge \neg c\}S_2\{Q\}}{\{P\}\ \textbf{if}\ c\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}}$$

(*IRCb*)
$$\frac{\{P \wedge c\}S_1\{Q\}, \{P \wedge \neg c\} \rightarrow \{Q\}}{\{P\}\ \textbf{if}\ c\ \textbf{then}\ S_1\ \{Q\}}$$

Example

- We wish to prove that under *any circumstance*, the assertion

  $z \geq y \wedge z \geq x$ holds as a consequence of executing
  if $x > y$ **then** $z := x$ **else** $z := y$, i.e.,

- $\{T\}$ **if** $x > y$ **then** $z := x$ **else** $z := y$ $\{z \geq y \wedge z \geq x\}$
- [Questions:
  - Is this a specification for the computation of the maximum between two numbers?
  - Is this a statement that computes the maximum between two numbers?]
  - $T \wedge x > y \rightarrow x \geq y \wedge x \geq x$
  - $\{x \geq y \wedge x \geq x\}\ z := x\ \{z \geq y \wedge z \geq x\}$ thus, by IR2,
  - $\{T \wedge x > y\}\ z := x\ \{z \geq y \wedge z \geq x\}$
- Similarly
  - $T \wedge \neg(x > y) \rightarrow y \geq y \wedge y \geq x$
  - $\{y \geq y \wedge y \geq x\}\ z := y\ \{z \geq y \wedge z \geq x\}$ and, by IR2,
  - $\{T \wedge \neg(x > y)\}\ z := y\ \{z \geq y \wedge z \geq x\}$
- Finally, the desired result is deduced by IRCa

- We are now able to “prove” simple program fragments built by means of:
  - assignments
  - sequencing
  - conditional statements
- The rules are compositional, i.e. they can be inductively applied following the structure of the program (fragment)
- However, we know that the heart of most algorithms consists of iteration (or recursion):

$$(IRW) \qquad \frac{\{I \wedge c\}\ S\ \{I\}}{\{I\}\ \textbf{while}\ c\ \textbf{do}\ S\ \{I \wedge \neg c\}}$$

I is called the (better: *a*) loop invariant

IRW states that if I holds as a precondition of the loop, then at loop exit (if ever! …) I and the negation of loop condition hold.

How do we use *IRW*???

A first "complete" example

(let's forget about I/O, which anyway can be treated in a trivial way)

- {y>0}
  begin
  x:= 1; j :=1;
  while y ≥ j
  do
  j := j + 1; x := x *z
  end
  end
  {x = z^y}

- The method suggests to go "backwards":
- We start from the postcondition and we try to build assertions {Ai} such that
- {Pre} Fr1 {A1}, {A1} Fr2 {A2} …. Frn {Post}

In our case:

- {y>0}
  begin
  x:= 1; j :=1;
- {I?}
  while y ≥ j
  do
  j := j + 1; x := x *z
  end
  end
  {x = z^y}

- We need an *I* such that
- {y>0} x:= 1; j :=1 {I}
  and
- {I} **while** y ≥ j **do** j := j + 1; x := x *z **end** {I ∧ y < j} → **{x = z^y}**

- Here we are faced with the main problem: we must invent a suitable invariant!
- Let's first proceed in a "blind" way: I1 $\equiv$ Post $\equiv \mathbf{x} = \mathbf{z}^y$: By this way we would immediately have $(I1 \wedge \neg c) \rightarrow$ Post and we would be finished.
- Unfortunately, it is immediately clear that Post is not a loop invariant
  - In fact we should prove $\{\mathbf{x} = \mathbf{z}^y \wedge y \geq j\}$ j := j + 1; x := x *z $\{\mathbf{x} = \mathbf{z}^y\}$, i.e., $\mathbf{x} = \mathbf{z}^y \wedge y \geq j \rightarrow x * z = \mathbf{z}^y$ which is obviously false.
- This failure shows a major point: we cannot proceed fully automatically (remember Rice's theorem!): we must use our *ingenuity* to invent a suitable invariant. This simply means analyzing our program just as we do with "normal" informal analysis, but formulating our analysis through the mathematical notation.
- The notion of loop invariant is not just a matter of mathematical formalism: it is a way of capturing the essence of the loop: transforming program state by keeping a fundamental property "invariant" and extending its scope up to "cover" all necessary cases (a whole array or list, a set of candidate elements where to look for a searched one, etc.)
- In our case the initialization and the loop body suggest a natural relation between the counter j and the exponent of z (in x):

- Let us try I2 ≡ $\mathbf{x = z^j}$:

  This is certainly a loop invariant: $\mathbf{x = z^j} \wedge y \geq j \rightarrow x * z = \mathbf{z^{j+1}}$;
  but $\mathbf{x = z^j} \wedge y < j$ does not imply $\mathbf{x = z^y}$ (in fact at the beginning j = 1 and x = $\mathbf{z^0}$.) Thus,

- I3 ≡ $\mathbf{x = z^{j-1}}$ seems more appropriate. I3 is invariant.
  I3 $\wedge$ y < j does not contradict $\mathbf{x = z^y}$ but it does not imply it as well.
  ---> It is good but "not enough": too "weak": we need info to derive the Post.
- Finally we augment I3 as I4 ≡ $\mathbf{x = z^{j-1}} \wedge y \geq j - 1$. I4 is loop invariant too.
- This time however, I4 $\wedge$ y < j does imply $\mathbf{x = z^y}$.
- We are almost done: it remains only to verify that the invariant is stated by the initialization:
- {y > 0} x := 1; j := 1{$\mathbf{x = z^{j-1}} \wedge y \geq j - 1$}, i.e. $y > 0 \rightarrow y \geq 0$

- and the proof is really complete!

Handling quantified assertions

- Consider the following example, derived by a "blind application" of axiom A1:
  $F \equiv \{\exists\ z(z=2.z \wedge z > 0)\}\ x := z\ \{\exists\ z(x=2.z \wedge z > 0)\}$
  Intuitively, the retropropagation should produce the precondition that guarantees that, after executing x := z, x is an even number.
- The failure is due to the fact that *z is not free for x* in $\{\exists\ z(x=2.z) \wedge z > 0\}$ (for the same reason we cannot freely substitute x by y in $\forall x \exists y(A(x,y))$).
- The example, however, also shows a major difference between the meaning of variables such as z, which is quantified and variables such as x, which is not: here z denotes a generic value, whereas x denotes a program variable, i.e. an abstraction of a memory cell: two sharply different concepts are expressed by the same notation!
- To avoid confusion, in the following we will use overbarred identifiers to denote quantified variables (which are *not* program variables) and normal ones for program variables. In the above example we should rewrite:

$$\{\exists\ \bar{z}\ (z = 2.\bar{z} \wedge \bar{z} > 0)\ \}\ x := z\ \{\exists\ \bar{z}(x = 2.\bar{z} \wedge \bar{z} > 0)\ \}$$

which makes perfectly sense.

- As a further example, suppose you wish to state that, after execution of a program fragment S variable z has the maximum value among all other program variables.
- At a first glance, one might try $S\{\forall \overline{y}(z \geq \overline{y})\}$.
- However, we would obtain

  $\{\forall \overline{y}\,(z \geq \overline{y})\}\; x := z + 1\; \{\forall \overline{y}\,(z \geq \overline{y})\}$

  which is clearly wrong.
- The error, again, abides in a confusion between variables denoting just values and program variables:
- the above formula actually states that “the value of -the contents of- variable z” is greater than or equal to any *integer value*, which is clearly unsatisfiable.
- The above informal requirement should have been formalized as:

  $S\;\{\forall \overline{y}\,(\overline{y} = x_1 \vee \overline{y} = x_2 \vee \ldots \rightarrow (z \geq \overline{y})\}$

  where x_i denotes a generic program variable identifier.

- This time, backwards substitution through x := z + 1 would -correctly yield False (in fact, there is no way that after executing x := z + 1, z will be the greatest program variable.)

A new example: Euclid's algorithm

- Let us consider the classic Euclid's algorithm to compute the greatest common divisor (GCD) of two non-zero natural numbers. As usual, let us "fake" I/O for simplicity:
- PGCD:
- **begin**
  - x := z; y := w;
  - **while** x ≠ y **do**
    - **if** x > y **then** x := x - y **else** y := y - x **end**
  - **end**
- **end**
- Its specification is {z > 0 ∧ w > 0} PGCD {x = GCD(z,w)}, where we use x = GCD(z,w) as a short notation for the formula:

$$\exists k\left((k=x)\wedge\left(\exists u_1,v_1\left(z=u_1\cdot k\wedge w=v_1\cdot k\right)\right)\wedge\left(\forall h,u_2,v_2\left(z=u_2\cdot h\wedge w=v_2\cdot h\wedge h>k\right)\right)\right)$$

- Our goal: find a suitable I such that:

- {Pre} = {z > 0 ∧ w > 0}
- **begin**
  - x := z; y := w; {I}
  - **while** x ≠ y **do**
    - **if** x > y **then** x := x - y **else** y := y - x **end**
  - **end** **{I ∧ x = y}**
- **end**
- {Post} = {x = GCD(z,w)},

- Let us focus immediately on the loop invariant: from our informal knowledge of Euclid's algorithm, a natural one appears to be
  I: GCD(x, y) = GCD(z, w), which, as before, it is intended as a short notation for:

$$\exists \overline{k}\left(\begin{array}{l}\left(\exists \overline{x_1}, \overline{y_1}\left(x = \overline{x_1}\cdot\overline{k} \wedge y = \overline{y_1}\cdot\overline{k}\right) \wedge \nexists\, \overline{h}, \overline{x_2}, \overline{y_2}\left(x = \overline{x_2}\cdot\overline{h} \wedge y = \overline{y_2}\cdot\overline{h} \wedge \overline{h} > \overline{k}\right)\right) \wedge \\ \left(\exists \overline{z_1}, \overline{w_1}\left(z = \overline{z_1}\cdot\overline{k} \wedge w = \overline{w_1}\cdot\overline{k}\right) \wedge \nexists\, \overline{h}, \overline{z_2}, \overline{w_2}\left(z = \overline{z_2}\cdot\overline{h} \wedge w = \overline{w_2}\cdot\overline{h} \wedge \overline{h} > \overline{k}\right)\right)\end{array}\right)$$

- First step:
  $\{z > 0 \wedge w > 0\}$ x := z; y := w $\{I\}$, i.e., GCD(z, w) = GCD(z, w)
  which is obviously true.
- Second step:
  $I \wedge x = y$ implies x = GCD (z, w), which is also true in a fairly obvious way (in fact x = y implies $\overline{k}$ = x = y).
- It remains to be proved that I is indeed a loop invariant, i.e.,
  $\{I \wedge x \neq y\}$ **if** x > y **then** x := x - y **else** y := y -x **end** $\{I\}$

- The proof of {I ∧ x ≠ y} **if** x > y **then** x := x - y **else** y := y -x **end** {I} in turn is split into:
  - I ∧ x > y → GCD(x-y, y) = GCD(z, w)
  - I ∧ y > x → GCD(x, y - x) = GCD(z, w)
  - Then, since the loop condition x ≠ y implies (x > y) ∨ (y > x) (and therefore (x ≠ y ∧ ¬ (x > y)) → y > x) the proof will be complete.
- Le us prove the first conjunct (the second one being symmetric).
- First, notice that, for any $\overline{k}$:

$$x > y \wedge \exists \overline{x_1}, \overline{y_1}\left(x = \overline{x_1} \cdot \overline{k} \wedge y = \overline{y_1} \cdot \overline{k}\right) \rightarrow x - y > 0 \wedge \exists \overline{x_1}, \overline{y_1}\left(x - y = \left(\overline{x_1} - \overline{y_1}\right) \cdot \overline{k}\right)$$

$$\text{Thus}, \exists\, \overline{x_1}, \overline{y_1}\left(x - y = \overline{x_1} \cdot \overline{k} \wedge y = \overline{y_1} \cdot \overline{k}\right)$$

I.e., $\overline{k}$ is also a common divisor of x -y and y

- Now, suppose that, $\bar{k}$ is not the GCD of the twos. Then:

$$\exists \bar{h}, \bar{\bar{x}}_2, \bar{\bar{y}}_2 \left(\bar{h} > \bar{k} \wedge x - y = \bar{\bar{x}}_2 \cdot \bar{h} \wedge y = \bar{\bar{y}}_2 \cdot \bar{h} \right)$$

This would imply

$$x = \bar{\bar{x}}_2 \cdot \bar{h} + \bar{\bar{y}}_2 \cdot \bar{h} = \left(\bar{\bar{x}}_2 + \bar{\bar{y}}_2 \right) \cdot \bar{h} \wedge y = \bar{\bar{y}}_2 \cdot \bar{h}$$ and thus

$$\exists \bar{h}, \overline{x_2}, \overline{y_2} \left(\bar{h} > \bar{k} \wedge x = \overline{x_2} \cdot \bar{h} \wedge y = \overline{y_2} \cdot \bar{h} \right)$$

thus contradicting I.

Handling indexed variables

- Let us consider the backwards substitution axiom when formulas and statements involve the use of indexed variables (or pointers -which is conceptually similar).
- A first example such as
  {T} a[x] := 5 {a[x] = 5}
  seems to suggest that the axiom applies as well in this case too …
- … however, at a deeper inspection:

  {a[x] = 3} a[k] := 5{a[x] = 3 $\wedge$ a[k] = 5}
  is not so correct: what happens if k = x? Other similar examples are:
  - Let P be $\forall \bar{i}((1 \le \bar{i} \le 10) \rightarrow a[\bar{i}] = 5)$. Then we would obtain
    {P} k := 3; a[k] := 6 {P}, which is clearly incorrect.
  - Let P be a[x] = 1 $\wedge$ x = a[2]. Then
    {T} x := a[2]; a[x] := 1 {P} is falsified if a[2] = 2
- In general, the source of the problems is the *aliasing* induced by indexed variables (as well as pointers): *different symbols mean the same variable*. Thus, hidden effects are produced by assignments to given -variable-identifiers.

- To overcome this difficulty aliasing must be explicitly taken into account during backwards substitution. This leads to the following reformulation of the main axiom:

$\overline{A1})$ $\quad \{\overline{P}^{t}_{a[i]}\}\ a[i]:=t\ \{P\}$

where $\overline{P}^{t}_{a[i]}$ is the result of replacing any occurrence of a[x] in P by the expression

if $x = i$ **then** t **else** $a[x]$

- For instance, if

$$P = \forall \overline{p}\left(\left(a\left[\overline{p}\right] \geq a[i]\right) \rightarrow \left(\forall \overline{z}\left(a\left[\overline{z}\right] \neq a[j]\right)\right)\right)$$

- Then $\overline{P}^{t}_{a[i]}$ is:

$$\overline{P^{t}_{a[t]}} = \forall \overline{p}\left(\begin{array}{l}\left(\textbf{if } \overline{p} = i \textbf{ then t else } a\left[\overline{p}\right]\right) \geq t \rightarrow \\ \qquad \left(\forall \overline{z}\left(\textbf{if } \overline{z} = i \textbf{ then } t \textbf{ else } a\left[\overline{z}\right]\right) \neq \left(\textbf{if } \ j = i \textbf{ then } t \textbf{ else } a[j]\right)\right)\end{array}\right)$$

A nontrivial example: a sorting algorithm and its correctness

- (Let us recall) The specification of the sorting problem:

$$\{1 \le n \wedge \forall \bar{p}((1 \le \bar{p} \le n) \to a[\bar{p}] = b[\bar{p}])\}$$

  - PSORT

$$\left\{\begin{array}{l}(\forall \bar{p}(1 \le \bar{p} \le n-1 \to (a[\bar{p}] \le a[\bar{p}+1]))) \wedge \\ (\forall \bar{p}(1 \le \bar{p} \le n) \to (\exists \bar{r}(1 \le \bar{r} \le n) \wedge (a[\bar{p}] = b[\bar{r}]))) \wedge (\forall \bar{r}(1 \le \bar{r} \le n) \to (\exists \bar{p}(1 \le \bar{p} \le n) \wedge (a[\bar{p}] = b[\bar{r}])))\end{array}\right\}$$

- A sorting algorithm:
- **begin**
  - i := 2;
  - **while** i <= n **do**
    - x := a[i]; a[0] := x; j := i -1;
    - **while** x < a[j] **do** a[j+1] := a[j]; j := j -1 **end**
    - a[j+1] := x; i := i + 1
  - **end**
- **end**

Let us now face the proof of the algorithm w.r.t. the above specification

- This time, the job is far from trivial. Thus, as usual with complex problems, we need to proceed judiciously, by applying some suitable *method*. In particular, proofs can and should be *modularized* as well as every other complex artifacts.
- Let us first observe that, if we have a specification of the type
  {Pre1 $\wedge$ Pre2} Prog {Post1 $\wedge$ Post2}
  and we succeed in proving separately
  {Pre1} Prog {Post1} and {Pre2} Prog {Post2}
  then we can deduce the proof of the whole specification.
- In our case let us therefore try to prove separately

A

$$\{1 \le n\}$$

PSORT

$$\left\{\left(\forall \bar{p}\left(1 \le \bar{p} \le n-1 \rightarrow \left(a\left[\bar{p}\right] \le a\left[\bar{p}+1\right]\right)\right)\right)\right\}$$

B

$$\left\{\forall \bar{p}\left(\left(1 \le \bar{p} \le n\right) \rightarrow a\left[\bar{p}\right] = b\left[\bar{p}\right]\right)\right\}$$

PSORT

$$\left\{\begin{array}{l}\left(\forall \bar{p}\left(1 \le \bar{r} \le n\right) \rightarrow \left(\exists \bar{r}\left(1 \le \bar{r} \le n\right) \wedge \left(a\left[\bar{p}\right] = b\left[\bar{r}\right]\right)\right)\right) \wedge \\ \left(\forall \bar{r}\left(1 \le \bar{r} \le n\right) \rightarrow \left(\exists \bar{p}\left(1 \le \bar{p} \le n\right) \wedge \left(a\left[\bar{p}\right] = b\left[\bar{r}\right]\right)\right)\right)\end{array}\right\}$$

Let us give here the proof of A, leaving the proof of B as an exercise

- This time our algorithm consists of two nested loops; thus we will need two invariants. Let us begin with the external loop, as natural.
- A first informal analysis easily shows that the goal of the external loop is to partition the array a into two parts: the "lower" a[1..i-1] is sorted; then by increasing i from 2 (with i = 2, a[1..i-1] is sorted initially in a trivial way) to n+1 the whole array will be sorted. Let us translate this first informal analysis into an invariant for the external loop:

$$I_{EXT} : 1 \le i \le n+1 \wedge \forall \overline{p}\left(\left(1 \le \overline{p} < i-1\right) \rightarrow a\left[\overline{p}\right] \le a\left[\overline{p}+1\right]\right)$$

- Now, it is immediate to verify that $I_{EXT} \wedge i > n$ imply Post1, i.e., $\left\{\left(\forall \overline{p}\left(1 \le \overline{p} \le n-1 \rightarrow \left(a\left[\overline{p}\right] \le a\left[\overline{p}+1\right]\right)\right)\right)\right\}$
- Also immediate is the proof of $\{1 \le n\}$ i := 2 $\{I_{EXT}\}$
- It remains to be proved that I_{EXT} is indeed invariant

Proof of I_{EXT}'s invariance

- First, we backwards propagate through a[j+1] := x; i := i+1, obtaining

$$I^*_{EXT} : (0 \le i \le n) \wedge \forall \overline{p} \left(\begin{array}{l} (1 \le \overline{p} < i) \rightarrow \\ (\textbf{if } \overline{p} = j+1 \textbf{ then } x \textbf{ else } a[\overline{p}]) \le (\textbf{if } \overline{p}+1 = j+1 \textbf{ then } x \textbf{ else } a[\overline{p}+1]) \end{array} \right)$$

Notice that we applied the enriched axiom, to go through the assignment to a[j+1]

$I_{EXT}{}^*$ can be rewritten into the equivalent and more readable form:

$$(0 \le i \le n) \wedge \forall \overline{p} \left(\begin{array}{l} (1 \le \overline{p} < i) \rightarrow \\ \textbf{if } \overline{p} = j+1 \textbf{ then if } \overline{p}+1 = j+1 \textbf{ then } x \le x \textbf{ else } x \le a[\overline{p}+1] \\ \qquad \textbf{else if } \overline{p}+1 = j+1 \textbf{ then } a[\overline{p}] \le x \textbf{ else } a[\overline{p}] \le a[\overline{p}+1] \end{array} \right)$$

and then simplified into:

$$I^*_{EXT} : 0 \le i \le n \wedge \forall \overline{p} (1 \le \overline{p} < i) \rightarrow \textbf{if } \overline{p} = j+1 \textbf{ then } x \le a[\overline{p}+1]$$
$$\textbf{else if } \overline{p} = j \textbf{ then } a[\overline{p}] \le x$$
$$\textbf{else } a[\overline{p}] \le a[\overline{p}+1]$$

since $\overline{p} = j+1$ implies $\overline{p}+1 \ne j+1$

- Now, we "hit" the inner loop: thus, we need an invariant for it.
  As usual, let us first analyze informally the rationale of the loop:
  Its purpose is to create a "hole" for a[i] to be inserted in the appropriate position. This is obtained by shifting elements of the lower part of a. Thus, a[i] must be saved (in x). Thus, the inner loop further partitions the lower subarray a[1..i-1] plus a[i] into two sorted subarrays a[1..j] and a[j+1..i] such that each element of the latter is ≥ x, which in turn is equal to the "sentinel" a[0]. Furthermore a[j] ≤ a[j+1], except when j = i-1. All this is formalized as:

$$I_{IN} : \forall \overline{p}\left(\begin{array}{l}\left(\left(1 \le \overline{p} < j\right) \rightarrow a\left[\overline{p}\right] \le a\left[\overline{p}+1\right]\right) \wedge \\ \left(\left(j+1 \le \overline{p} < i\right) \rightarrow a\left[\overline{p}\right] \le a\left[\overline{p}+1\right]\right)\end{array}\right) \wedge$$
$$x \le a[j+1] \wedge 0 \le j < i \wedge x = a[0] \wedge \left(j < i-1 \rightarrow a[j] \le a[j+1]\right)$$

- At this point, to complete our proof, we need to prove:
  - The invariance of I_{IN}
  - That $I_{IN} \wedge x \ge a[j] \rightarrow I_{EXT}^{*}$
  - That $I_{EXT} \wedge i \le n \rightarrow$ the result of backwards substitution of I_{IN} through x := a[i]; a[0] := x; j := i-1

Step 1: proof of the invariance of I_{IN}

- Backwards substitution of I_{IN} through a[j+1] := a[j]; j := j -1 yields

$$I^*_{IN} : \forall \overline{p} \left(\begin{array}{l} \left(\left(1 \le \overline{p} < j-1 \right) \rightarrow a\left[\overline{p}\right] \le a\left[\overline{p+1}\right] \right) \wedge \qquad /* \ \overline{p} \neq j+1, j */ \\ \left(j \le \overline{p} < i \right) \rightarrow \left(\begin{array}{l} \textbf{if } \overline{p} = j+1 \ \textbf{ then } a[j] \le a\left[\overline{p}+1\right] \\ \qquad \textbf{else if } \overline{p} = j \textbf{ then } a\left[\overline{p}\right] \le a[j] \\ \qquad\qquad \textbf{else } a\left[\overline{p}\right] \le a\left[\overline{p}+1\right] \end{array} \right) \end{array} \right) \wedge$$

$$x \le a[j] \wedge 0 \le j-1 < i \wedge x = a[0] \wedge$$
$$\left(j < i \rightarrow a[j-1] \le a[j] \right)$$

- Let us now prove that $I_{IN} \wedge x < a[j]$ implies I_{IN}^* step by step:
  1. $x < a[j]$ implies $x \le a[j]$
  2. $x = a[0]$ implies $x = a[0]$
  3. $j < i$ implies $j - 1 < i$
  4. $j = 0$ implies a contradiction between $x < a[j]$ and $x = a[0]$ so that $x < a[j]$ implies $j \neq 0$ and $0 \le j \wedge j \neq 0$ implies $0 \le j - 1$

- So far we have proven $x \le a[j] \wedge 0 \le j-1 < i \wedge x = a[0]$

- It remains to be proven:

$$\forall \overline{p}\left(\begin{array}{l}\left(\left(1 \le \overline{p} < j-1\right) \rightarrow a[\overline{p}] \le a[\overline{p+1}]\right) \wedge \\ \left(j \le \overline{p} < i\right) \rightarrow \left(\begin{array}{l}\textbf{if } \overline{p} = j+1 \textbf{ then } a[j] \le a[\overline{p}+1] \\ \qquad \textbf{else if } \overline{p} = j \textbf{ then } a[\overline{p}] \le a[j] \\ \qquad\qquad \textbf{else } a[\overline{p}] \le a[\overline{p}+1]\end{array}\right)\end{array}\right) \wedge$$

$$(j < i \rightarrow a[j-1] \le a[j])$$

5. $\forall \overline{p}\left(1 \le \overline{p} < j\right) \rightarrow a[\overline{p}] \le a[\overline{p}+1]$

   obviously implies

   $\forall \overline{p}\left(1 \le \overline{p} < j-1\right) \rightarrow a[\overline{p}] \le a[\overline{p}+1] \wedge a[j-1] \le a[j]$

6. $\forall \overline{p}\left(j+1 \le \overline{p} < i\right) \rightarrow a[\overline{p}] \le a[\overline{p}+1]$

   implies that

   $\forall \overline{p}\left(j \le \overline{p} < i\right) \wedge \overline{p} \ne j \wedge \overline{p} \ne j+1 \rightarrow a[\overline{p}] \le a[\overline{p}+1]$

- It remains to consider the cases : $\overline{p} = j$ and $\overline{p} = j+1$

7. If $\bar{p} = j+1$ then $j < i-1$ (because $\bar{p} < i$) and therefore $a[j] \le a[j+1]$,
   but also $a[j+1] \le a[j+2]$ and so $a[j] \le a[j+2] = a[\bar{p}+1]$

8. If $\bar{p} = j$, then $a[\bar{p}] = a[j] \rightarrow a[\bar{p}] \le a[j]$

9. Finally, $j < i \rightarrow a[j-1] \le a[j]$ can be deduced by I_{IN}, when $j > 1$,
   and by $x = a[0] \wedge x < a[j]$ when $j = 1$.

/*
From the first row of IN
$((1 \le \bar{p} < j) \rightarrow a[\bar{p}] \le a[\bar{p}+1])$
with $\bar{p} = j$-1
* /

Step 2 : Proof that $I_{IN} \wedge x \geq a[j] \rightarrow I^*_{EX}$, i.e.,

$$I_{IN} : \forall \overline{p} \left(\begin{array}{l} \left(\left(1 \leq \overline{p} < j \right) \rightarrow a[\overline{p}] \leq a[\overline{p}+1] \right) \wedge \\ \left(\left(j+1 \leq \overline{p} < i \right) \rightarrow a[\overline{p}] \leq a[\overline{p}+1] \right) \end{array} \right) \wedge$$
$$x \leq a[j+1] \wedge 0 \leq j < i \wedge x = a[0] \wedge \left(j < i-1 \rightarrow a[j] \leq a[j+1] \right)$$

implies

$$I^*_{EXT} : 0 \leq i \leq n \wedge \forall \overline{p} \left(1 \leq \overline{p} < i \right) \rightarrow \textbf{if } \overline{p} = j+1 \textbf{ then } x \leq a[\overline{p}+1]$$
$$\textbf{else if } \overline{p} = j \textbf{ then } a[\overline{p}] \leq x$$
$$\textbf{else } a[\overline{p}] \leq a[\overline{p}+1]$$

1. The case $\overline{p} \neq j \wedge \overline{p} \neq j+1$ is trivial
2. The case $\overline{p} = j+1$ comes from $x \leq a[j+1] = a[\overline{p}] \leq a[\overline{p}+1]$
3. The case $\overline{p} = j$ comes from $a[\overline{p}] = a[j]$ and $x \geq a[j]$

Notice that $0 \leq i \leq n$ is obviously invariant in the inner loop.

It has not been explicity mentioned in I_{IN} in order to make the notation not too heavy.

Step 3 : $I_{EX} \wedge i \leq n \rightarrow \overline{I_{IN}}$ where $\overline{I_{IN}}$ is the result of backwards propagating I_{IN} through $x := a[i]; a[0] := x;\ j := i-1$, i.e., the predicate $\forall \bar{p}\left(1 \leq \bar{p} < i-1\right) \rightarrow a\left[\bar{p}\right] \leq a\left[\bar{p}+1\right] \wedge 0 \leq i-1$.

In fact, backwards substitution of $j+1 \leq \bar{p} < i$ through $j := i-1$ gives F so that $\left(i \leq \bar{p} < i\right) \rightarrow a\left[\bar{p}\right] \leq a\left[\bar{p}+1\right]$ is always T.

The same happens for $j < i-1$

We are eventually finished!

A lot of tedious details?
Yes, but most of them are trivial: we need to be systematic - and *helped* (not *replaced*!) by some machine. This does happen -sometimes- in practice: we will see the B method.

The termination issue

•Consider the following program:

•P: **begin**

- x := n; y := m; c := 0;
- **while** x ≠ 0 **do** x := x - y; c := c + 1 **end**

• **end**

•It is immediate to prove {True} P {c = n/m} by means of the invariant

J: n - y.c = x ∧ y = m

•However, it is also clear that P's execution never terminates unless n is a multiple of m.

•This is due to the fact that the while-loop proof rule formalizes the semantics that "if and when the loop execution terminates, then we can rely on the invariant and the negation of the loop condition to hold". Nothing is said about termination however. For this reason Hoare's correctness proofs are called *partial correctness* proofs.

- If program execution termination is not obvious we need to prove termination in addition to partial correctness.
- Here we briefly introduce a termination proof method (with not much emphasis)
- Let us start from the above example. Consider the following assertion:

$$J': \exists \bar{k}\left(x = y \cdot \bar{k} \wedge x \geq 0 \wedge y > 0 \wedge \bar{k} \geq 0\right)$$

- J' defines the set of multiples of y as possible values of x.
- J' is invariant too, w.r.t. P's loop, i.e., if x is a multiple of y at the beginning of loop execution, then, it is so even at the end:
  - Backwards substitution:

$$\left\{\exists \bar{k}\left(x - y = y \cdot \bar{k} \wedge x \geq y \wedge y > 0 \wedge \bar{k} \geq 0\right)\right\} x := x - y; c := c + 1$$
$$\left\{\exists \bar{k}\left(x = y \cdot \bar{k} \wedge x \geq 0 \wedge y > 0 \wedge \bar{k} \geq 0\right)\right\}$$

– Implication proof:

$$
\begin{aligned}
&J' \wedge x \neq 0 \equiv \\
&\exists \bar{k}\left(x = y \cdot \bar{k} \wedge x \geq 0 \wedge y > 0 \wedge \bar{k} \geq 0\right) \wedge x \neq 0 \rightarrow \\
&\exists \bar{k}\left(x = y \cdot \bar{k} \wedge x > 0 \wedge y > 0 \wedge \bar{k} > 0\right) \rightarrow \\
&\exists \bar{k}\left(x - y = y \cdot \left(\bar{k} - 1\right) \wedge x > 0 \wedge y > 0 \wedge \bar{k} > 0\right) \rightarrow \\
&\exists \bar{h}\left(x - y = y \cdot \bar{h} \wedge x > 0 \wedge y > 0 \wedge \bar{h} \geq 0\right) \rightarrow \\
&\exists \bar{h}\left(x - y = y \cdot \bar{h} \wedge x - y \geq 0 \wedge y > 0\right) \wedge \bar{h} \geq 0) \rightarrow \\
&\exists \bar{k}\left(x - y = y \cdot \bar{k} \wedge x \geq y \wedge y > 0 \wedge \bar{k} \geq 0\right)
\end{aligned}
$$

- Thus:

$$\left\{\exists \bar{k}\left(n = m \cdot \bar{k} \wedge \bar{k} \geq 0 \wedge m > 0\right)\right\} P \left\{n = m \cdot c\right\}$$

- Notice that the above proof is still a *partial correctness* proof!

- However, we now know that, if

$$\exists \bar{k}\left(n = m \cdot \bar{k} \wedge \bar{k} \geq 0 \wedge m > 0\right)$$

  - holds before P's execution, then x always ranges over the positive -or null- multiples of m. It also happens that x's value is decreased by the positive value y = m at every iteration. Thus, x must eventually reach the 0 value and therefore the loop must be terminated.

  - The above example can be easily generalized to the following ...

Termination proof method

- Let L = **while** C **do** L_B **end** be any loop within a program P
- Let z denote the collection of P's variables and let Z denotes its state space.
- Let z' and z" denote the value of z before and after execution of the loop body, respectively.
- If we find:
  - 1. A well-founded set $< \mathcal{W}, \succ >$
    (a well founded set is an ordered set such that there do not exist infinite decreasing sequences $w_0 > w_1 > w_2 > \ldots$)
  - 2. A (possibly partial) function f : Z $\rightarrow \mathcal{W}$
  - 3. A loop invariant J such that $J \rightarrow f(z) \in \mathcal{W}$
- And we prove that
  - 4. $J_z^{z'} \wedge C_z^{z'} \rightarrow (f(z') \succ f(z''))$
- Then, we can conclude that L does terminate if J holds before its execution

- In our example we used:
  - The conjunction of the partial correctness invariant J and J' as the loop invariant
  - <N, > > as well founded set
  - f(x,y) = x
  - This allowed to prove conditions 3. and 4. above.
  - Thus we could conclude the termination of P under the assumption (precondition) of:

$$\exists \bar{k}\left(n = \bar{k} \cdot m \wedge \bar{k} \geq 0 \wedge m > 0\right)$$

which guarantees that the invariant holds at the loop entry

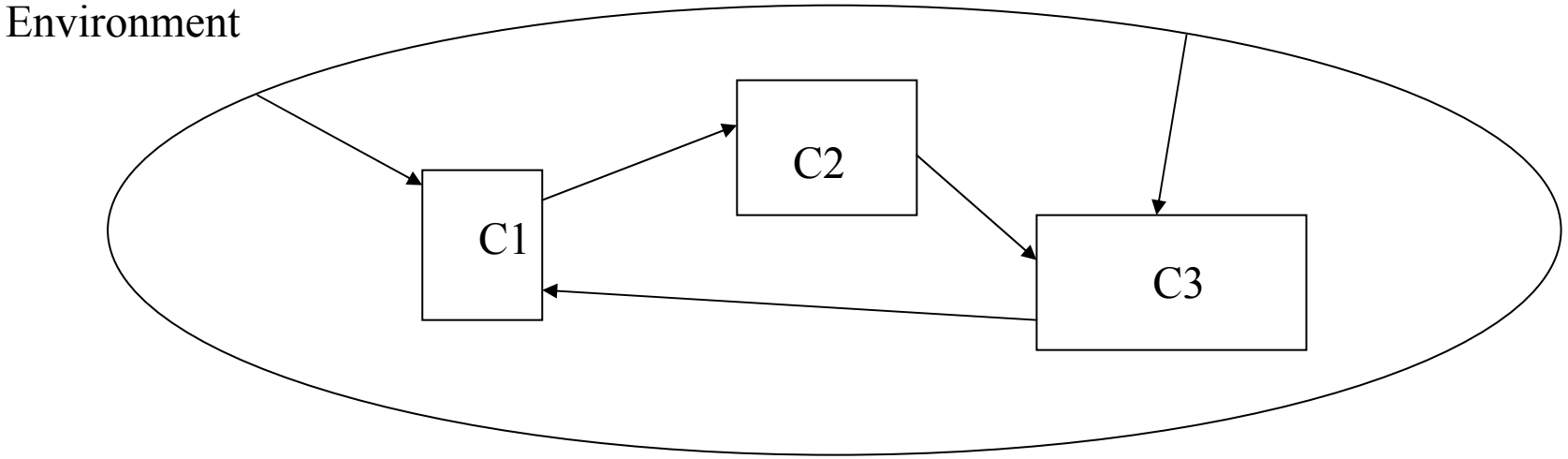
The soundness and completeness issues

- Of course every proof method must be *sound* (theorems must be true!) and possibly/hopefully *complete* (truths should be provable as theorems).
- It is not difficult to show (meta-prove) that Hoare's method is sound.
- Since the method includes arithmetics, and arithmetics is intrinsically incomplete (Goedel's theorem), Hoare's method cannot be complete.
- However, the notion of *relative completeness* has been introduced:
- Intuitively (and without deepening this issue):
  - Suppose an "oracle" can supply truth or falsity of arithmetic formulas
  - Then a proof system is *relatively complete* w.r.t. arithmetics if every truth of the system can be proved by using arithmetics true formulas as hypotheses.
- Hoare's method is not even relatively complete (w.r.t. arithmetics) but other more sophisticated methods are.

Formal models and methods for concurrent, distributed, and real-time systems

- Concurrent systems: several computations/processes are ongoing in parallel and need coordination
- Real-time systems: time, which is usually abstracted away in the description and analysis of -computing-systems becomes an essential component of their behavior. Typically this is due to the fact that a system (component) must react with given time constraints to stimuli coming from other components (environment). Thus, in most cases, real-time systems are also concurrent (also called *reactive*) systems.
- Let us first consider “purely concurrent” systems and let us postpone the further issue of real-time

- What does it change when moving from a sequential to a concurrent/parallel system?
- Externally:
  - Is it still adequate to formalize a “problem” as a function to be computed, or a string to be translated, or accepted by some device?
    - In some cases yes: e.g., we wish to exploit parallelism to compute -faster- the inverse of a matrix, …
    - In general however, the picture is rather different:

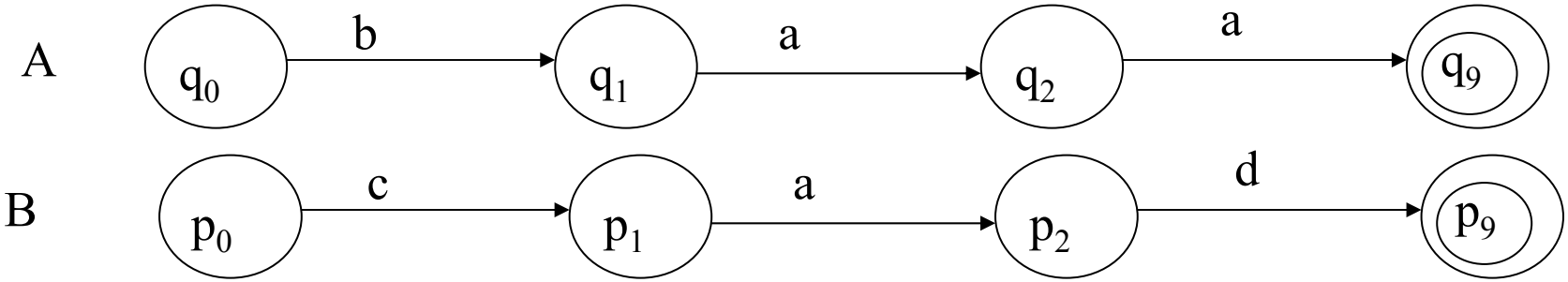


Is the analogy of the (Turing machine) tape(s) still valid?

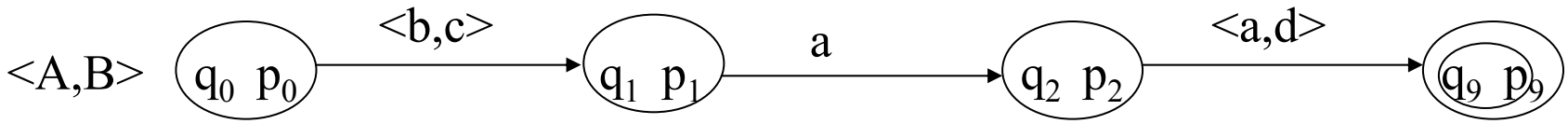
- Furthermore, for such systems, talking about <input data-computation-output data>, or begin-end of computation, is often inadequate: most computations are “never ending”
- We could -and can- resort to the notion of infinite strings and related elaboration (a full theory of infinite strings formal languages has been developed);
- But still we should deal with the *interleaving* of signals/data flowing through different “channels”: how can we describe and manage different sequencings in such signals? Should we consider all possible interleavings? Is this feasible/useful?
  Think e.g. to the following cases:
  - The system is a collection of PCs with independent users performing independent jobs
  - The system is a collection of processors managing signals coming from a plant (sensors) and human operators
- We can easily imagine that a further order of complexity will arise when *temporal relations* among such signals will become a critical issue (e.g. reacting to alarms)

Let us move a few preliminary and basic steps

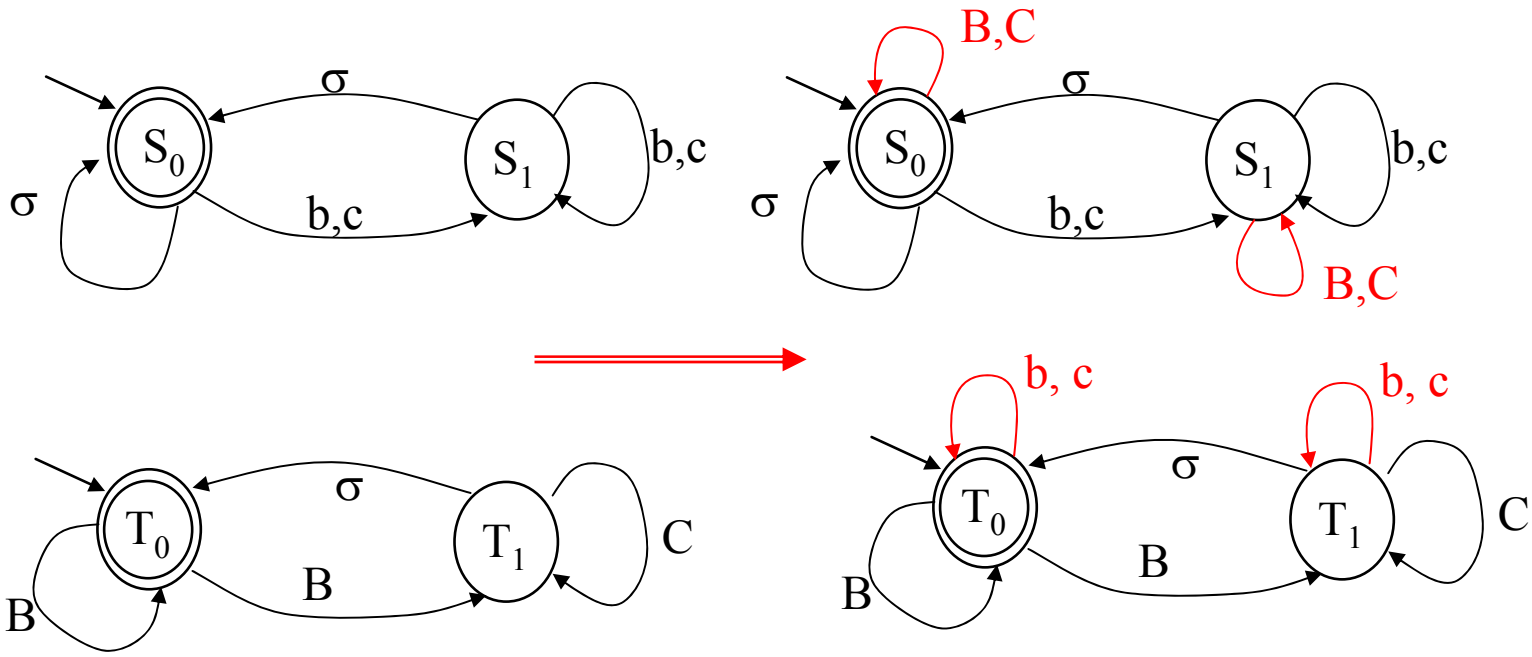
- Let us first look at the issue from an operational point of view:
- A *system* is simply a collection of *abstract machines* -often called *processes*.
- In some cases we can easily and naturally build a *global state* as the composition of *local states* of single machines:



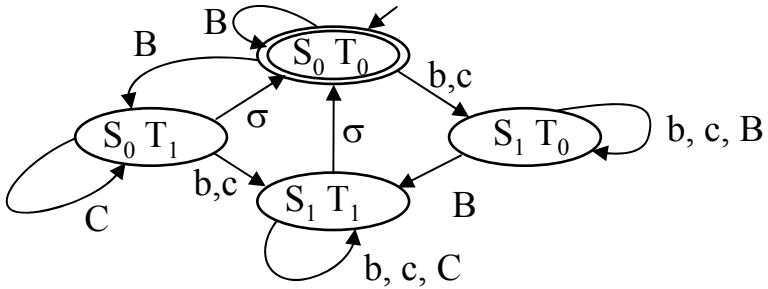
Let us recall the construction of "intersection machines"



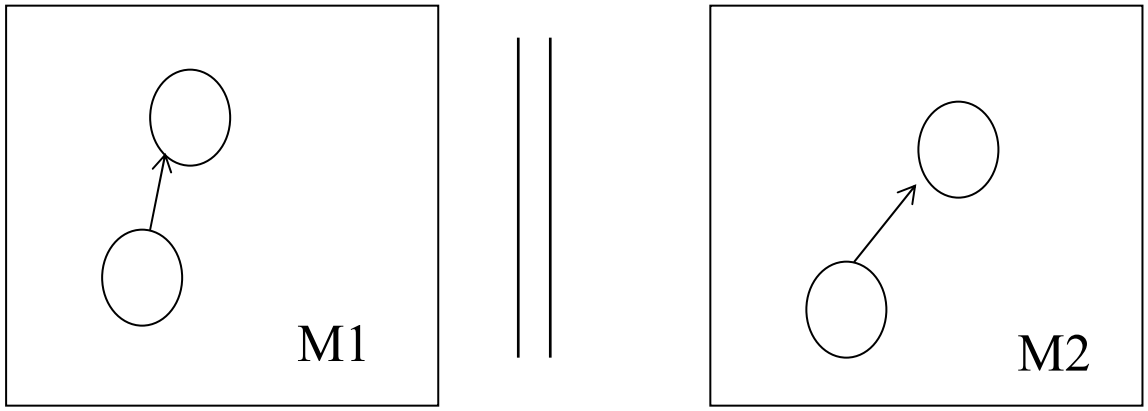
Parallel composition vs. intersection



A1 || A2 = A1' ∩ A2'



- Typical case: composition of FSMs, as in the case of Statechart



We call such an approach *synchronous*:

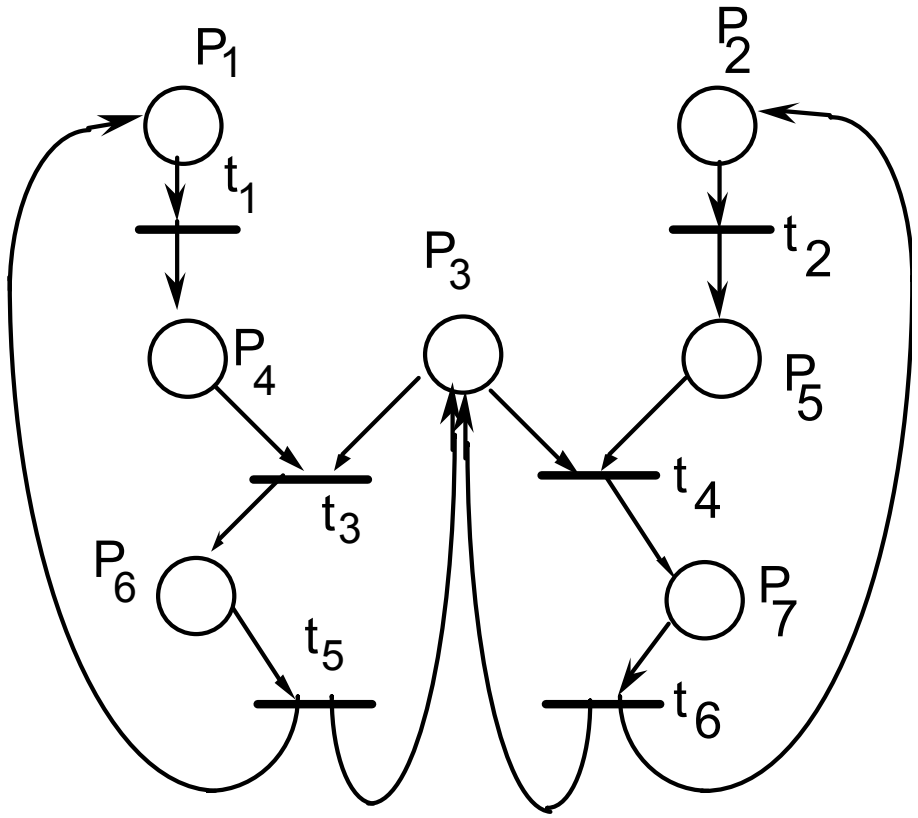
•there is a global clock, with a global state

•at each time tick each component changes its own local state synchronously with other components

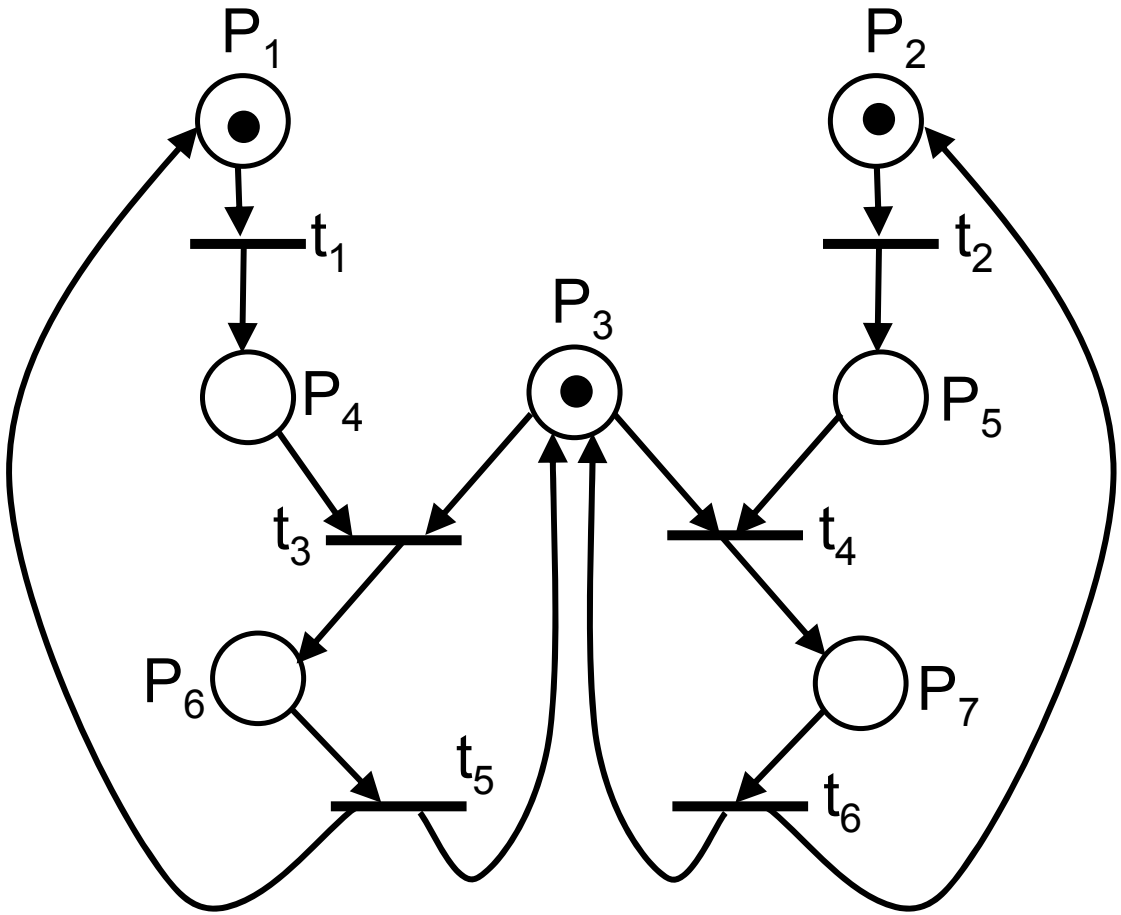
•signals can be local or shared (e.g. the output of a machine is the input to another one)

•time -whether explicit or implicit- is typically discrete

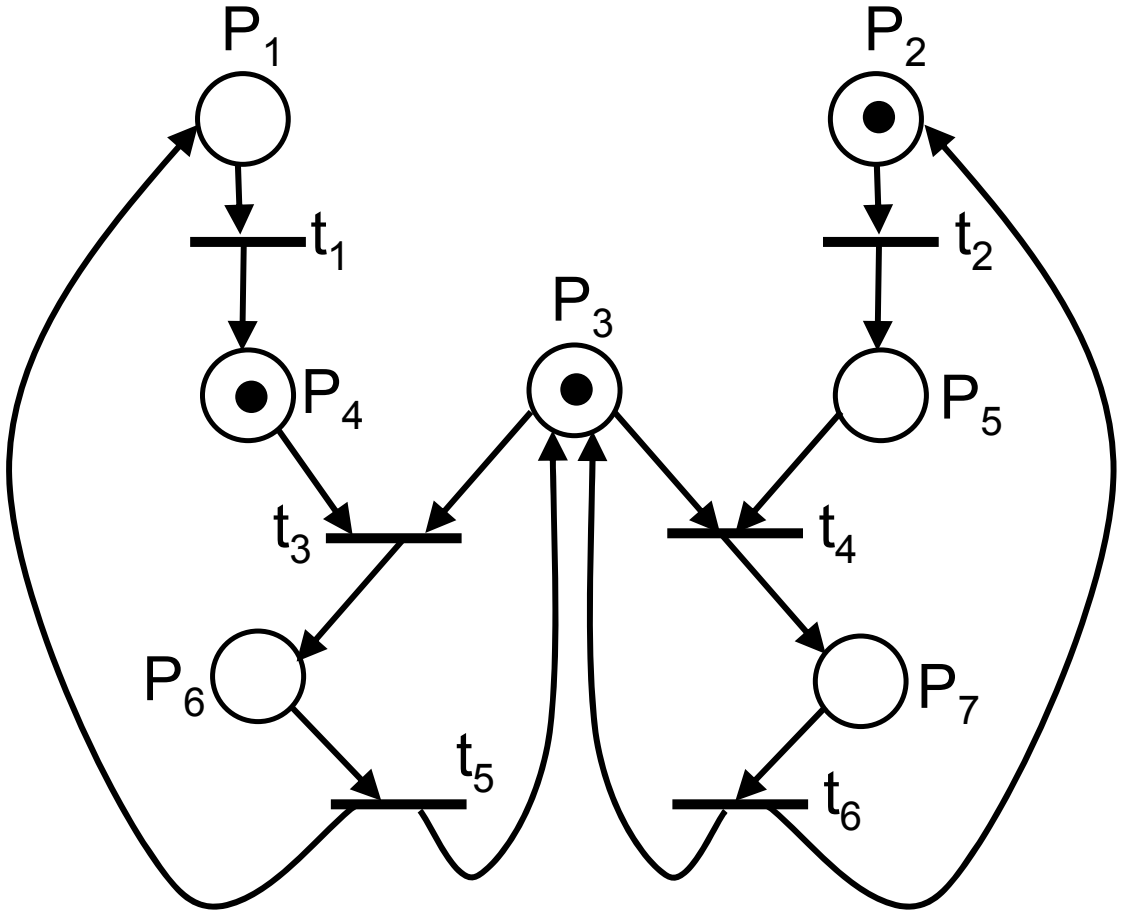
- In other cases it is inconvenient -or even impossible- to “view” a global state which evolves synchronously throughout every component
- Each process evolves autonomously and only occasionally synchronizes with other processes
- Typical *asynchronous* abstract machines are *Petri nets*.
- (already known? If not, basic theory available in the text book, Ch. 1 and 6)



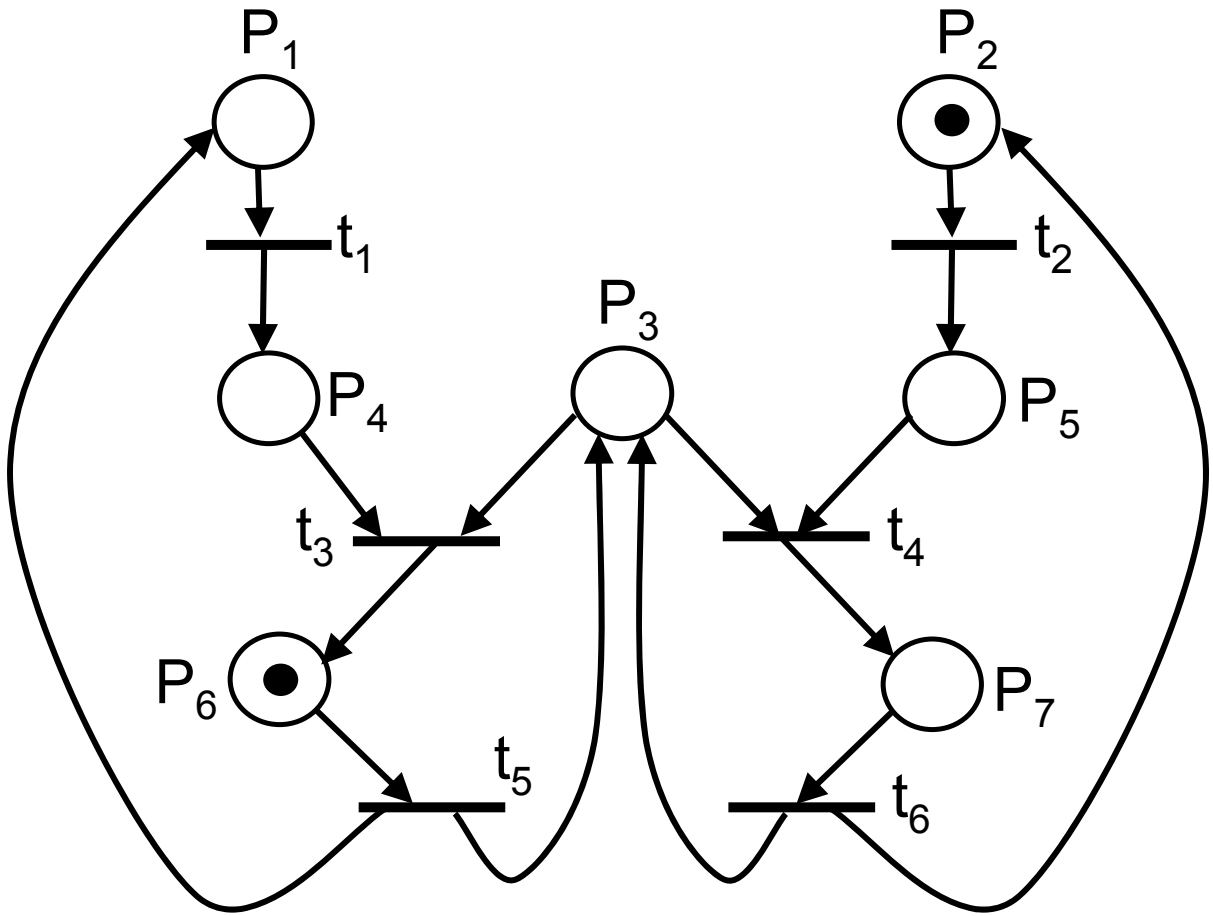
PNs marking and firing sequences:



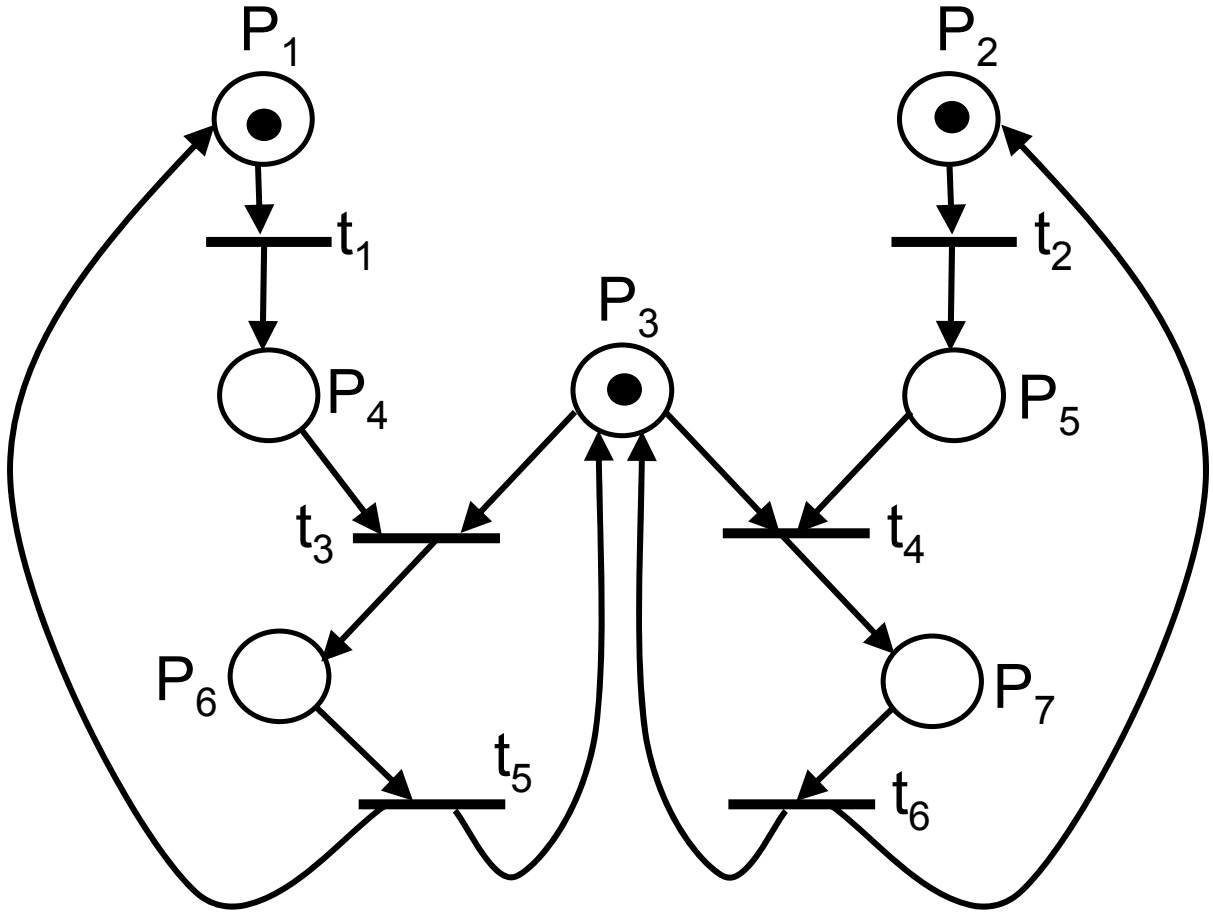
t_1 fires …



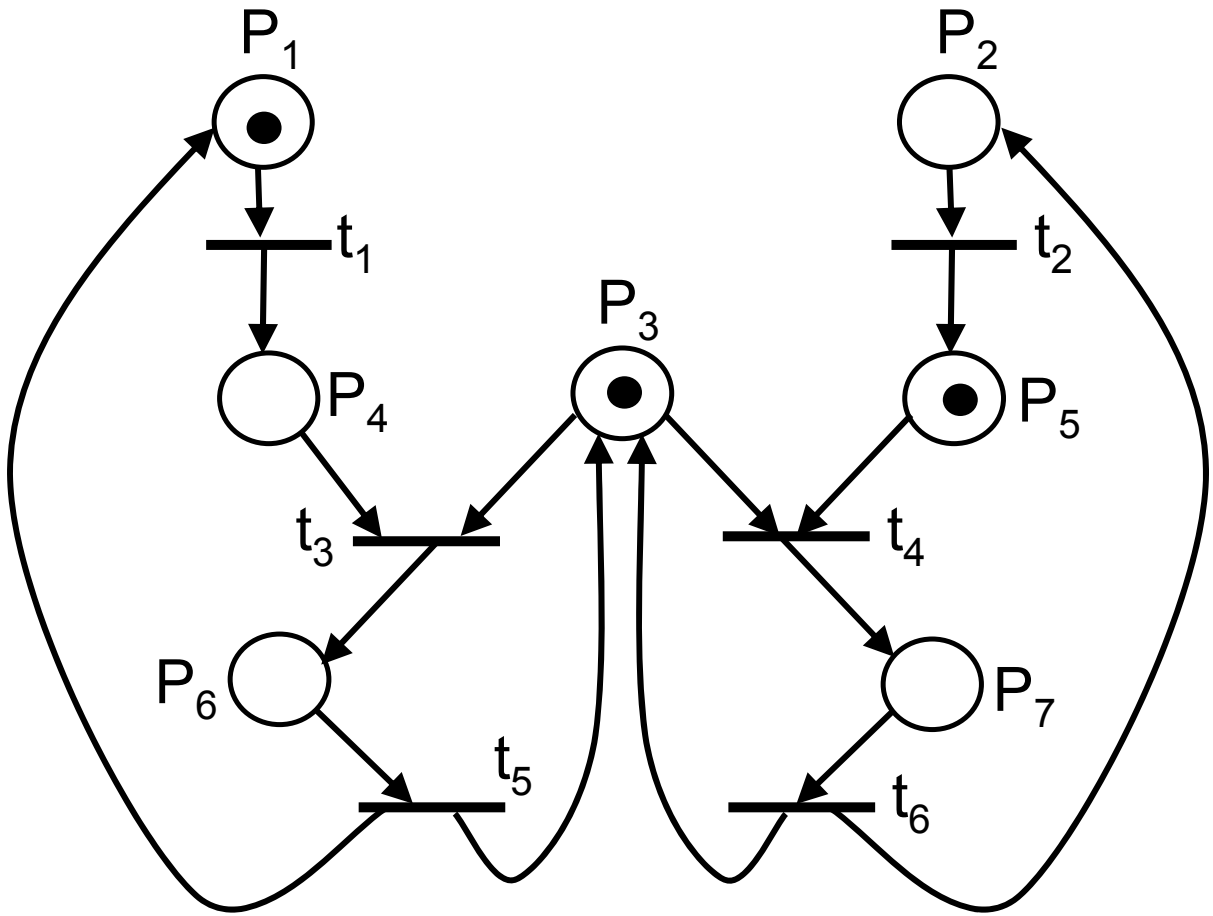
… t_3 fires …



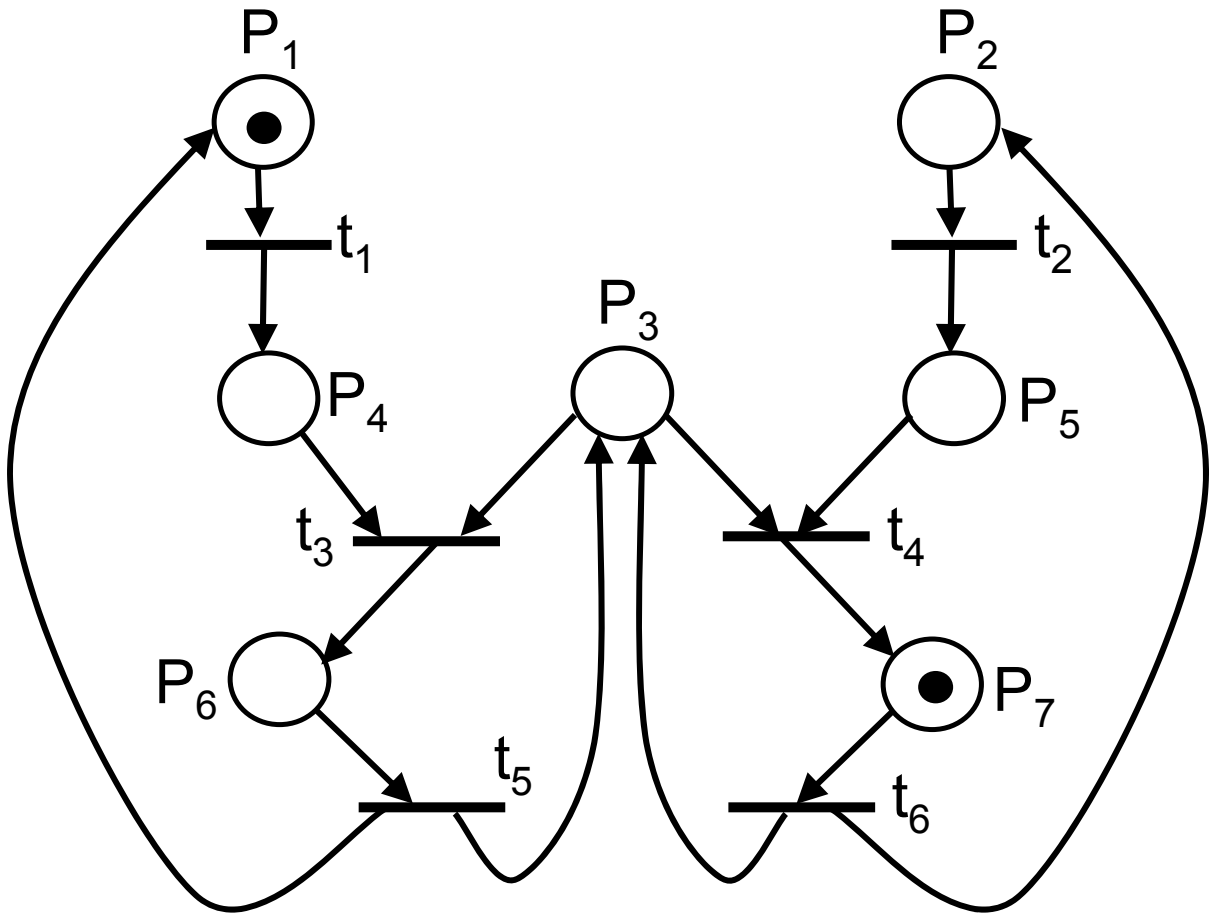
… t_5 fires …



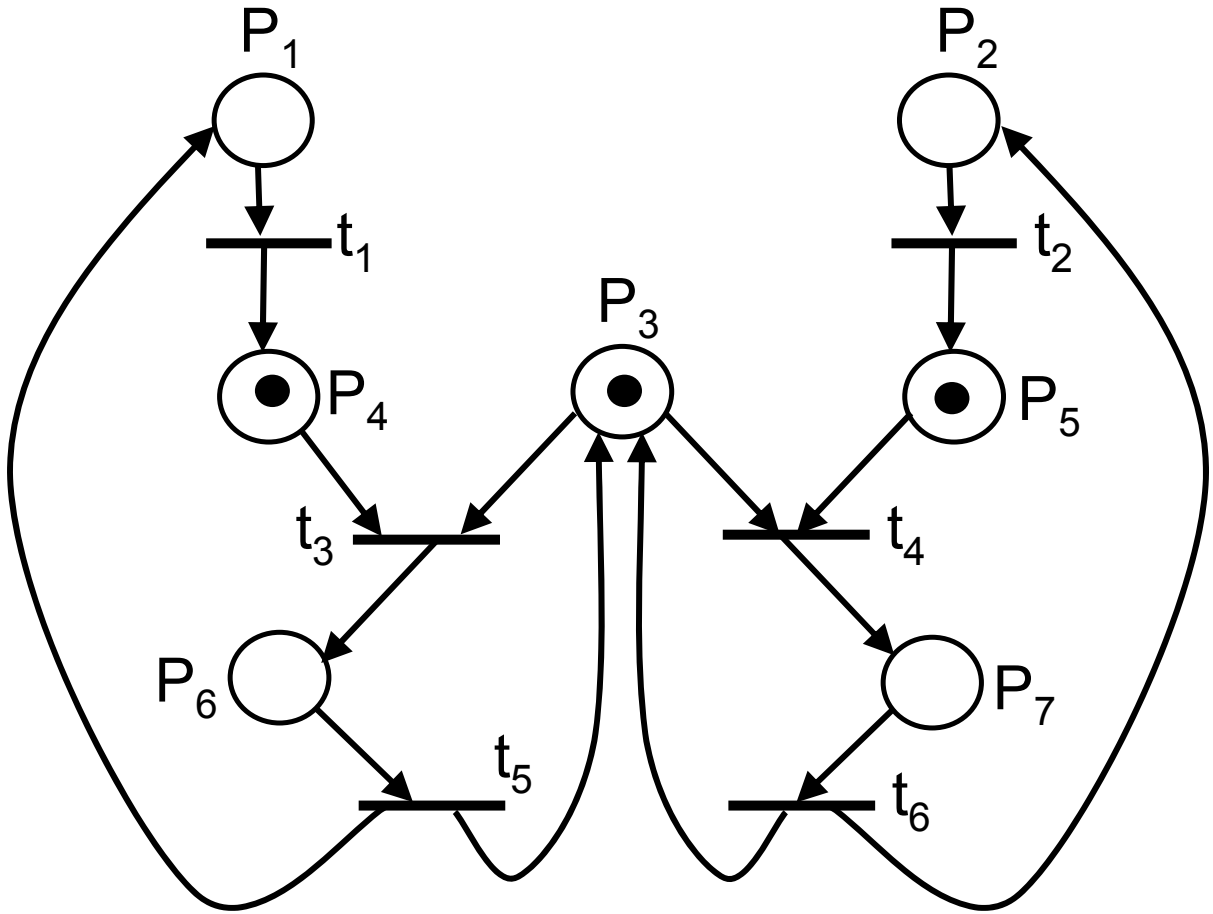
...


P1
P2
t1
t2
P3
P4
P5
t3
t4
P6
P7
t5
t6

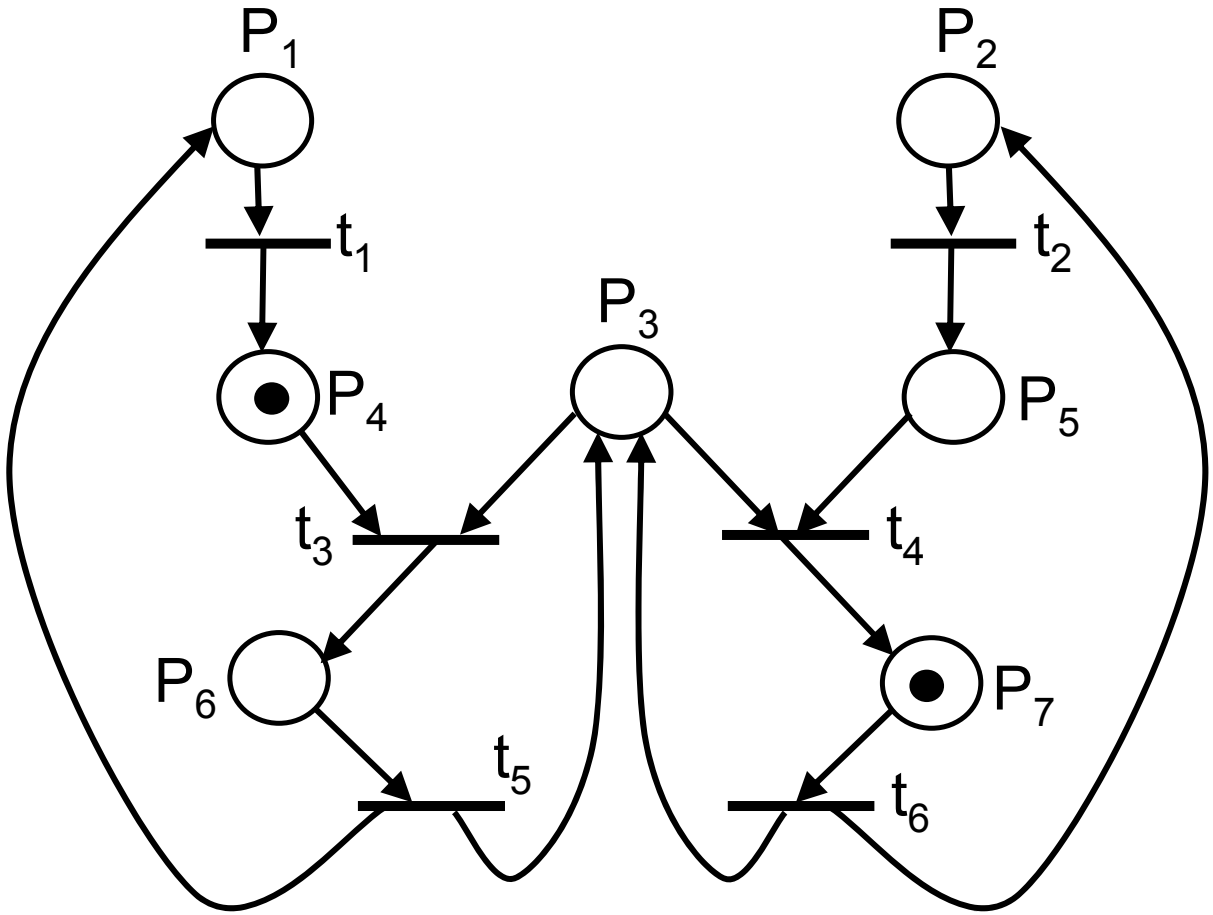
....


P1
P2
t1
t2
P3
P4
P5
t3
t4
P6
P7
t5
t6

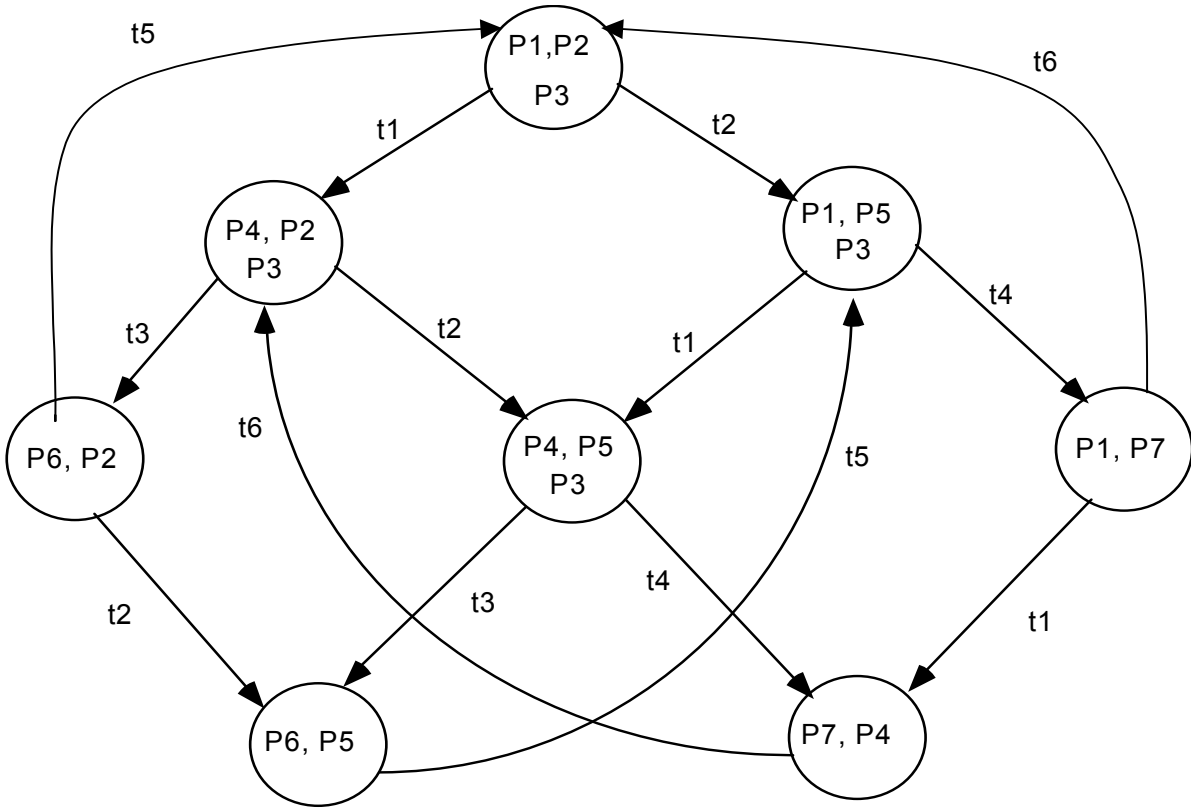
t_1 and t_2 fire simultaneously …



....

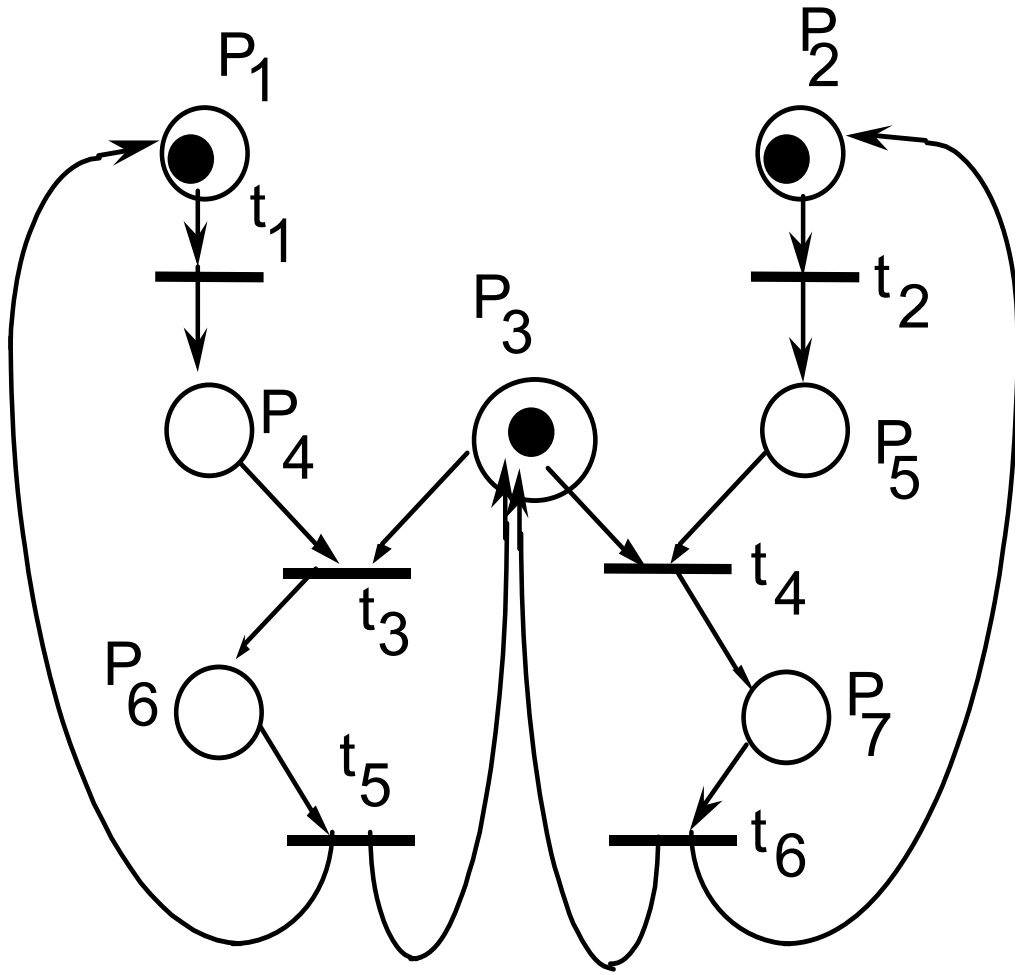

P1
P2
t1
t2
P3
P4
P5
t3
t4
P6
P7
t5
t6

•In some cases we can formally reduce one machine to another one:



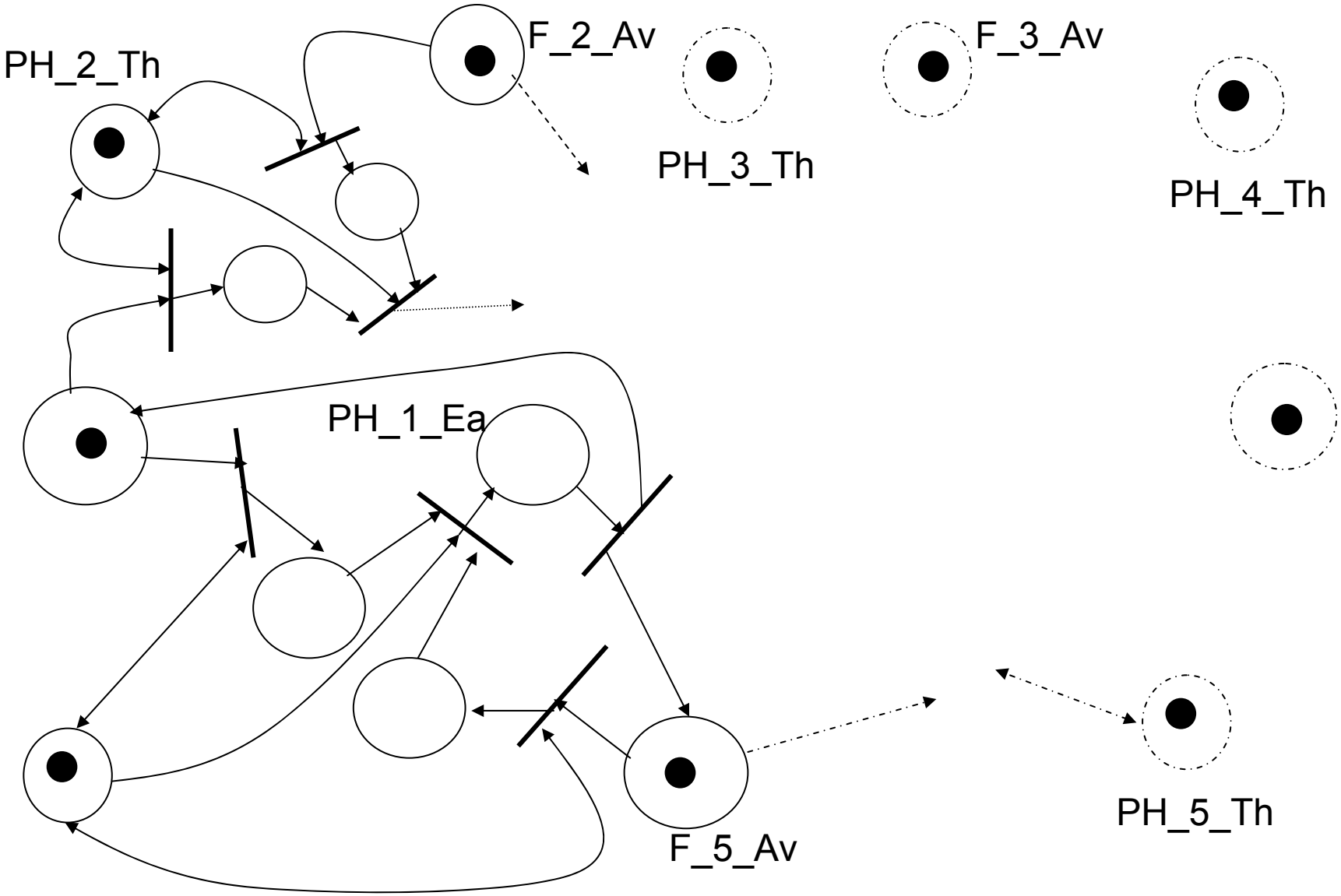
- A FSM description of:

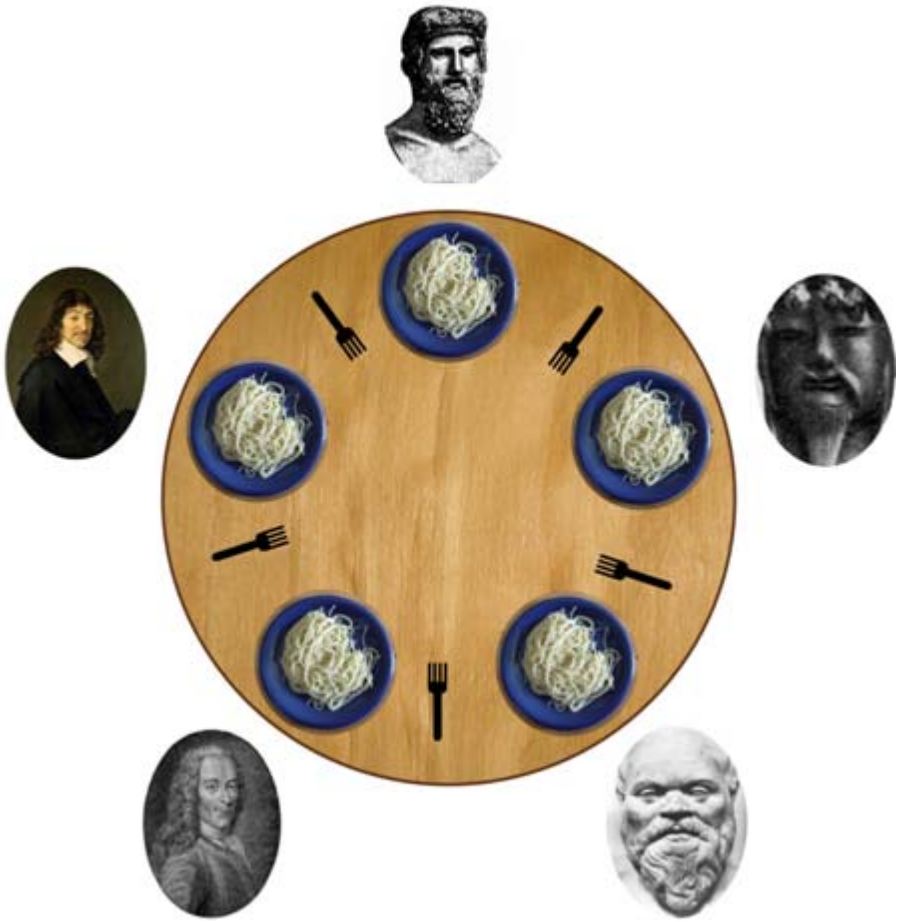
… of the more understandable Petri net



- The FSM formalizes the *interleaving semantics* of the PN
- However synchronous and asynchronous abstract machines have some "philosophical" differences (the "religious war" of asynchronous people against synchronous ones …)
- In some cases however, it is simply impossible to talk about a global system state:
  - in distributed systems the various components are physically located in different locations
  - they communicate through signals flowing in channels
  - when components evolution proceeds at speeds comparable with the light speed it is meaningless saying that "at a given time t component C1 is in state Qa and component C2 in state Qb so that the global system is in state <Qa, Qb>

A PN formalization of the …Dining Philosophers





When time comes into play ...

- … things become even more critical
- Side remark:
  unlike traditional engineering (see dynamical system theory) computer science tends to abstract away from time and to deal with it in a fairly separate fashion (complexity/performance evaluation)
  this may work fine in several cases but certainly not for real-time systems, whose correctness, by definition, does depend on time behavior.
- We must deal with:
  - time occurrence -and order- of events;
  - duration of actions and states
  - in critical cases such time values may depend on data values and conversely (e.g. if the reaction to an alarm depends on some computation, whose duration, in turn, depends on some input data)

- For a fairly long time -in the 80s- there has been a -somewhat annoying- debate on the necessity of dealing explicitly and “metrically” with time in high level specifications (requirements): pure theorists did not like making their formalisms “dirty” with the addition of time (e.g. Petri nets).
- The claim was that what really matters in requirements is an *ordering* of events, not a time measure: e.g.:
  “The system must be shut down before temperature reaches a critical value”
- The point is that the notion of specification is a relative concept:
- If the plants heats at a given speed, then I must *compute* the *time amount* I have within which the system must be shut down.
- Nowadays there is much more agreement that, after all, we do need models that allow managing explicitly time *metrics*.

Time has been "added" to formalisms in several ways

- In operational formalisms:
  - durations have been added to transitions (whether FSM transitions or PN transitions, or …)
  - there is a large literature of timed PNs (time added to transitions, to places, to arcs, ….)
  - often such "added times" are constant values; in more sophisticated cases they may depend on data
  - Some approaches consider time as "yet another system variable, t". Time elapsing is formalized by assignments to such a tick variable. We will -partially- illustrate and criticize such an approach.
- In descriptive formalisms:
  - Pure temporal logic leaves time implicit and with no metrics: time is infinitely "elastic": typical operators such as **eventually**, **until** predicate the occurrence and order of events/states, but do not measure how long these things take
  - Several "metric temporal logics" have been defined (bounded **until**, bounded **eventually**, ….)

A survey of some major timed formalisms

- Timed (finite) state machines
  - Timed Statechart
  - TTM
  - Lynch and Heitmeyer infinite state machines
  - …
- Timed CSP/LOTOS, …timed algebraic approaches
- Timed Petri nets
- Timed (temporal) logics
  - RTL (Koymans)
  - RRTTL (Ostroff)
  - Abadi & Lamport
  - TRIO
  - ASTRAL

- The “dual language” approach (Ostroff, TRIO/PN …)
  - The system is described through an operational formalism (automata, PN, …)
  - Its properties are asserted through a descriptive formalism (e.g. logic)
  - It is “proved” - through some axiomatization- that the -designed- system does indeed exhibit the desired properties

- Reference reading:
  - Heitmeyer and Mandrioli (eds) Formal methods for real-time computing
  - In particular, see the -survey- chapter 1
  - Mandrioli: Transparencies of a Pisa Seminar, November 2003
  - (A more thorough and comprehensive reading):

- We will examine in some detail a few cases among the above formal models
- We will consider moving from formal models to formal methods
- We will experiment with prototype tools and case studies (sometimes imported from real-life projects)
- The usefulness of a suitable benchmark for case studies:
  - a testbed for checking applicability of methods
  - a reference for comparisons
  - a source for inspiration
  - … let's avoid the annoying debate between "toy problems" and "real-life projects" … the two's are often complementary and in some cases mutually derived

Sample benchmarks
(not fully exploited but a good source of -possibly personal- exercises)

- The dining philosophers
- The elevator system
- The (generalized) railroad crossing system
- The reservoir system (see next slides)
- ….

- The reservoir system. Version 1

  - The purpose of the example is to clarify how the description of a system state can be refined through several levels by applying the principle that the refinement adds to, but does not change, the information contained in the upper level.

  - **Informal description.**
  - At the upper level we describe the system as composed of two processes:
  - reservoir and control.
  - Reservoir exports a variable *level* whose evolution depends on the amount of water flowing in and on the fact that the sluice gate is open or closed.
  - A *sluice gate* is managed by the control process which is in charge to open it when the level exceeds a given value and to close it when the level is less than another value.

  - **Refined view**:
  - the sluice gate is operated by engines which take some time to change its state so that it is necessary to express the fact that it may be opening or closing.
  - We maintain an external view where only the fact that the gate is open or closed are visible, but we add an internal more detailed description which takes into account the state of the sluice gate from the point of view of the engines operating on it.

- The reservoir system. Version 2

- The purpose of this example is to help understanding and defining possible implementation mechanisms that allow increasing the number of parallel processes between upper level specifications and lowe level ones without invalidating properties already proved to hold for the UL. The proposed example is an enrichment of previous version 1.

- **Informal description.**

- **Top level.**
- The system consists of two processes, reservoir and control, as in version.1.
- The differences are the following:

- **1.**
- The reservoir exports two variables l1 and l2. These are the level measure obtained by two different sensors which are located in different places of the reservoir. They are intended to measure in some way the "global level" of the reservoir, but also to account and to manage possible little differences due to waves or major anomalies (big waves due to earthquake or sensors malfunctions).

- **2.**
- The control process elaborates the values of l1 and l2 with the purpose of
- a) opening and closing the sluice gate (S_G) when they exceed given minimum and maximum values as it happened in version 1;
- b) raising an anomaly detection signal (variable alarm) when the two measures appear to be inconsistent.
- Jobs a) and b) can be roughly described as follows:

  - a) If the average of the two averages of variables l1 and l2 computed in the last Delta time units exceeds LMAX then the gate must be open.
  - Similarly for LMIN.

  - b) If during the last Delta time units the absolute value of the difference between the averages of l1 and l1 (still computed on the last Delta time units) exceeds a value k for more than h times then the alarm should be raised.

- **Refinement**

- The reservoir process is the same as in the UL

- The control process is refined into 4 parallel processes: P1, P2, PC, and PM
  - P1 and P2 are identical and are in charge of computing updated values l1^ and l2^ of the averages (over the last Delta time units) of l1 and l2 , resp.
  - PC is on charge of computing how many times within the last Delta time units l1^ and l2^ differed by more than k.
  - PM gathers the above results from P1, P2, PC and decides whether to open or close the gate (in the same way as in example 1) and to raise the alarm.

- Comments:
- This type of parallel refinement seems nicely realistic. Think of coprocessors that make heavy sequential computations such as FFTs instead of simple averages and comparisons as in this toy example.
- It also shows that we can have a good amount of parallel and asynchronous computation still keeping the principle that events that represent UL events (start and end of transitions) are "synchronized" with their UL counteparts.

The case of **the TRIO language**

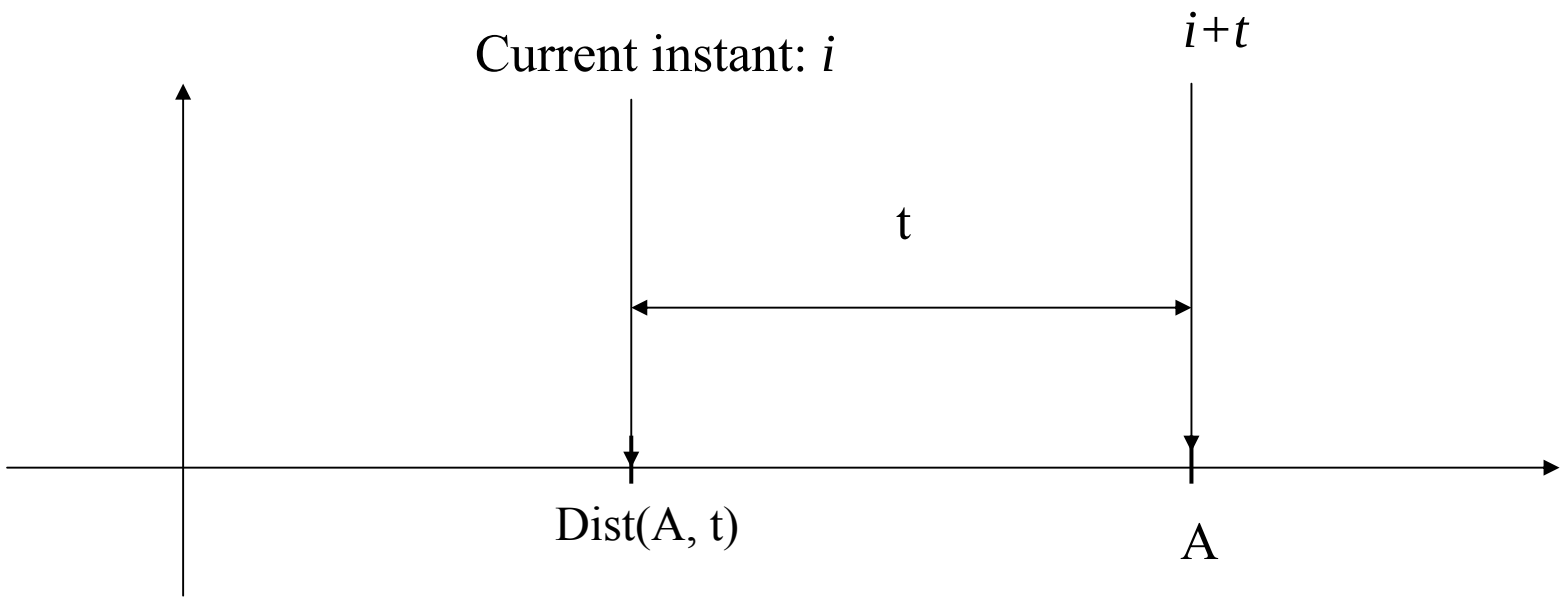
- One of the -first- “metric extensions” of temporal logics
- One of the -first- OO+ graphically extended formalisms to cope with the many typical problems of using formalisms in large, real-life cases (now we have VDM++, Z++, Objective Z, OO-PN, …)
- First we look at the “kernel” of TRIO
- Later we will develop the full language/methods/tools/applications

The logic kernel of TRIO

- Imports from classical temporal logic:
  - Extending first order logic
  - Implicit current time
  - asserting future (and past evolution)
- Introduces
  - Quantitative measuring time distances
    Thus, there exists an explicit time domain which is not fixed but must be numeric in nature: it requires basic arithmetic operations and (in general, total) ordering.
    Typical time domains are:
    - discrete domains such as Naturals and Integers
    - dense domains such as rationals and reals
    - This is a distinguishing feature of TRIO: many other temporal formalisms fix time domain to be either discrete or continuous
  - Only one basic temporal operator:

Dist(A, t)

- If A is any TRIO formula and t is a term in the time domain
- Dist(A, t) is a TRIO formula and -intuitively- means that it holds if and only if A holds at distance t from the current instant (i.e., the instant when Dist(A, t) is claimed) in the temporal domain.



- In TRIO, as well as in classical temporal logic, all formulas refer to an implicit current instant, the “now” when the formula is claimed
- Apart from the above distinguishing feature, the rest of TRIO syntax and semantics are typically first-order (and thus, does not need further explanations)
- First example:
- Light_on $\wedge$ Dist($\neg$ Light_on, -3) $\wedge$ Dist($\neg$ Light_on, 3) means that now the light is on, but it was not on -*exactly*- 3 time units ago and will not be on 3 time units in the future.
- Notice that the operator Dist predicates the holding of a formula *exactly* at the -relative- time indicated: it does not claim anything about the time interval in between now and the indicated instant: thus, the above formula says nothing about, say 1 time unit ago.

- What if we wanted to claim that the light was on *only* since -1 to now and off (better, not on) in the remaining of the interval [-3 ..3], including the boundaries?
- $\forall\ t\ \ (-3 \le t < -1) \rightarrow (\text{Dist}\ (\neg\ \text{Light\_on}, t)$
  $\wedge$
  $(-1 \le t \le 0) \rightarrow (\text{Dist}\ (\text{Light\_on}, t)$
  $\wedge$
  $(0 < t \le 3) \rightarrow (\text{Dist}\ (\neg\ \text{Light\_on}, t)$
- The only operator Dist is general enough to deal with all possible temporal properties we wish to predicate
- In fact any other temporal operator can be defined as a derived operator in terms of Dist. For instance:
- Property P holds from now on for (at least) K time units:
- $\text{Lasts}(P, K) \quad \Leftrightarrow \quad \forall t\ (0 \le t \le K \rightarrow \text{Dist}(P, t))$
  The symbol $\Leftrightarrow$ will denote “is defined as”

A (partial) list of useful derived temporal operators
The list is open: others can be defined as useful

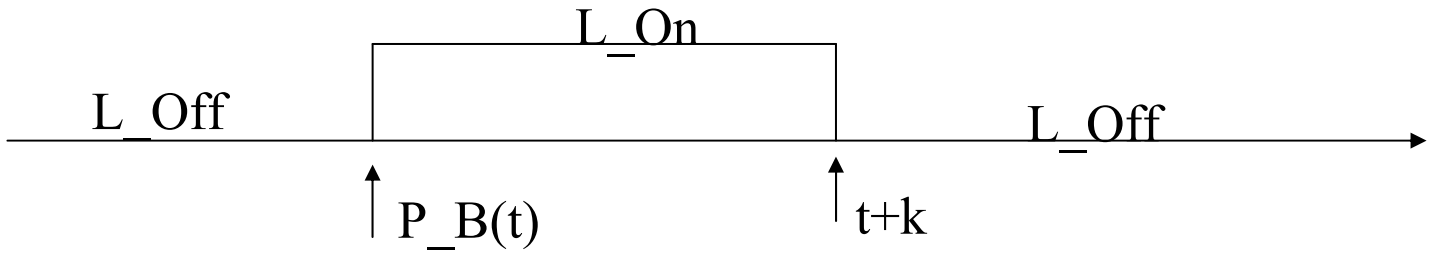
• Futr(F, d)	$\Leftrightarrow$	$d \geq 0 \wedge$ Dist(F, d)	Future
• Past(F, d)	$\Leftrightarrow$	$d \geq 0 \wedge$ Dist(F, -d)	Past
• $Lasts_{ee}(F, d)$	$\Leftrightarrow$	$\forall d'$ $(0<d'<d \rightarrow$ Dist(F, d'))	F holds over a period of length d (boundaries of the interval are excluded)
• $Lasted_{ie}(F, d)$	$\Leftrightarrow$	$\forall d'(0 \leq d'<d \rightarrow$ Dist(F, -d'))	F held over a period of length d in the past (now is included, now -d is excluded)
• Alw(F)	$\Leftrightarrow$	$\forall d$ (Dist(F, d))	F always holds
• SomF (A)	$\Leftrightarrow$	$\exists d(d \geq 0 \wedge$ Dist(F, d))	Sometimes in the future F will hold

- $\text{Within}_{ee}(F, d) \Leftrightarrow \exists d'(0<d'<d \wedge \text{Dist}(F, d'))$ F will occur within d time units

- $\text{UpToNow}(F) \Leftrightarrow \exists d (d > 0 \wedge \text{Past}(F, d) \wedge \text{Lasted}_{ee}(F, d))$
  F held for a nonnull time interval that ended at the current instant
- $\text{Becomes}(F) \Leftrightarrow F \wedge \text{UpToNow}(\neg F)$
  F holds at the current instant but it did not hold for a nonnull interval that preceded the current instant
- $\text{NextTime}(F, t) \Leftrightarrow \text{Futr}(F, t) \wedge \text{Lasts}_{ie}(\neg F, t)$
  The first time in the future when F will hold is t time units apart from the current instant
- $\text{Until}(A_1, A_2) \Leftrightarrow \exists t (t > 0 \wedge \text{Futr}(A_2, t) \wedge \text{Lasts}(A_1, t))$
  A_1 holds until A_2 becomes true
- $\text{Since}(A_1, A_2) \Leftrightarrow \exists t (t > 0 \wedge \text{Past}(A_2, t) \wedge \text{Lasted}(A_1, t))$
  A_1 held since A_2 became true

- Why the choice of the implicit current time instead of predicating explicitly any time?
  - Most systems are “time invariant”: their behavior is in general identical for any initial time and give initial “state”
    ---->
  - most requirements are of the type: “if event E occurs, related event E1 must occur within Δ”, or “state S must last at least -or no more than- Δ since the time it is entered”, …
  - to better understand, let us compare previous formulas stated with explicit time with the corresponding TRIO formulas

Going back to our old timer:



Non TRIO

$$\forall t(L\_On(t) \leftrightarrow \neg L\_Off(t)) \quad \text{(we can argue about that)} \quad \wedge$$

$$Event\_P\_B \quad \wedge$$

$$\forall t((P\_B(t) \wedge Up\_to\_now\_L\_Off(t)) \rightarrow$$

$$\forall t_1((t \le t_1 < t+k) \rightarrow L\_On(t_1)) \wedge L\_Off(t+k)) \quad \wedge$$

$$\forall t_3, t_4(L\_Off(t_3) \wedge \forall t_5((t_3 \le t_5 \le t_4) \rightarrow \neg P\_B(t_5))) \rightarrow L\_Off(t_4)$$

TRIO

$$Alw(L\_On \leftrightarrow \neg L\_Off) \quad \wedge$$

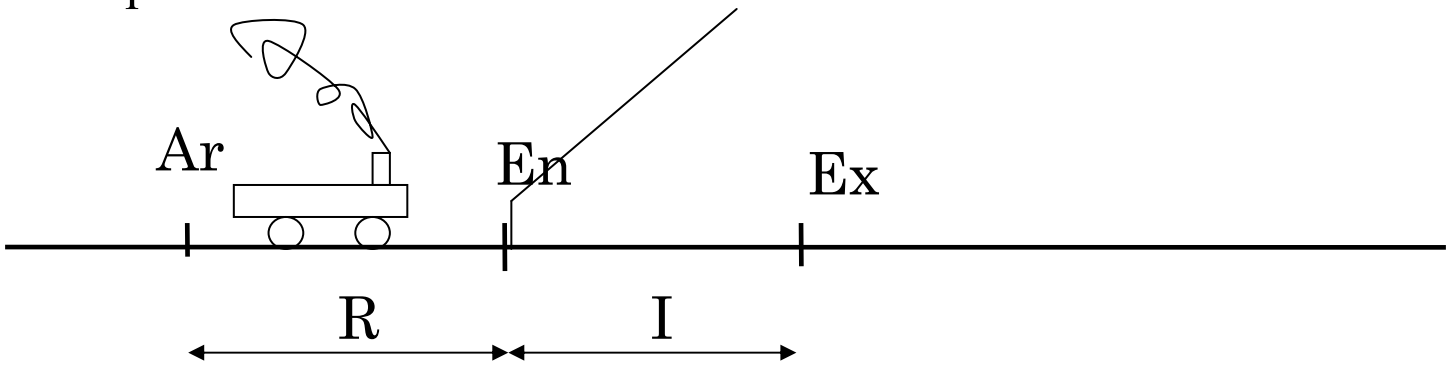
$$Event\_P\_B \quad \text{(we will introduce the notion of event as a built - in TRIO concept)} \quad \wedge$$

$$((P\_B \wedge Up\_to\_now(L\_Off) \rightarrow Lasts_{ie}(L\_On, k) \wedge Futr(L\_Off, k)) \quad \wedge$$

$$(L\_Off \wedge Lasts_{ie}(\neg P\_B, k)) \rightarrow Lasts_{ie}(L\_Off, k)$$

Let us now go back to another old example: the railroad crossing system (with a first, minor, generalization) (KRC: Kernel railroad crossing)

- Informal description:



  - Only one train
  - d_m and d_M, minimum and max time, resp., to go from the beginning of R to the beginning of I
  - h_m and h_M, minimum and max time, resp, to go through I
  - The gate can be open or closed but also moving up and down
  - The moving of the gate takes γ and cannot be interrupted

Let us replay the original formalization with the minor necessary modifications and with the use of TRIO instead of pure first-order calculus

- *EnterR*, *EnterI*, and *ExitI* are time dependent predicates, i.e., predicates that may hold or not in different instants (with zero arguments). They represent, respectively the events of a train entering region R, entering region I, or exiting region I (and hence R). Predicates *InR* and *InI* represent the state of a train being in region R (i.e., it has entered the region but it did not exit yet) or, respectively, in region I. Since we consider only one single train, each of the events EnterR, EnterI, and ExitI occur at most once:
- (k1) $\text{EnterR} \rightarrow \text{AlwP}(\neg\text{EnterR}) \wedge \text{AlwF}(\neg\text{EnterR})$
- (k2) $\text{EnterI} \rightarrow \text{AlwP}(\neg\text{EnterI}) \wedge \text{AlwF}(\neg\text{EnterI})$
- (k3) $\text{ExitI} \rightarrow \text{AlwP}(\neg\text{ExitI}) \wedge \text{AlwF}(\neg\text{ExitI})$

- Axioms describing the train movement are TRIO restatements of previous formalization:
- (k4) $\text{EnterR} \rightarrow \exists t(d_m \leq t \leq d_M \wedge \text{Futr}(\text{EnterI}, t))$
- (i.e., its takes the train a time between d_m and d_M to go from region R to region I).
- (k5) $\text{EnterI} \rightarrow \exists t(h_m \leq t \leq h_M \wedge \text{Futr}(\text{ExitI}, t))$
- (i.e., its takes the train a time between h_m and h_M to go exit region I once entered it).
- Also, an exit event occurs only after the corresponding enter event.
- (k6) $\text{EnterI} \rightarrow \exists t(d_m \leq t \leq d_M \wedge \text{Past}(\text{EnterR}, t))$
- (k7) $\text{ExitI} \rightarrow \exists t(h_m \leq t \leq h_M \wedge \text{Past}(\text{EnterI}, t))$
- The above time bounds on train movements satisfy the following obvious axiom.
- (k8) $d_M \geq d_m > 0 \wedge h_M \geq h_m > 0$
- A train is inside region R or I if it did not exit that region since it entered it.
- (k9) $\text{InR} \leftrightarrow \text{Since}(\neg\text{EnterI}, \text{EnterR})$
- (k10) $\text{InI} \leftrightarrow \text{Since}(\neg\text{ExitI}, \text{EnterI})$

- The formalization of the gate control is slightly more detailed to take into account the movement of the gate:
- go(up) and go(down): (events) commanding the gate to open or close
- open, closed, mvUp, and mvDown are the states of the gate
- the axioms for gate movement are the following:
  - When the gate in the closed state receives a go(up) command, it will move upwards for γ time units and will remain in the open position until a go(down) is issued
  - (M1) $\text{UpToNow}(\text{closed}) \wedge \text{go}(\text{up}) \rightarrow \text{Lasts}_{ie}(\text{mvUp}, \gamma) \wedge \text{Futr}(\text{Until}_{ie}(\text{open}, \text{go}(\text{down})) , \gamma))$
  - Symmetrically:
  - (M2) $\text{UpToNow}(\text{open}) \wedge \text{go}(\text{down}) \rightarrow \text{Lasts}_{ie}(\text{mvDown},\gamma) \wedge \text{Futr}(\text{Until}_{ie}(\text{closed},\text{go}(\text{up})),\gamma)$
  - Initially, i.e., before any operation takes place, the gate is open (we assume here that the gate is installed before any train arrives).
  - (M3) $\text{AlwP}_i(\neg\text{go}(\text{down})) \rightarrow \text{open}$

- The goal we want to achieve is stated by the following requirements:
  - Safety: $InI \rightarrow closed$
    Of course, we want the gate closed when the train is in the cross.
    However, we also do not want the gate to be uselessly closed. This is somewhat more difficult to formalize since a requirement such as
  - $\neg InI \rightarrow open$
    would be clearly unfeasible. Thus we define the following property:
  - Utility: $Lasted_{ii}(\neg InI, \gamma) \wedge Lasts_{ii}(\neg InI, d_M - d_m + \gamma) \rightarrow open$
- To achieve the above goal(s) we "design" the following control:
  - (c1) $go(down) \leftrightarrow Past(EnterR, d_m - \gamma)$ /* $d_m > \gamma$, of course*/
  - (c2) $go(up) \leftrightarrow ExitI$.

- All above specifies:
  - How the system components (train and gate) behave
  - What do we want from the system
  - How we design (manage) the system to achieve the desired goals
- Thus, we have all the elements to prove -or disprove!- that our design is correct:
- We need to show that all axioms k's, M's, c's, imply the safety and the utility formulas.
- To do so we need a formal deduction system for TRIO.
- TRIO has two natural formal semantics
  - An interpretation-based one
  - A deduction system
  - Both are quite intuitive and naturally derived by extension of the traditional semantics of first-order theories
    (Soundness and -relative- completeness thereof has been proved)

- TRIO's only "proper"axioms are the following (quite natural) ones:
  - **T1.** $Dist(\alpha, 0) \leftrightarrow \alpha$
  - **T2.** $Dist(\alpha, t1 + t2) \leftrightarrow Dist(Dist(\alpha, t1), t2)$
  - **T3.** $Dist(\alpha \wedge \beta, t) \leftrightarrow (Dist(\alpha, t) \wedge Dist(\beta, t))$
  - **T4.** $Dist(\neg\alpha, t) \leftrightarrow \neg Dist(\alpha, t)$
  - **T5.** $\alpha \leftrightarrow Alw(\alpha)$ if α is time independent

We are now ready for the proof of the safety property
(the proof of the utility property is left as an exercise)

- InI is equivalent, by (k10), to Since(¬ExitI, EnterI) which, by the definition of the Since operator, implies Past(EnterI, t) for some t>0. This in turn by (k6) ensures that Past(EnterR, t') for some t'>d_m and then, by (c1), that Past(go(down) t") for some t">γ, i.e, the gate was lowered more than γ time units ago. Now by (k2), (c1), and (M3) this implies Past(UpToNow(up), t"), so that at a time t" before the current time the premise of (M2) is satisfied so that its conclusion holds at that time, i.e.,
- (◆) Past($Until_{ie}$(closed,go(up)), t"-γ), with t"-γ > 0.
- But (k10) together with (c2) ensures that no go(up) command has been issued since t' time units ago up to the present time, so that, by (◆), *closed* holds now.

Let us now go back to the fundamental issue: applying FMs in the practice of industrial projects

- Let us first recall the main obstacles that hamper(ed) diffusion of FMs in industrial environments:
  - Historically: Formal models are just like "assembly languages to build complex programs". We need:
    - structuring mechanisms (modularization, abstraction, information hiding, …)
    - modifiability/reusability mechanisms
    - utilities to make their use comfortable (graphical interfaces, …)
    - supporting tools
    - In summary: most of the utilities that have been proved useful in the programming activity -typically OO machinery- are useful as well for "higher level" languages and their use
    - All this stuff is now available for most formalisms (with something still to be desired about tools)
  - Nevertheless … we are still far from wide adoption
  - Why?

- The need of high mathematical skill
- Overhead costs (really?)
- Tools inadequacy
- Lack of methods (remember that any document -whether formal or informal- is the result of a human intensive activity: such an activity must be a right blend of repetition/experience and intuition/innovation)
- The dilemmas evolution/revolution; innovation/conservation (look at the history of programming languages)
- Possible remedies?
  - Political (in a generalized sense) commitment:
    - Pursuing quality rather than time to market
    - Imposing standards, at least for critical systems
    - Making the market more mature so that cost/benefits must be reevaluated
      Anecdotes about MS SW:
      - would you buy a car with the same reliability as MS-Word?
      - Would you change a car after 3 months you bought it just because a new model with a few fancy optionals has been put into the market?

- Thinking seriously on the introduction of FMs in the industrial environment:
  - avoid overselling
  - plan carefully
  - be modest in stating the goals
  - apply incrementality at every level:
    - e.g., be happy enough with introducing formal specification and forget for a while about formal verification, mainly formal correctness proof
  - plan carefully the training process and give it the right priority (too bad experiences on this respect)
  - The best results are obtained with young unbiased people
  - Thus, in the long term, success can come mainly or only from the "school" (this happened to Pascal)

- Reference reading:
- Ciapessoni et al. "From formal models to .. Methods" .. ACM TOSEM, January 1999.

The TRIO case

- From the formalism to the (specification) language
- TRIO-based verification
  - Correctness proofs
  - Test case generation: verification and validation
  - Model checking (more recent)
- Experiences with the TRIO tools
- Case studies
- Towards exploitation of TRIO throughout the whole life cycle

The TRIO+(*) specification language

- (What we know so far of) TRIO is not a real specification language:
  - It lacks structuring mechanisms; therefore, the management of large specifications which are typical of real-life systems may become a difficult task.
  - It does not support incrementality: there is no mechanism to facilitate the construction of complex formulas through several versions at increasing level of precision and formality.
  - TRIO is similar to a Pascal-like toy programming language which is suitable to describe algorithms but not to build large programs. In other words, TRIO is a *specification language in the small*, whereas we need a *specification language in the large* as well as we need programming languages in the large.
- Most of the typical programming in the large features -OO ones in particular- are good for specification in the large as well:
  - structuring, abstraction, modularization reusability, …. are concepts that work as well for specs as for implementation
  - thus, classes, inheritance, genericity, … are good for specs too
  - TRIO+ embeds such concepts into the logic stage of the basic TRIO.
  - Furthermore, it is provided with a graphical interface to support ease of use.

- A TRIO$^+$ specification is built by defining suitable *classes*.
- A class is a set of axioms describing a system, constructed in a modular, independent way, following information hiding principles and object oriented techniques.
- Classes may be *simple* or *structured*, and may be organized in *inheritance* hierarchies.
- A *simple* class is a group of TRIO axioms, preceded by the declaration of all occurring predicates, variables, and functions.
- As a first example of simple class let us consider the class CrossingBar reported below, which includes all the axioms regarding the gate and the declarations of the related predicates, variables and constants. Here we refer to a generalized version of the railroad crossing system (GRC):
- 1. Any *finite* number of trains can enter or leave the same region at the same time instant.
- 2. In every finite interval only a finite number of trains can enter R.
- 3. There is no bound to the number of tracks (this is consistent with assumption 1).
- 4. The gate movement in the up direction can be interrupted by an opposite command (go down)

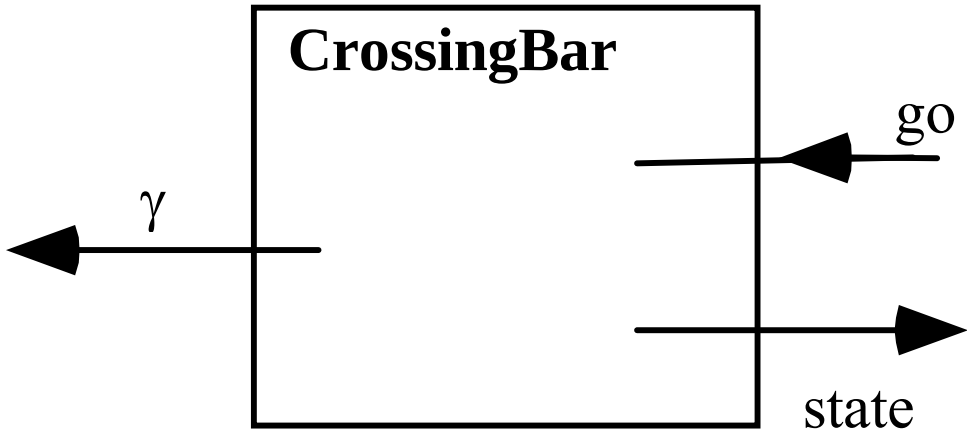
- **class** CrossingBar; -- class header
- **visible** γ, go, state; -- class interface
- **temporal domain** real; -- the temporal domain to be considered in the specification
- **TD items** -- the declarations of time dependent predicates, variables and functions
- **predicates** go({up, down});
- **vars** state: {open, closed, mvUp, MvDown};
- **TI items**
- **consts** γ: real;
- **axioms** -- the axioms of the specification (implicitly preceded by an Alw operator)
- **vars** t: real; -- the time independent variables, implicitly universally quantified
- positiveγ: $\gamma > 0$
- M3: -- when the gate moving up receives a go(down) command it inverts its motion, at the same speed reaches again the closed position and stays there until the next go(up)
- $UpToNow(mvUp) \wedge go(down) \wedge LastTime(go(up),t) \rightarrow$
- $Lasts_{ie}(mvDown,t) \wedge Futr(Until_{ie}(closed,go(up)),t)$
- Other axioms M1, M2, M4, and M5 are reported similarly ... see below
- **end** CrossingBar.

- For the sake of completeness:
- (M1) When the gate in the closed state receives a go(up) command, it will move upwards for γ time units or until a go(down) is issued:
  $UpToNow(closed) \wedge go(up) \rightarrow Until_{ie}(mvUp, go(down) \vee Lasted_{ie}(\neg go(down), \gamma))$
- (M2) When the gate in the open state receives a go(down) command, it will move downwards for γ time units and then stay closed until a go(up) command is issued:
  $UpToNow(open) \wedge go(down) \rightarrow Lasts_{ie}(mvDown,\gamma) \wedge Futr(Until_{ie}(closed,go(up)),\gamma)$
- (M3) When the gate moving up receives a go(down) command it inverts its motion, at the same speed reaches again the closed position and stays there until the next go(up):
  $UpToNow(mvUp) \wedge go(down) \wedge LastTime(go(up),t) \rightarrow$
- $Lasts_{ie}(mvDown,t) \wedge Futr(Until_{ie}(closed,go(up)),t)$
- (M4) After moving up for γ time units, if there is no go(down) command the gate stays open until the next go(down) command:
  $Lasted_{ie}(mvUp, \gamma) \wedge \neg go(down) \rightarrow Until_{ie}(open, go(down))$
- (M5) Initially, i.e., before any operation takes place, the gate is open (we assume here that the gate is installed before any train arrives):
  $AlwP_i(\neg go(down)) \rightarrow open$

Explanations and comments

- The class header is followed by the **visible** clause, which defines the class interface. In the example, γ, *go* and *state* are the only visible symbols of the class.
- The **temporal domain** clause indicates the temporal domain of the specification.
- The keyword **TD Items** is followed by the declarations of the local time dependent functions, predicates and variables;
- the keyword **TI Items** is followed by the local time independent functions, predicates and constants. In the example γ is a real constant, *go* is a unary time dependent predicate on the set {up, down}, *state* is a time dependent variable whose value ranges on {open, closed, mvUp, mvDown}.
- The **axioms** are TRIO formulas. Comments, in the form of natural language sentences, are allowed, although they do not have any formal semantics.
- The TRIO formulas of the **axioms** are prefaced with an implicit universal classical and temporal quantification, i.e., all free variables are universally quantified and an *Always* temporal operator precedes the formula.
- A name can precede an axiom, to be used as a reference for axiom redefinition in inheritance. The name must not conflict with other names used in the class declaration. A **vars** clause after the keyword **axioms** lists the declaration of time independent, local variables.

- A class may have a graphic representation as a box, with its name written at the left top; the names of the items are written on lines internal to the box; if an item is visible, then the corresponding line continues outside the box. Class *CrossingBar* is represented in Figure below

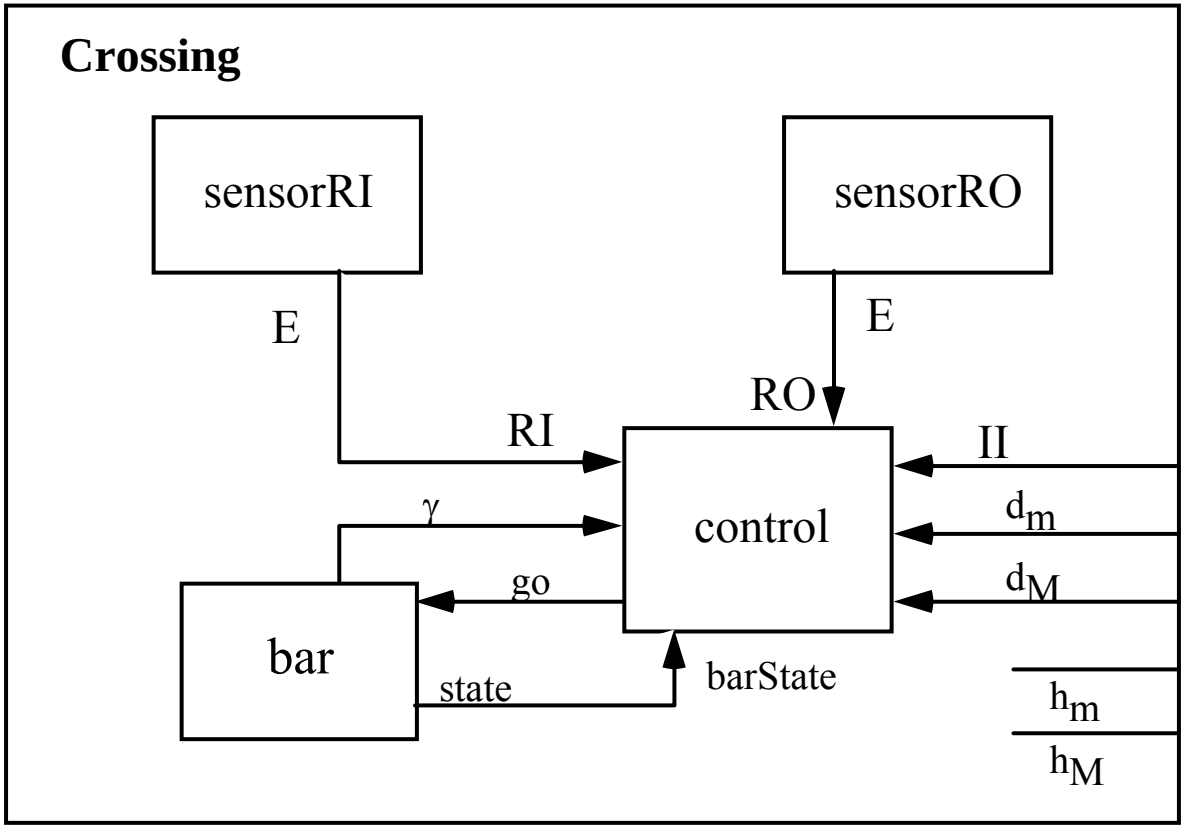


- *Structured classes* are classes that have components–called *modules*–of other classes. They permit the construction of TRIO$^+$ modular specifications, especially suited to describe systems in which *parts* can be easily recognized. In our example, the structured class *Crossing* models the railway crossing: it includes modules for the gate, the sensors at the RI and RO boundaries, and the control system.
- Each component corresponds to a module declared in the *modules* clause by indicating its class (the classes *sensor* and *RCControl* of modules sensorRI, sensorRO, and control will be declared subsequently). Items of the structured class, like II (a train enters region I), d_m, d_M, h_m, and h_M, may appear in the class axioms.
- Modules may not be used directly in axioms, since they represent a set of items and module definitions, with related axioms. Visible items of the local modules, denoted by using a dot notation, can however appear in axioms of the enclosing class: for instance, 'control.RI' appears in axiom B1 of class Crossing.
- The *connections* clause specifies equivalence or identity between pairs of items in the scope of the current class, and may often be interpreted as information flows between components. E.g., the connection (gate.state, control.barState) is: Alw(gate.state = control.gateState). This asserts that the value of the position of the gate is instantly known to the controller.

- **class** Crossing;
- **temporal domain** real;
- **TD items**
- **predicates** II(natural);
- **TI items**
- **constants** d_m, d_M, h_m, h_M: real;
- **modules** sensorRI, sensorRO: sensor;
- gate: crossingGate;
- control: RCControl;
- **connections** { (sensorRI.E, control.RI),
- (sensorRO.E, control.RO),
- (II, control.II),
- (gate.γ, control.γ),
- (gate.go, control.go),
- (gate.state, control.gateState),
- (d_m, control.d_m),
- (d_M, control.d_M) }
- **axioms ….**

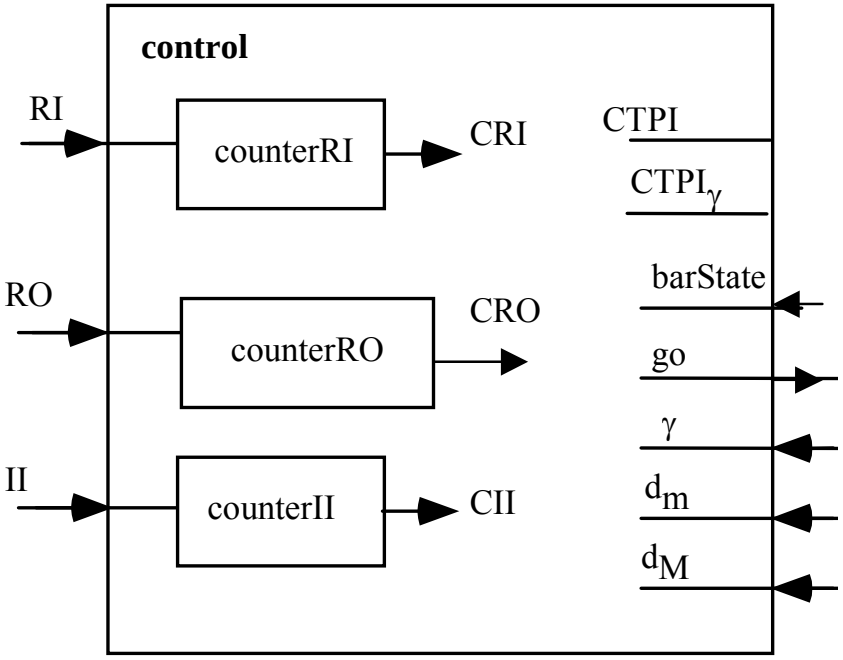
- ….
- **axioms**
- **vars** k: natural;
- t: real;
- positiveTimeConsts: $d_M > d_m > 0 \wedge h_M > h_m > 0$
- B1: -- the k-th train enters region I with a delay between d_m and d_M after the k-th train enters R
- control.RI(k) $\rightarrow \exists t\ (d_m \leq t \leq d_M \wedge Futr(II(k),t))$
- -- Axioms (B2)÷(B4) below have a similar meaning:
- (B2) $\forall k(II(k) \rightarrow (\exists t\ d_m \leq t \leq d_M \wedge Past(RI(k),t))$
- (B3) $\forall k(II(k) \rightarrow (\exists t\ h_m \leq t \leq h_M \wedge Futr(RO(k),t))$
- (B4) $\forall k(RO(k) \rightarrow (\exists t\ h_m \leq t \leq h_M \wedge Past(II(k),t))$
- **end** Crossing.

Graphical version of the Crossing class



Remark: A connection is pictorially represented by a line joining the two items. If these have the same name, then it is repeated only once, near the linking line. The advantage of using a connection in place of an axiom is that a connection has an intuitive, pictorial notation, while an axiom does not.

Let us go a little further in the -top-down- development of class definition



- **class** RCControl;
- **temporal domain** real;
- **visible** RI, II, RO, d_m, d_M, go, gateState, γ;
- **TD items**
- **predicates** RI(natural);
- II(natural);
- RO(natural);
- go({up, down});
- **variables** gateState: {open, closed, mvUp, mvDown};
- CTPI, $CTPI_\gamma$: natural; --CTPI are counters to be defined next
- **TI items**
- **consts** d_m, d_M, γ: real;
- **modules** countRI, countII, countRO: counter;
- **connections** { (RI, countRI.E),
- (II, countII.E),
- (RO, countRO.E) }
- ...

- ...
- **axioms**
- ...axioms defining CTPI and $CTPI_{\gamma}$, plus axioms (C1) and (C2) defining the commands ...
- safety: -- safety property for the GRC: if the number of trains that entered the critical region exceeds the number of trains that exited, then the gate must be closed
- counterII.C > counterRO.C $\rightarrow$ gateState = closed
- **end** RCControl.

- Remark: The last "axiom" in class RCControl is an interesting example of a *property* which does not describe directly the required system behavior but expresses some desired system properties that are *ensured* by the other axioms. Such special "axioms" should be called, more appropriately, *properties*, or *theorems*.
  They are the counterpart of B's Invariants. TRIO+ -unfortunately- does not -yet- distinguish between basic and derived properties (axioms and theorems).
  This is essentially due to TRIO's history which privileged industrial application w.r.t. theoretical elaboration.

- **class** counter;
- **temporal domain** real;
- **visible** E, C;
- **TD items**
- **predicates** E(natural);
- **variables** C: natural;
- **axioms**
- **vars** k: natural;
  - (A1) $\text{Counter}(C,E) =_{def}$ $\text{Alw}(\forall k (k \geq 1 \rightarrow ((C = k) \leftrightarrow (\text{SomP}_i(E(k)) \wedge \neg \text{SomP}_i(E(k+1))))) \wedge$
- $\text{Som}(C=0 \wedge \text{AlwP}(C=0))$
- **end** counter.

- The class sensor includes the axioms characterizing the *events* generated by the sensors in the railway crossing. One axiom, called *nonZeno* specifies the requirement that at any time only a finite number of trains will cross any region boundary.
- **class** sensor;
- **temporal domain** real;
- **visible** E;
- **TD items**
- **predicates** E(natural);
- **axioms**
- **vars** k, j: natural;

(E1) –– Event *occurrences* are *unique*, i.e., they can be true at most in a single instant within the time axis.
$\forall k(E(k) \rightarrow (AlwP\ (\neg\ E(k)) \wedge AlwF\ (\neg E(k))))$ -- E(k) denotes the k-th occurrence of event E

(E2) ––Events must satisfy a (weak) monotonicity of parameter k with respect to time flowing, i.e.:

$\forall k(E(k) \wedge k > 1 \rightarrow SomP_i(E(k-1))$

- (E3) -- By convention: $Alw\ (\neg\ E(0))$
- nonZeno: -- at any time only a finite number of trains can enter or leave any region

$\exists k(\forall h\ (h > k) \rightarrow \neg E(h))$

- **end** sensor.

Inheritance in TRIO+ (hints)

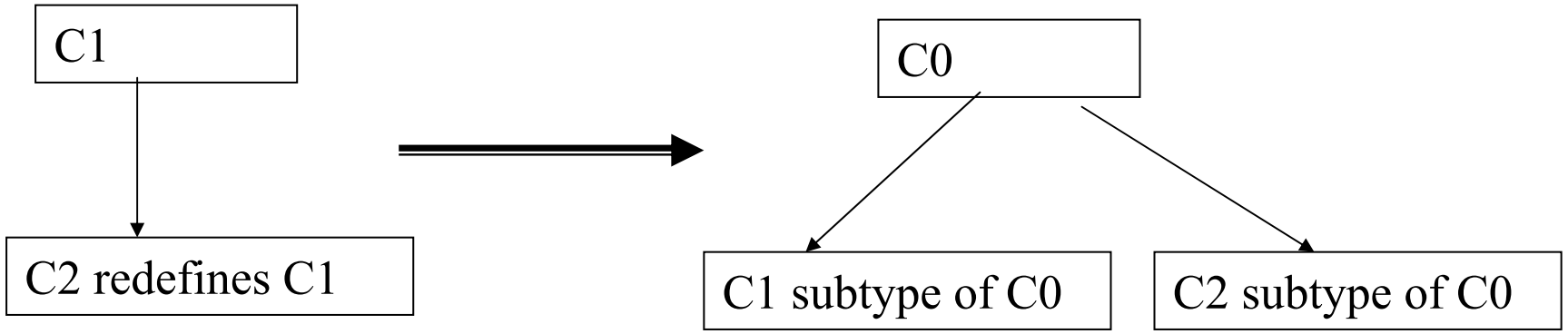
- As usual, inheritance provides the possibility for a class—also named *subclass,* or *descendant,* or *heir class*—to receive attributes from other classes—named its *superclasses* or *ancestors*. To permit the correct evaluation of inherited axioms, inheritance in TRIO$^+$ requires a form of *monotonicity of the external interface* (a visible item must remain visible in the descendants), and the *monotonicity of the data components* (inherited items cannot be canceled). On the other hand, TRIO$^+$ permits to redefine freely the inherited axioms, i.e., to change completely the semantics of the descendants.
- This possibility is particularly useful for a specification language, which must be flexible in the first place because modeling and requirement specification are often exploratory activities : see later.

In our example:

- Suppose we wish to specify a class sensorWithNTracks, i.e., a kind of sensor to be used in a crossing where the number of tracks is not left unbound as in the general case but is assumed to be equal to a given constant *nTracks*. This variation in the sensor specification is modeled by means of the inheritance construct.
- The class sensorWithNTracks inherits from class sensor; it adds the constant item *nTracks* modeling the number of distinct tracks in the crossing, and redefines the axiom *nonZeno* by stating that no more than *nTracks* simultaneous events of type *E* can occur at any time, thus imposing this number as a limit on the number of trains that can simultaneously cross any region boundary.
- **class** sensorWithNTracks;
- **inherits** sensor **[redefine** nonZeno**]**
- **TI items**
- **consts** nTracks: natural;
- **axioms**
- nonZeno: -- no more than *nTrack* trains are simultaneously sensed by the sensor
- $E(k) \wedge E(j) \rightarrow k\text{-}j < nTracks$
- **end** sensorWithNTracks

Hints towards TRIO's evolution

- Higher-order TRIO (HOT): Class = Type! → modules can be items → they can be quantified
- Strictly monotonic inheritance → equals subtyping.
- However free inheritance is kept to allow flexibility
- Methodological hints towards the managing of inheritance:



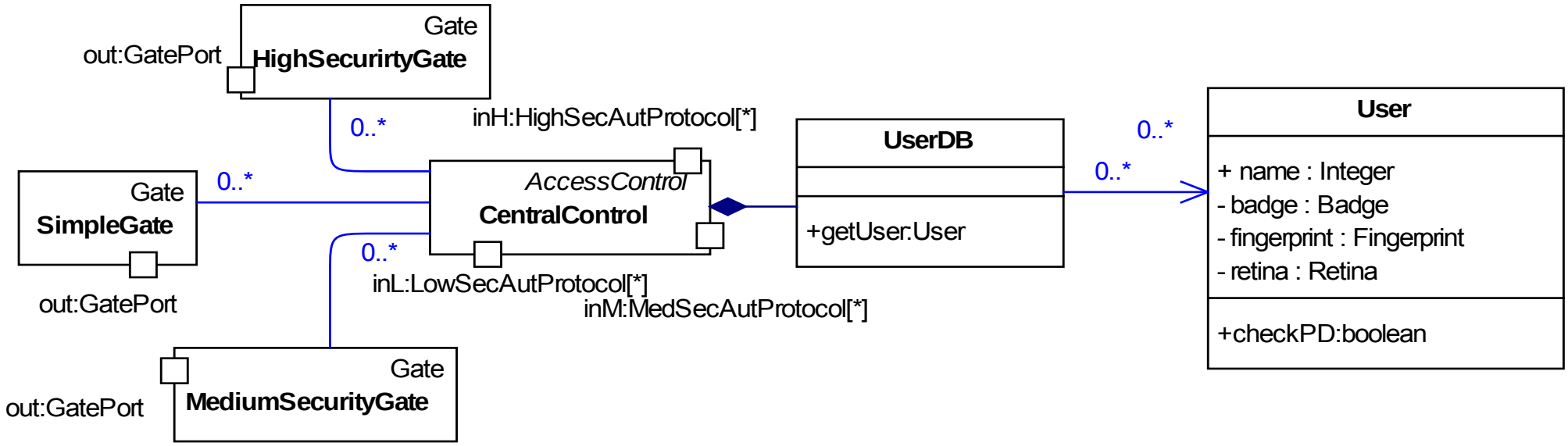
A brief overview of ArchiTRIO

- The basic idea:
  a user starts by drawing a UML (class/structure) diagram and then refines/specializes/completes it until (s)he obtains a complete architecture, consisting of Composite Structure diagrams and their ArchiTRIO semantics, possibly augmented with exclusively ArchiTRIO concepts, on which formal verification can be carried out.

Running example: Building Access Control System

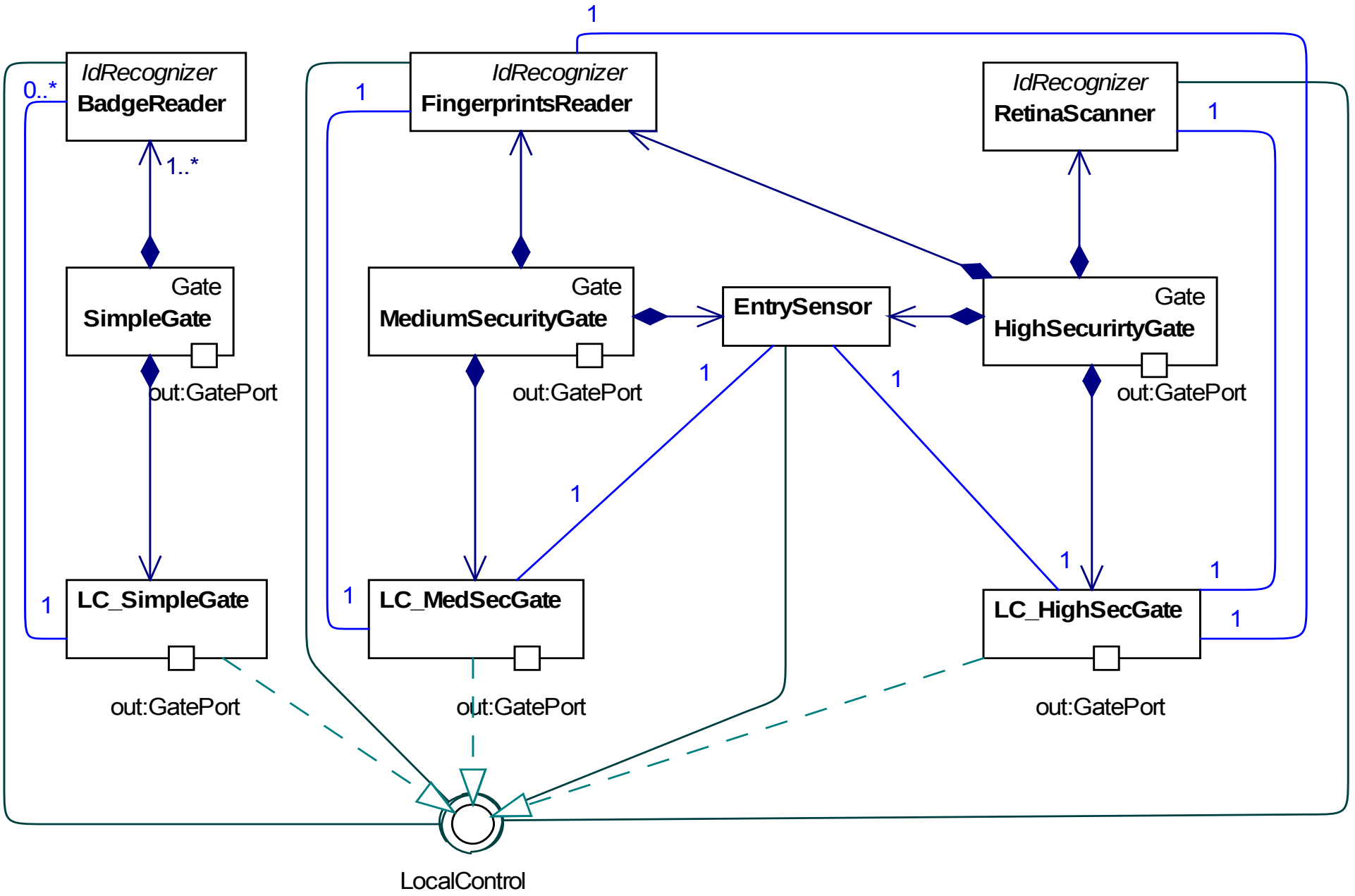
- Three different security levels: *low*, *medium*, and *high*.
- The building may contain zero or more areas of a given security level.
- The access control is enforced essentially through two kinds of entities: a local mechanism based on the concept of *security gate*, and a *central control* connected to a user database.

First, high level UML view



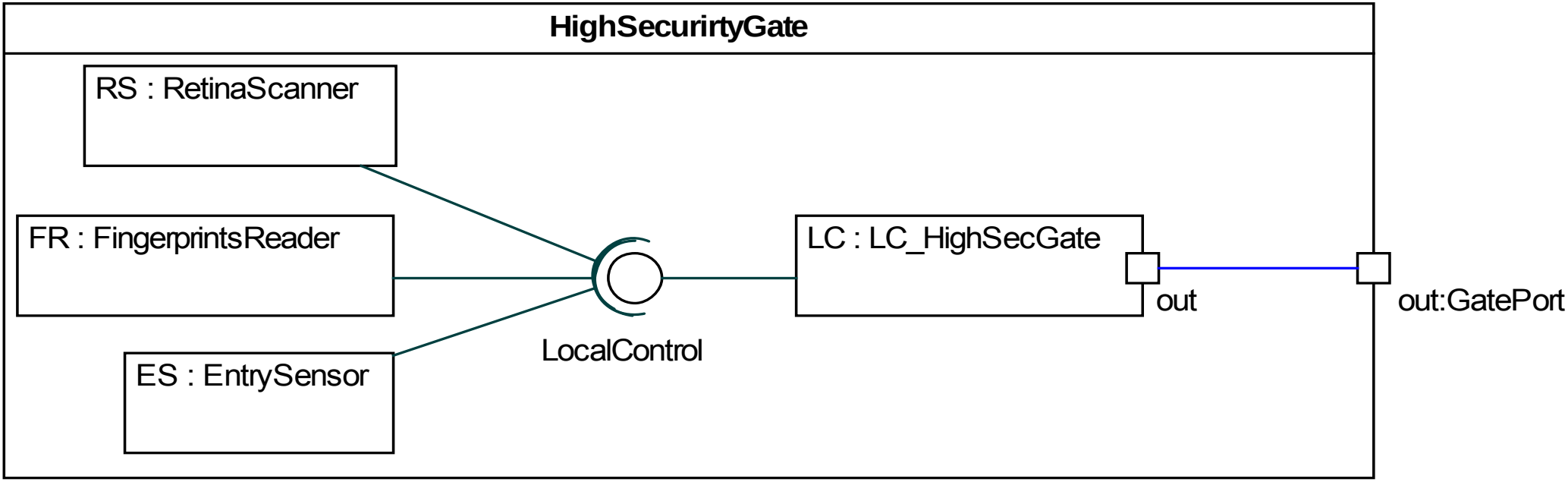
- UML 2.0 introduces the useful concept of *port*, which is essentially an interface container. In this example ports are used to define the protocols used by the CentralControl, to get from, send data to, and manage gates. In Figure 1, every gate has a port of type GatePort, while CentralControl has three different ports, LowSecAutProtocol, MedSecAutProtocol, and HighSecAutProtocol that are used to communicate with SimpleGates, MediumSecurityGates, and HighSecurityGates, respectively.

Moving in a top-down fashion, …



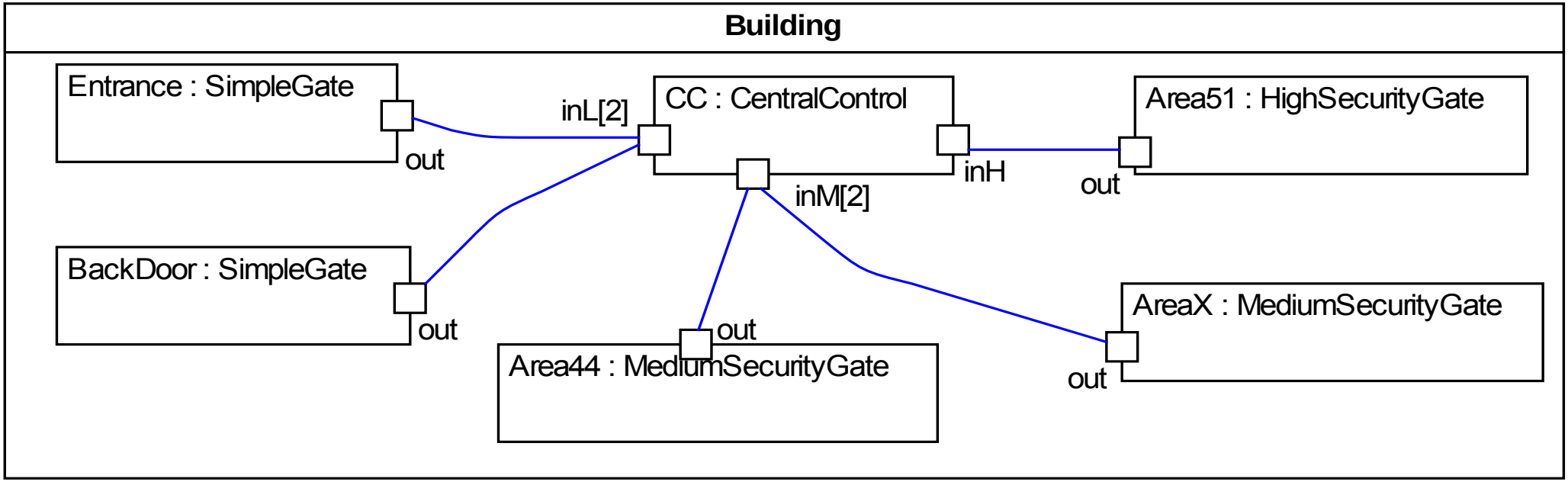
- The low security gate is the simplest one, and is depicted in the left part of the diagram. A SimpleGate is an entity having one or more BadgeReaders (a subclass of IdRecognizer), managed by a local controller LC_SimpleGate. Communication between BadgeReader and LC_SimpleGate is based on the interface LocalControl, implemented by the latter.
- The medium security level gates are described in the central part of the diagram. A MediumSecurityGate is based on a more sophisticated IdRecognizer, a fingerprints reader (class FingerprintsReader), and has an entry sensor (class EntrySensor). In the typical usage scenario of a medium security gate, the user approaches the gate and his/her fingerprints are scanned; his/her data is then sent to the central control to be checked. If control is passed, the gate remains open either for a short fixed time interval, or until the entry sensor actually detects the user getting in. This scenario could typically be described in UML by a sequence diagram, not reported here. Analogously to the simple gate, a medium security gate is supervised by a local controller, LC_MedSecGate, and communication between the local controller and the sensors is based on the interface LocalControl.
- The most complex type of gate is the HighSecurityGate, on the right side of the Figure: it consists of two kinds of IdRecognizers, a FingerprintsReader and a RetinaScanner; an EntrySensor; and a local controller LC_HighSecGate. Its behavior is basically analogous to the medium security level one, but for the retina scanner: the access control has to check both the user's fingerprints and retina to open the gate.

From class diagrams to *structure* diagrams



Structure diagrams are more oject-oriented than class-oriented:
Every component is an instance of the corresponding class

The whole building structure



Structure diagrams are pretty much the same as TRIO+ diagrams

- UML *per se* does not precisely define many of the constructs we used for describing our system here.
- For instance, a precise definition of timeouts management and local control behavior is nowhere to be found.
- More generally, we would like to be able to precisely express a critical property and possibly to verify it.
- In our example an unwanted behavior like the following should not be possible: Alice has clearance to enter Area51 and authenticates herself at the gate, at the same time a malicious Bob is waiting for her authentication behind a corner nearby, trying to enter into the restricted area right behind her. On one hand it is easy to correctly model the local control by using behavioral diagrams (e.g. statecharts or SDL); on the other hand however, general properties, such as “the entry sensor must signal a single entrance after a valid authentication, and it must occur not before k ms and not after $k+n$ ms”, are very hard to express, let alone to verify, if one uses pure UML.

At this point TRIO comes into play: UML + TRIO = ArchiTRIO

- Class LC_HighSecGate provides interface LocalControl and has a port of type GatePort; interface LocalControl defines two operations, incomingData and personEntered.
- In addition to the aforementioned UML port and interface, class LC_HighSecGate includes three logic items, inGate, lastUser and gate_open .
  - Item inGate is time-independent and represents the identifier of the Gate to which the controller belongs;
  - item lastUser is time-dependent and models the data corresponding to the user who had either his/her fingerprints or his/her retina scanned;
  - item gate_open, instead, is a state and models the intervals in which the gate is open.

- **Axiom dataRelay** states that when an invocation of operation incomingData (exported through interface LocalControl) is received by the controller and the value of the rawData parameter is pd, within T time units in the future the controller will invoke (an instance of) operation sendPersData on port out, passing pd and the value corresponding to item inGate as parameters.
-
```
vars: iD  : incomingData;
      sPD : sendPersData;
      pd  : PersonalData;
dataRelay:
  iD.inv_rec(pd) ->
    ex out.sPD(WithinF(out.sPD.invoke(pd, inGate), T);
```

- A *port* is a collection of provided and required interfaces.
  - It can be used to define a *protocol*, intended as a combination of invocations of operations that can be received (from a provided interface) or issued (to a required interface).
  - Thus, an ArchiTRIO port can contain axioms defining the corresponding protocol in terms of the involved operation invocations .

- HighSecAutProtocol provides interface AccessControl, and requires one instance of interface FromAccessControl.

interface AccessControl
operations:
User sendPersData(**in** PersonalData rawData,
in GateId gate)
raises UserNonExistentException;
enterPerson(**in** User user,
in GateId gate)
raises UserNonExistentException;
end

interface FromAccessControl
operations:
openGate();
end

The port defines the authentication protocol for gates that require that a user authenticates him/herself through both a fingerprint and a retina scan: the two scans can occur in any order, but always within a maximum delay one from the other for the authentication to be successful (i.e. for the controller to allow the user to enter by opening the gate through an openGate command).
Exercise: formalize through TRIO axioms

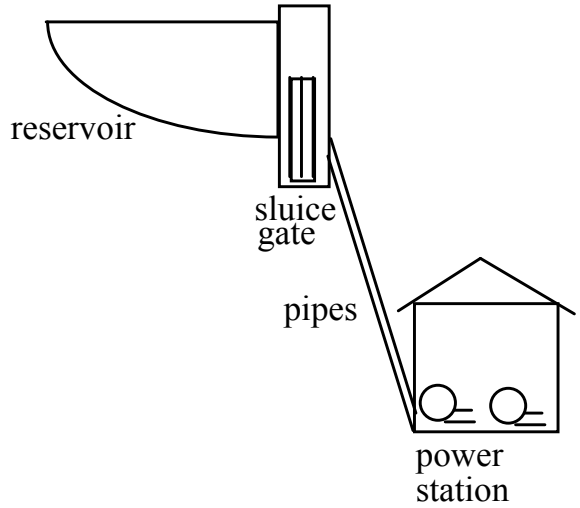
Case stud(ies) from industrial experience(s)

1. From an established, informal, specification method to the exploitation/introduction/addition of a formal (TRIO-based) approach
2. Dealing with fault tolerance requirements
3. Dealing with -specification- configurability

4. Towards extending the approach to other design phases

1. From informal to formal specs: a smooth approach

- The informal IEC standard:
  - System description: static and dynamic views:
    - The static view identifies the main components (the environment in which it operates, the process or plant to be controlled, ...) and the relations occurring between them (e.g., physical and logical connections).

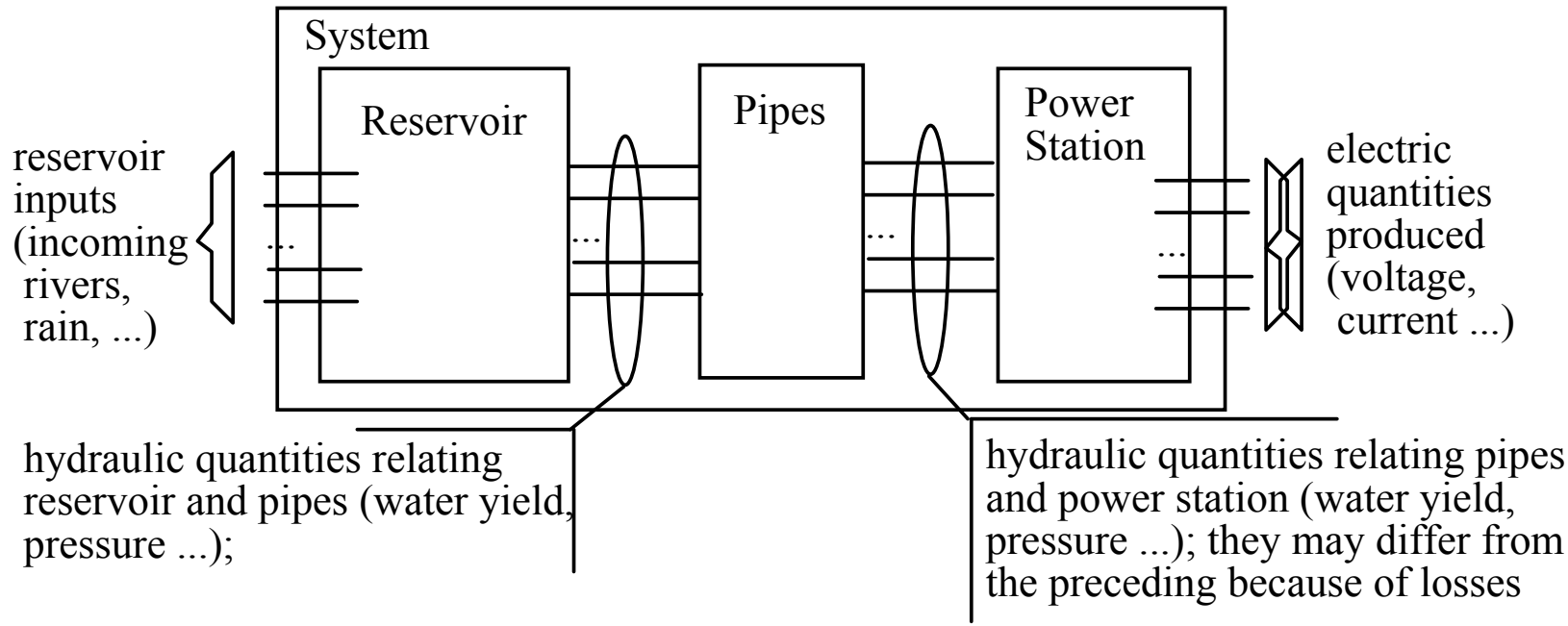


- The dynamic view defines all the relevant system properties. In most cases such properties are expressed in an *operational style*, that is by means of *state variables* (e.g., temperatures, pressures, the level of a reservoir, the power produced by a plant, …) and *state transformations*, or *transitions* (e.g., the opening or closing of gates or valves, the switching of electric connections, ...).
  Since the relevant properties of SCS are usually quite numerous and may differ deeply in scope and relevance, they are structured as follows:
  - **Functional properties,** related to data monitoring and elaboration. For instance: "the Acquisition and Pre-processing resource must periodically acquire from plant sensors the measurements of 31 analog variables (mostly pressures and temperatures) and 3 digital variables".
  - **Performance properties**. A typical example is the time within which an alarm must be managed.
  - **Safety properties**, including system protection from external disturbances, built-in safety mechanisms and protections against risks, suitable safety rules for the personnel.
  - **Relevant external conditions** under which the system has to operate during its mission. For example, an alarm managing module may assume that sensors –and, therefore, the controlled plant– do not generate more than 10 alarms per second: under this hypothesis its requirements state that all of them must be served within 10 seconds.
  - **Definition of system behavior under permanent or transient faults**.
    under normal conditions, the module must guarantee some given requirements;
    under exceptional conditions it must instead guarantee some other requirements.

- The above properties are grouped into a larger category, named *operating properties*. There are also other types of properties specifying managing and maintenance procedures, services supplied by vendors and users, documentation requirements, training procedures, HW and SW compatibility, communications, etc.
- Among these other properties special attention is devoted to *configurability requirements*, since system configurability is a fundamental quality for most industrial systems. Designing a configurable system is a typical example of the *design for change* principle stated by Parnas. In fact, in most cases a system is a member of a family of similar but not identical elements.
  For instance, in the case of hydroelectric power systems many similar plants may differ in the number, type, and altitude of reservoirs, in the number and power of power stations, etc.
- The result of system analysis is a *specification document* which is structured into several chapters and sections reflecting the above categorization.

Formalizing the specification document

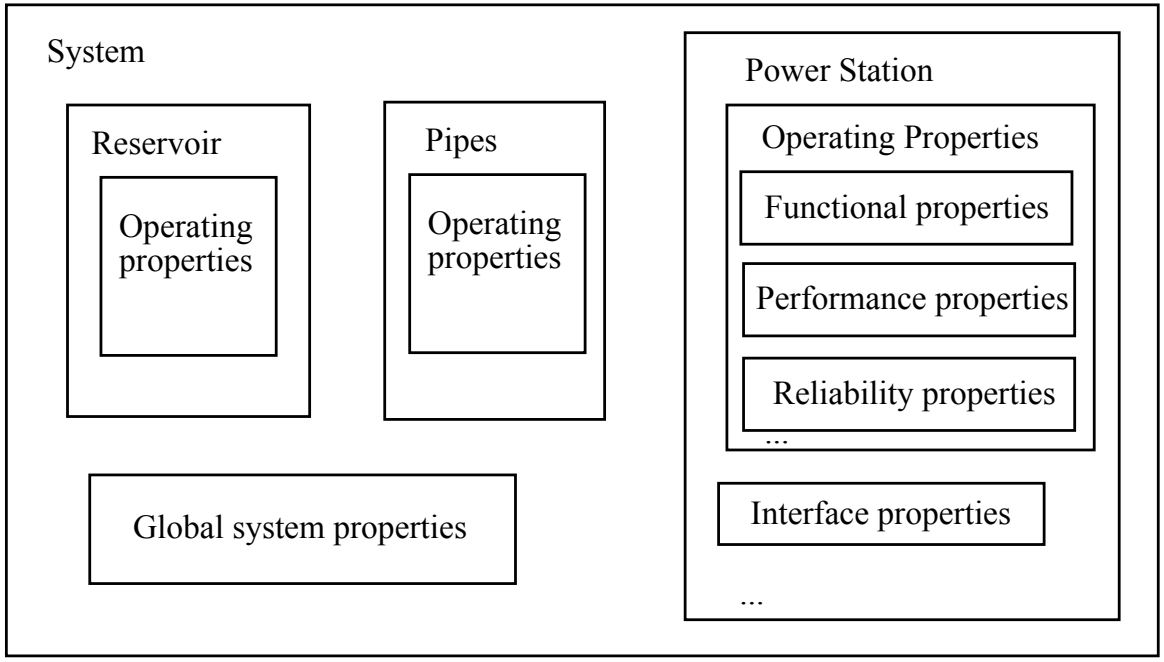
- First step: writing a collection of boxes that correspond one-to-one to system's main components:



- Remarks and comments ---> methodological suggestions
  - Strictly speaking, TRIO is more class-oriented than object-oriented, since a specification consists of a collection of class rather than object definitions: in most cases we are not interested in specifying the properties of a single object. Only when configuring a particular system we describe the specific features of subclasses possibly down to single plants.
  - TRIO does not have an explicit notion of input/output relations among modules. Thus, for instance, an object "monitor" receiving an alarm from the environment and reacting by providing a shutdown command to the plant, can be formally expressed by an item *alarm* and an item *shutdown* in the box *monitor*. However, the user can augment lines denoting items by giving them a direction (denoted by an arrowhead).
  - The *connections* between TRIO classes must not be confused with possible *physical links* between the objects denoted by those classes: a TRIO connection simply denotes that the two items represented by the line segment (a level measure, a signal, …) are the same thing. Therefore, a physical link between objects, such as a pipe connecting a reservoir with a power station or an electrical junction between circuit components, does not necessarily result into a TRIO connection in the specification document.
  - …. more later.

Formalizing system properties

- Once we have formalized the system structure by means of a suitable collection of TRIO classes, items, and connections we move to the formalization of the system properties.
- This activity too follows the structure of the original informal document: first, we group the (usually numerous) properties into several modules and sub-modules.
- This is accomplished by using classes, but in a different way than when defining the system structure:



- Notice that it is not necessary that every system component specifies every type of properties: for instance, interface properties make sense only for those components that involve some human-machine interaction.
- Finally, we move from "specification in the large" (i.e., system structure and properties) to "specification in the small", where we formalize single properties using pure TRIO formulas. To simplify at best this critical and often difficult activity, we used the following guidelines:
  - ***Not everything must be formalized***
  - ***Use an operational attitude***
  - ***Introduce some "thumb-rules"***
  - In order to help users who are not familiar with the formal notation, the language manuals should include a collection of “thumb-rules” to move from prose description to formal specification, pointing out a few typical traps and suggesting how to avoid them.

A few thumb rules derived from experience

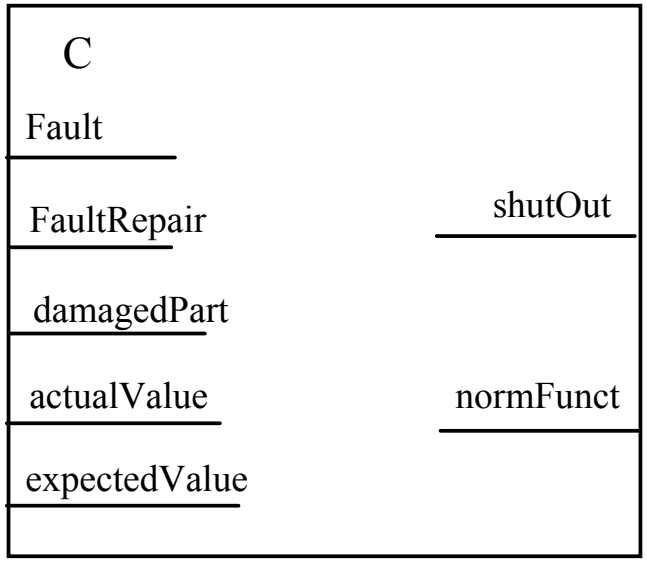
- Translate an informal "such that" into an implication when it is paired with a universal quantification and into a conjunction when it is paired with an existential quantification.
  "a reservoir located in the Alps must not have an area greater than k m^2" becomes
  $\forall r\ (alpine(r) \rightarrow area(r) \leq k))$
  "there are reservoirs in the Alps that are located at an altitude greater than h" becomes
  $\exists\ r\ (alpine(r) \wedge altitude(r) \geq h))$.
- Avoid –whenever possible– the use of explicit quantifiers when specifying time-related properties: use instead derived temporal operators.
  For instance the formula *Lasts(P, d)* is more immediate and understandable than
  $\forall d'\ (0 < d' < d \rightarrow Dist(P, d'))$.

- Clearly, the usefulness and the style of such suggestions strongly depend on the cultural background of the industrial environment where the formal method must be adopted.

2. Dealing with fault tolerance requirements

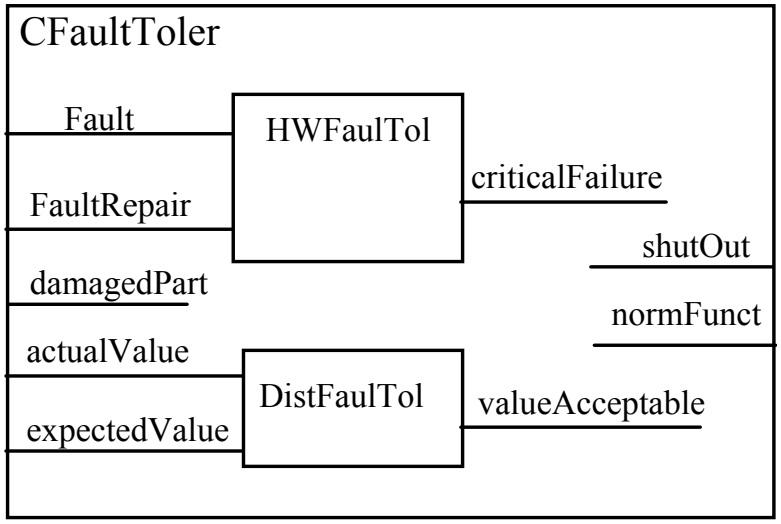
- It is useful, from a methodological point of view, to distinguish requirements related with fault tolerance from requirements regarding system operation in absence of faults. ------>
  structuring the specification into parts describing “normal” functioning and other parts describing “exceptional” or “faulty” conditions
- **Example**
- Suppose we want to specify a system subject to both hardware faults and disturbances which tend to change the value of a state variable, thus causing incorrect system operation. Informally, the basic requirements on the system behavior are specified as follows:
  - a)When a hardware fault occurs and remains unrepaired for at least *delta* time units, the system must be able to find the damaged part and put it off-line;
  - b)The system operates normally only if the value of the state variable is not altered by a disturbance.

- In order to formalize the above requirements which involve the distinction between normal and faulty operation, it is necessary to distinguish items into *actualValues* and *expectedValue*:

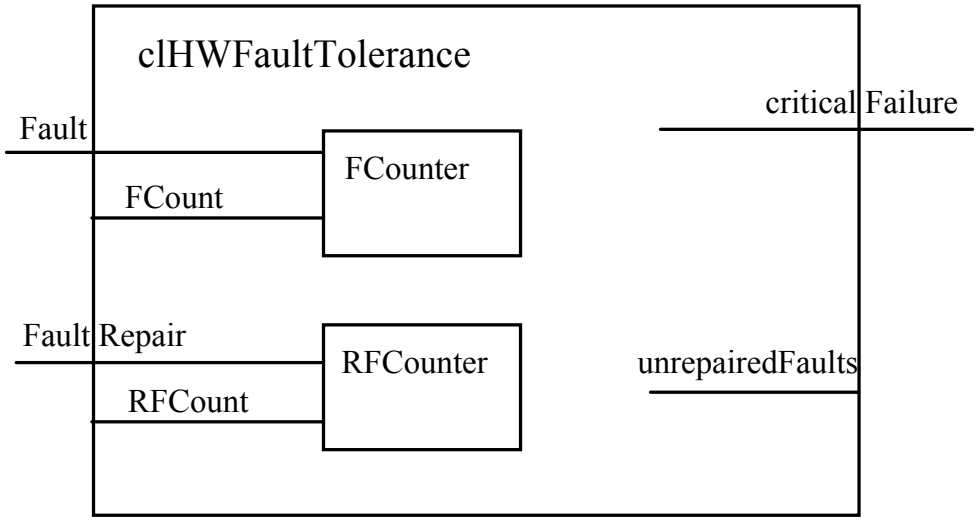


- **class** C
- .... declarations corresponding to the graphical representation of Figure above
- **axioms**
- /*normFunct is the proposition that models the normal functioning of the system: it is defined as equivalent to the conjunction of C_1 ... C_n, the other axioms describing normal functioning in ordinary conditions */
- normFunct $\leftrightarrow$ $C_1 \wedge$ ... $\wedge C_n$
- /*the axiom below formalizes requirement a) */
- whenShutOut: shutOut(damagedPart) $\leftrightarrow$ Lasted(Since(¬FaultRepair, Fault), delta)
- /*the axiom below formalizes requirement b) */
- disturbFunct: normFunct $\rightarrow$ actualValue = expectedValue
- **end** C

- Suppose now that we need to specify the following *additional* fault tolerance properties to make the system more robust in presence of faults:
  - a)the damaged part should be put off-line only whenever the second unrepaired hardware fault occurs (it is assumed that the system can be managed correctly in presence of a single unrepaired fault) and stays unrepaired for a given time;
  - b)the system should operate normally if the value of the state variable, though recently altered for some time intervals, maintains the expected value for at least half of the time during the last *delta1* time units.
- We specify these stronger fault tolerance requirements by defining a *heir class* of C, called *CFaultToler*, that inherits from class *C* and adds two modules, *HWFaulTol* and *DistFaulTor* that include the specification of properties of fault tolerance with respect to hardware faults and disturbances on the state variable, respectively.



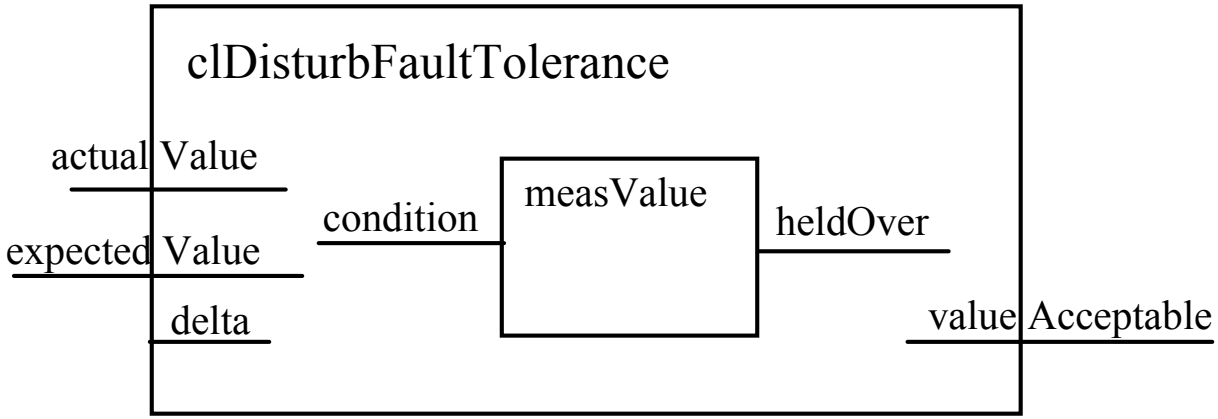
- Let us first consider the property of tolerance to hardware faults.
  The module HWFaulTol is of class *clHWFaultTolerance*, defined by :



class clHWFaultTolerance
... item and module declarations ...
axioms
unrepairedFaults = FCount - RFCount
criticalFailure $\leftrightarrow$ unrepairedFaults ≥ 2
end clHWFaultTolerance.

- Class *clHWFaultTolerance*, in turn, includes two modules, *FCounter* and *RFCounter*, of class *EventCounter*, to model the counting operation of the events *Fault* and *FaultRepair*, respectively. A critical failure (i.e., one originated by some faults that cannot be “tolerated”) occurs when the number of unrepaired faults (defined as the difference between the number of faults and the number of fault repairs) is greater than or equal to 2.
- *EventCounter* is a typical and well known example of *library class* definition: it includes axioms requiring that a generic predicate models an event, and specifying the counting operation.

- In a similar manner the module *distFaulTol* of class *clDisturbFaultTolerance* specifies the tolerance to disturbances on the value of the state variable



class clDisturbFaultTolerance

.... module and item declarations as in the graphical representation

...

axioms

measValue.condition ↔ (actualValue = expectedValue)

valueAcceptable ↔ measValue.heldOver(delta1) ≥ delta1/2

end clDisturbFaultTolerance.

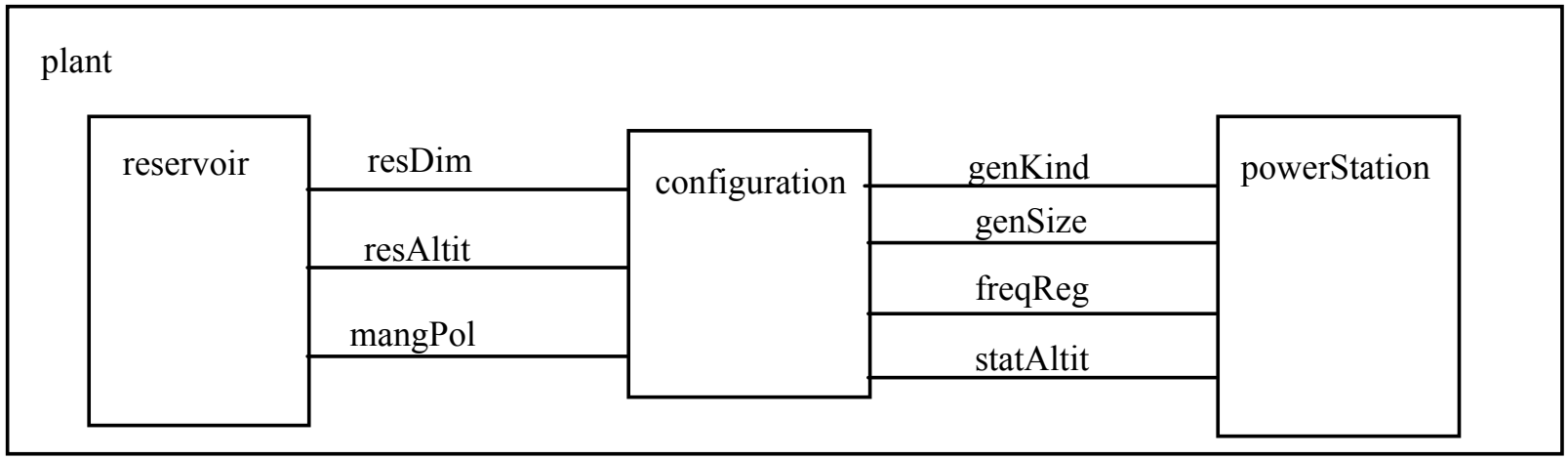
Module *measValue* of (library) class *measure* specifies that the actual value of the state variable has been equal to the expected value for at least half of the last *delta1* time units. Function *heldOver* determines the total amount of time over which a given condition holds, for each possible length of the time interval ending at current time.

- Finally, class *CFaultToler* redefines the axioms *whenShutOut* and *disturbFunct* describing the new conditions under which a hardware fault causes a shut out of the damaged part, and a disturbance of the value of the state variable hinders normal functioning. Notice that the axioms *whenShutOut* and *disturbFunct* of class *C* are redefined in *CFaultToler* in a similar way, by taking into account the items exported by modules *HWFaulTol* and *DistiFaulTol* which specify the fault tolerance requirements.

– **class** CFaultToler
– **inherit** C [**redefine** whenShutOut, disturbFunct]
– .... item declarations and connections according to the representation of corresponding figure
– **axioms**
– /*the damaged part is put off line iff a critical failure persists for at least *delay* time units*/
– whenShutOut: shutOut(damagedPart) $\leftrightarrow$ Lasted(HWFaultTol.criticalFailure, delay)
– /*functioning is normal only if the comparison between the actual and the expected value gives an acceptable result*/
– disturbFunct: normFunct $\rightarrow$ DistFaultToler.valueAcceptable
– **end** CFaultToler.

3. Dealing with -specification- configurability: a "static configuration" example

- A pondage power plant includes the modules *reservoir* and *powerStation* representing its physical components. To deal explicitly with its possible configurations, we add a module *configuration* that includes all information on the plant configuration. The configuration module does not correspond to a physical plant component: it is adopted as a *systematic* way to encapsulate the requirements concerning the allowed possibilities and the existing constraints on composing various kinds of plant components, thus determining the possible plant configurations.



- *resDim*, the reservoir dimension: large, medium, or small; the size of the reservoir is directly related to its maximum water delivery;
- *resAltid*, the altitude where the pondage is located.
- *mangPol*, the managing policy adopted for the reservoir: it can be *autonomous*, (i.e., the water of the pondage is managed according to a production program defined autonomously by the plant), or *external* (i.e., the water is managed according to a production program that is defined outside the plant).
- A station for power generation is characterized as follows:
- *genKind* the type of the power generating units in the station: there exist two kinds of unit: Francis and Pelton, which are respectively more suitable for low or high leap (difference in altitude) from the pondage to the station;
- *genSize*, the size of the generating units in the station, which depends on their nominal power (number of Volt–Ampere); in pondage power plants one can have only small (generated power < 10 MVA) or medium size (≥ 10MVA) power units;
- *freqReg*: the power units may (or may not) be able to finely adjust the frequency of the generated electric voltage around the nominal value of 50Hz;
- *statAltid*, the altitude of the station.

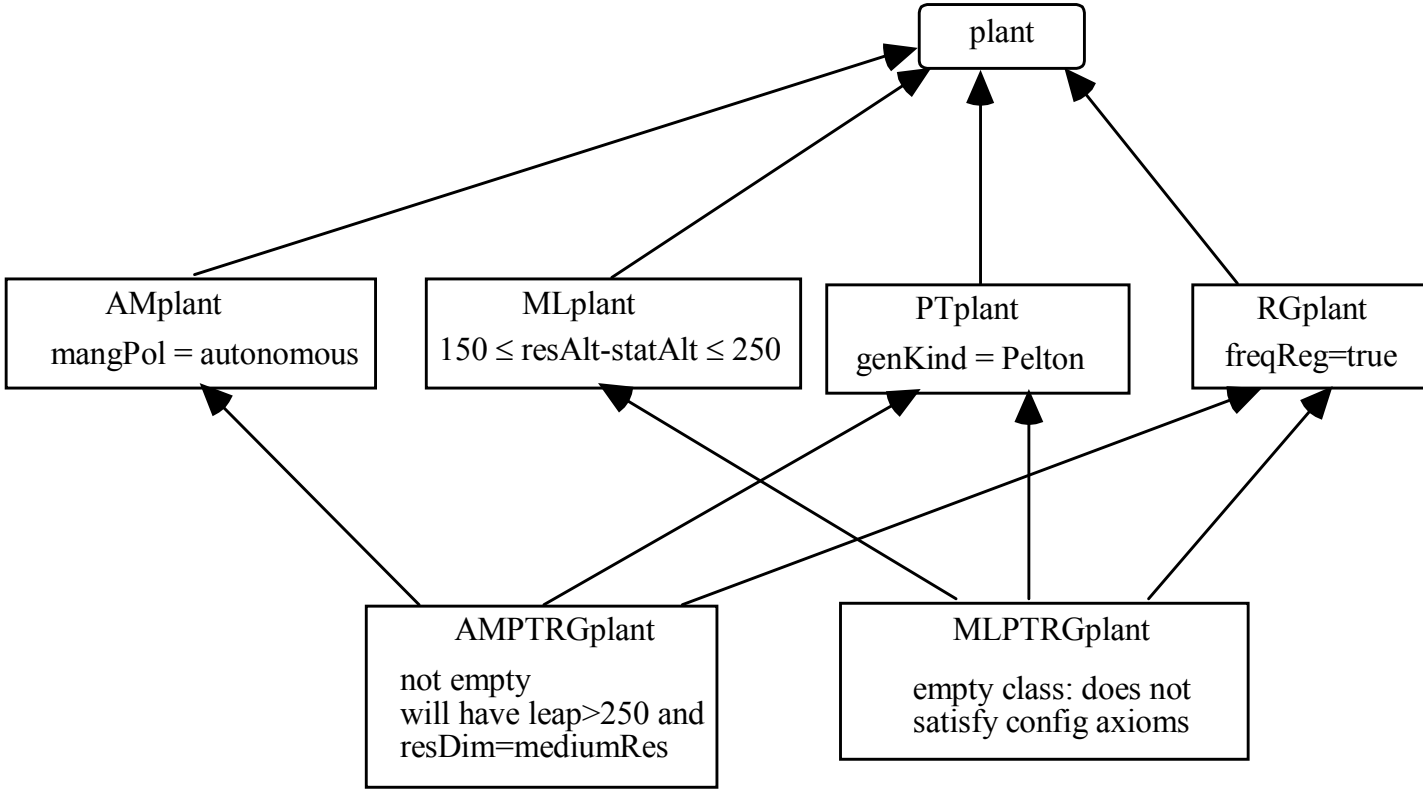
- The class declaration for the module *configuration* includes the axioms describing how the plant can be configured by varying the *type* and *dimension* of its components.

  - **type**
  - typeResDim = (smallRes, mediumRes, largeRes);
  - typeManPol = (autonomous, extern);
  - genKindType = (Francis, Pelton);
  - genSizeType = (small, medium);

- **class** clConfig
- **visible** ....
- **TI Items** ...
- **const** leap: natural; /*jump of the water from the reservoir to power station*/
- freqReg: boolean; /*true if the power units can regulate the frequency*/
- **axioms**
- /*the leap is defined as the difference in altitude between reservoir and power station */
- leap = resAltit - statAltit
- /*generating elements of the Francis kind must be used for small leaps (<150 m), Pelton ones for high leaps (> 250); for intermediate leaps both Francis or Pelton can be used but if the size of the reservoir (and hence its water delivery) is large Francis must be chosen */
- (leap <150 $\vee$ (leap ≤ 250 $\wedge$ resDim = largeRes)) $\rightarrow$ genKind = Francis
- leap > 250 $\rightarrow$ genKind = Pelton
- …
  /* Similar axioms state constraints among chosen generating units, reservoir size, managing policies, etc. For instance: "The size of the generating units is small if the leap is ≤ 250 and the reservoir is small or medium; it is medium otherwise; ..." */
- **end** clConfig.

- Notice that the properties asserted in the *clConfig* class may imply more constraints on the configuration activity than those explicitly stated. We could *derive*, for instance:
  - i) in a plant with a reservoir managed autonomously, Pelton power units and frequency regulation the leap must be high (>250) and the reservoir size must be medium;
  - ii) no plant can exist with a median leap ($150 \leq leap \leq 250$), Pelton power units and frequency regulation.

- Often, different categories of plants can be defined on the basis of some choice of the configuration parameters. We can describe such categories by pairing the configuration class definition with the inheritance mechanism:



Notice that a heir class is empty if it has properties that are incompatible with the constraints expressed in class *clConfig*. Thus, the properties of the configuration expressed in *clConfig* provide some discipline in determining the types and the dimensions of the components, by ruling out some unfeasible combinations.

- The selection of the allowed possibilities from those deriving from a pure combination of the immediate heir classes of class *plant* can be done by the specifier through informal reasoning on the axioms of *clConfig*, as in points i) and ii), or could be more effectively supported by semantic tools through the analysis of suitable formulas stating explicitly the feasibility of some combination of elements.
- In our example, class *clConfig* could include a *further set of axioms* expressing configurability of the plant, that is requiring that a certain combination of elements exists. E.g.: non-emptiness of class *AMPTRGplant*:
  - $\exists mp\ \exists gk\ \exists fr\ \ (mp=managPol \wedge gk=genKind \wedge fr=freqReg \wedge mp = autonomous \wedge gk = Pelton \wedge fr = true)$
- By analyzing the above axiom (possibly using an automatic tool), one can establish that the above formula is consistent with the other axioms and that when *mp*, *gk*, and *rg* have the indicated values, the leap will be over 250 meters, and the reservoir size will be medium: we obtained an automatic design from specs!

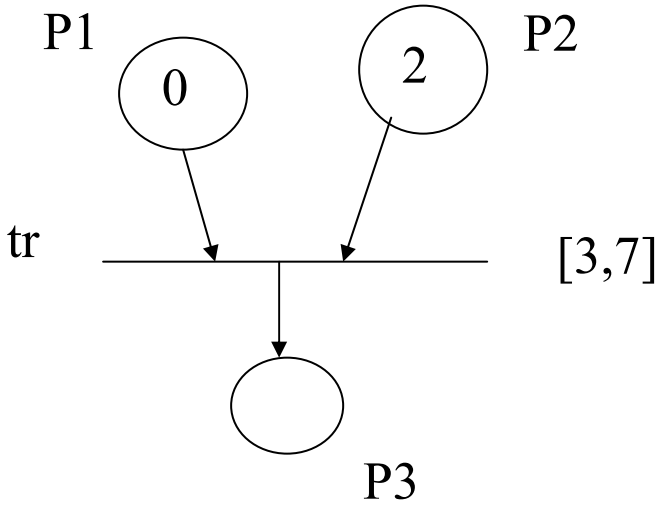
- The above method illustrates *static system configuration*;
  See the paper for *dynamic reconfiguration*.

Further case studies to be developed

- Completing the GRC and proving safety/utilities properties (exploiting TRIO tools)
- Developing the reservoir system
- …
- Investigating (more) thoroughly lower life cycle phases (design and verification)

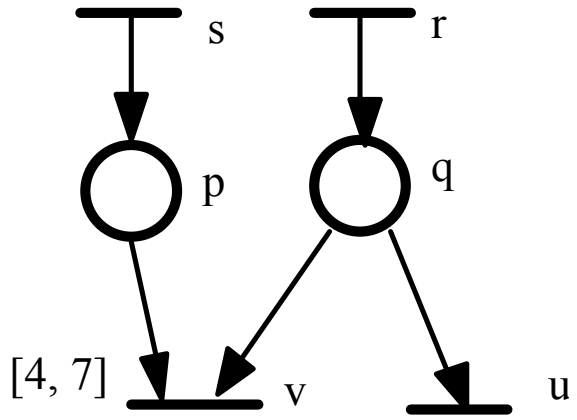
The case of **Timed Petri nets**

- Many versions thereof, more or less precisely formalized:
  - time attached to transitions (Merlin & Farber)
  - time attached to places, to arcs, …
  - combinations, reductions …
  - time + values associated with tokens and mutual dependencies
- Here we focus on Merlin & Farber (M&F)
- From the first, intuitive claims ….



If 0 and 2 are the times when tokens in P1 and P2 are produced, respectively, the tr fires nondeterministically in a time between 5 and 9

- … to a deeper semantic analysis:
  - Consider the net fragment in the figure below:



If transition *s* fires at time 0 and *r* fires at time 5, then *v* cannot fire before time 9. Furthermore, if within time 12 neither *v* nor *u* fired, then *v must* fire (unless *u* fires exactly at that time.)

The above rule is the usual interpretation of TPNs. It is called *strong time semantics* (STS) because it forces a transition to fire after its upperbound elapsed.

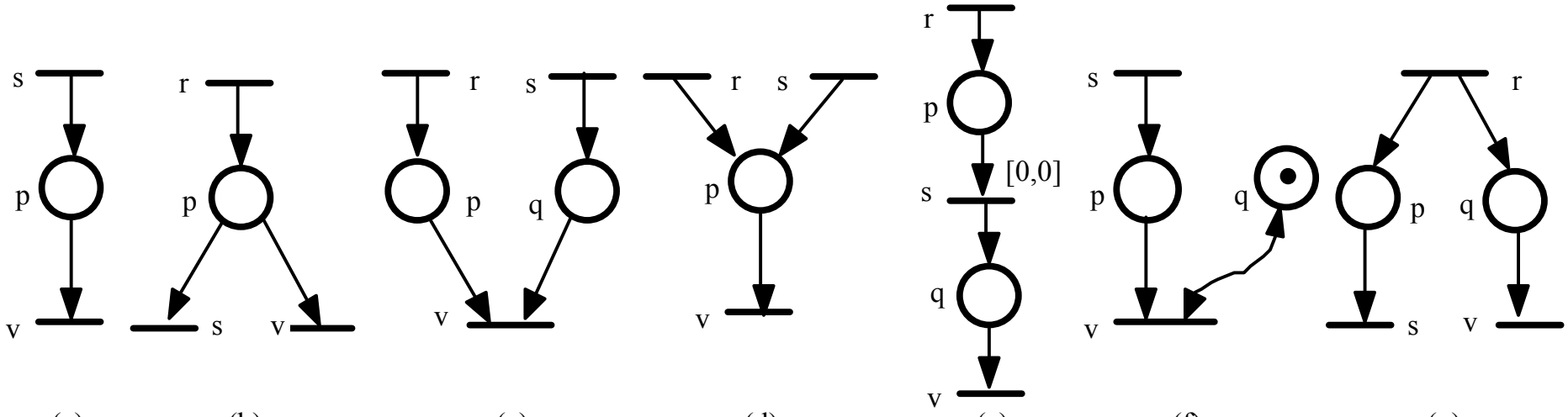
However, this semantics is perhaps inconsistent with traditional Petri nets semantics, where a transition is never forced to fire.

For this reason, a new semantics is proposed besides STS, that is called *weak time semantics* (WTS).

In WTS a transition is never forced to fire, but it is stated that, if it fires, it must fire in a time included in the interval $[m_v, M_v]$. For instance, suppose that in the net of the figure an upperbound of 3 is associated with *u*. Then, in STS *v* could never fire; instead, in WTS it could happen that *v* fires–after at least 4 time units since its enabling-because it would not be imposed that *u* fired before (in that case *u* could not fire anymore by using the token that enabled it for more than 3 time units.
Both semantics may be useful in different contexts.
Here, however, we will mainly focus on STS which is more widely adopted, leaving WTS for a few side remarks.

Notice that tokens are *uniquely* generated and consumed by transition firings. In particular any firing of a transition consumes one and only one distinct token from each place in its preset (we call this property IU, input unicity), and introduces one and only one token into each place of its postset; that token can contribute to no more than a single transition firing (we call this property OU, output unicity).

Let us now consider in some depth a few critical cases with the aid of the following figures



(a) (b) (c) (d) (e) (f) (g)

Simultaneous firings. Contemporary transition firings may occur in TPNs in two, deeply different, ways:

-*Simultaneous and concurrent firings*. This case is exemplified in Figure (g). Assume that both *s* and *v* have $m_v = M_v = 3$. Then, whenever *r* fires, *s* and *v* will *both* fire exactly 3 time units later. It is clear that, in general, they *could fire contemporarily* if and only if the intersection between their associated time intervals is not empty. A special case of simultaneous and concurrent firing will be examined shortly (meaning of the lowerbound).

-*Simultaneous but logically ordered firings (zero-time transitions)*. In general, it is not excluded that, for some transition, m_v–and possibly even M_v–are equal to 0. This abstraction is useful whenever the duration of the described event is negligible with respect to the other time constants of the net and it is widely adopted in the literature on TPNs.

Consider the fragment of Figure (e). In this case, whenever *r* fires, *s* fires *immediately* too, although the intuition associated with the usual interpretation of Petri nets would lead to say that *s* always fires *after r*. Thus, we are compelled to *clearly distinguish between logical ordering and temporal ordering*, although it is obvious that an event *s* that is the logical consequence of an event *r cannot precede r*, it is not implied that *s strictly* follows *r* in time.

Meaning of the lowerbound. Assume that in the net of Figure (a) $m_v = M_v = 3$. Then, if transition *s* fires in two instants that are 1 time unit apart, so does transition *v*, with the same delay. This semantics could not correspond to an intuitive interpretation of transition firing as executing some action that takes a given time. For instance, *v* could be interpreted as executing some algorithm on a datum that is modeled by the token produced in *p*. In such a case, if m_v gives the minimum time that is necessary to execute the algorithm, it is clearly wrong assuming that two consecutive computations may occur at a distance of 1 time unit. This interpretation of transition lowerbound is sometimes called the "recharge time".

There is no evidence that one of the two interpretations is preferable to the other one, but we chose the former because it is more general: the latter can be easily simulated by the former by simply adding a new place–with a single token assigned to it in the initial marking–that is both in the preset and in the postset of the considered transition.

Figure (f) shows how the fragment of Figure (a) should be transformed if one wants that two consecutive firings thereof never occur in a time interval that is shorter than m_v.

Remark: this semantics of the lowerbound may introduce another type of *simultaneous and concurrent firing of the same transition*. E.g., in the fragment of Figure (d), assume that *r* and *s* fire simultaneously and that $m_v = M_v = 3$.

Transitions with empty preset.

In non-timed Petri nets a transition *v* with no input places is conventionally considered as always enabled. This would allow an extension of the semantics to the timed case where, once the lowerbound m_v elapsed after the initial time, *v* may fire an unlimited number of times at any time.

This interpretation is clearly of no use in most cases since it reduces the meaning of m_v to that of an initial delay with no impact in the subsequent behavior of the net. Thus, we assume that such a transition cannot fire twice consecutively in less than m_v time units. If one wished to make this convention more explicit, it would suffice to add *v* a new place that is both input and output thereof in the same way as in Figure (f).

If in exceptional cases the other interpretation is preferred, this can be dealt with in several ways that are left to the reader.

Infinite upperbounds. If the upper bound time value M_v of a transition *v* is equal to ∞, then a behavior in which *v* never fires, and tokens entering the places of its preset stay there forever is admissible for the net. This is consistent with the traditional assumption that TPNs reduce to classical Petri nets when, for every *v*, $m_v = 0$ and $M_v = \infty$.

However, another possible intuitive meaning of the time interval $[m_v, \infty]$ is "the limit for $M_v \rightarrow \infty$ of $[m_v, M_v]$". Since in STS this means that *v must* fire in a time t, such that $m_v \leq t \leq M_v$ unless previously disabled, we could deduce that the meaning of $[0, \infty]$ is that *v must* fire in a time $t < \infty$, what is different from saying that the semantics of the net is the same as in non timed Petri nets, where an enabled transition could *never* fire.

In other words a *fairness* hypothesis could be assumed for the firing of enabled transitions (a transition cannot remain enabled for an infinite time without firing).

The new interpretation is more consistent with STS, whereas the traditional one is more consistent with WTS.

However, we stay as close as possible to the most widely known literature on TPNs, so we assume the original semantics for the interval $[m_v, \infty]$. We propose instead the use of the symbol '◆' to denote an unbounded finite value, i.e., the fact that the transition cannot remain enabled for an infinite time without firing.

> **Remark**: the symbol ◆ denotes "any finite value in an infinite domain", which is uncomputable. This should not prevent, however, from using it in *specifications*, by knowing a priori that it can only be *implied* by possible implementations (e.g., an implementation that guarantees the firing within a fixed time satisfies, but is not equivalent to, the requirement of guaranteeing the firing within ◆).

After having made precise the semantics of TPNs through informal analysis, we are now ready for its formalization (or, conversely, formalizing TPN semantics would force us to clarify the above issues: not everybody did so in the literature …).

This will be done shortly hereafter by exploiting the following logic-temporal language.

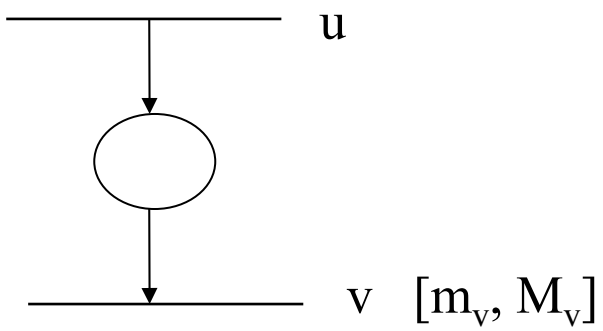
The essentials of the dual language approach illustrated through the case of Timed PN + TRIO

- We have often distinguished between operational and descriptive formalisms.
- We now know their distinguishing features, though we also realized that the borderline is not as sharp.
- It is not difficult to use a descriptive formalism with an operational style: it is sufficient to “plug in” the notion of state and its evolution: B is a typical example of a logic language tailored (perhaps too much) at describing abstract machines. (It ends up with a fairly intricate mixture of declarative and operational style: think e.g. of the meaning of ‘:=‘)
- It is a bit less natural to do the converse operation: abstract machines are used as “specifications” and lower level ones are their “implementation” the implementation relation is formalized through some notion of mapping (homomorphism). Algebraic approaches -Algebras, process algebras, CCS, LOTOS) are the typical realization of such an attitude. We do not explore them in this course.

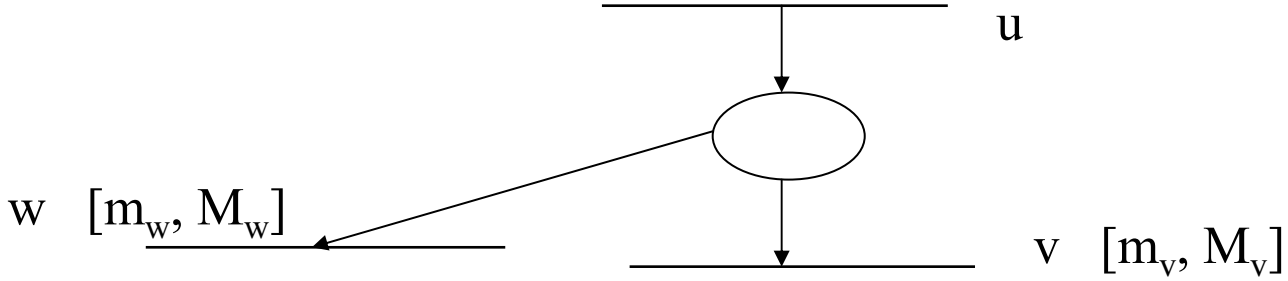
- Given that the two approaches are complementary, why not using them in combination?
- Actually, Hoare's approach already goes in this direction: we use a programming language (abstract machine) to implement algorithms and a logic language to express their properties (specification). The proof axioms and rules are nothing else than a formal definition of abstract machine semantics through the assertion language: it is also called *axiomatization* (when the assertion language is based on mathematical logics.)
- Another example is given by model checking: a logic language is used to express properties of a FSM; then, thanks to the decidability of most properties of FSM inductive algorithms are built to decide whether or not the FSM does indeed enjoy the desired properties.
- In general, such an approach
  [Describing a "design" through a suitable abstract machine; its properties through a descriptive language; providing a "bridge" between the twos through some kind of axiomatization, or translation schema]
  is called the dual language approach.
- Some of its pioneers were Pnueli and Ostroff (more or less finite state machines paired with more or less temporal logic.)

Here we will illustrate the dual language approach for TPNs and TRIO -in a highly simplified, yet still meaningful, context

- Main references:
  - Heitmeyer/Mandrioli (eds) Formal methods for real time computing, Wiley, chapter 6.
  - Felder M., Mandrioli D., Morzenti A., Proving Properties of Real-Time Systems through Logical Specifications and Petri Net Models, *IEEE Trans. on Software Engineering*, vol.20, no.2, Feb.1994, pp.127-141.
- We have seen that the -informal- semantics of TPNs is not as simple and straightforward as it appears at a first glance (several *errors* in the literature)
- Major sources of difficulties are:
  - The need of identifying tokens to compute firing times.
  - If contemporary firings are possible, saying that transition tr fires at time t is not enough: how many times does it fire -i.e., how many tokens does it produces?
  - The nondeterminism of the machine:



- Here, it is natural to state that, if u fires at time t, then v fires at t', with t+ $m_v \leq t' \leq t + M_v$.
- However, this cannot be generalized to say that, for



  - If u fires at time t, then v fires at t', with t+ $m_v \leq t' \leq t + M_v$.
  - If u fires at time t, then w fires at t', with t+ $m_w \leq t' \leq t + M_w$.
  - The "and" of the two claims generates a contradiction!

- To simplify, we adopt the following restrictions (for the general case see Felder et al.):
  - 1-bounded TPNs (à la Merlin & Farber)
  - The pre- and post- sets of every place have at most one transition each (marked graphs)
  - There are no loops of transitions with minimum firing time = 0 (non-Zenoness)
- By this way there is no doubt on the origin of a token in a given place;
- only a finite sequence of 0-time transition firings can occur contemporarily
- This strongly simplifies TPNs axiomatization, to be given next.

Thanks to the above simplifications we only need to consider PN fragments of the two types depicted in Figures (a) and (b) below:

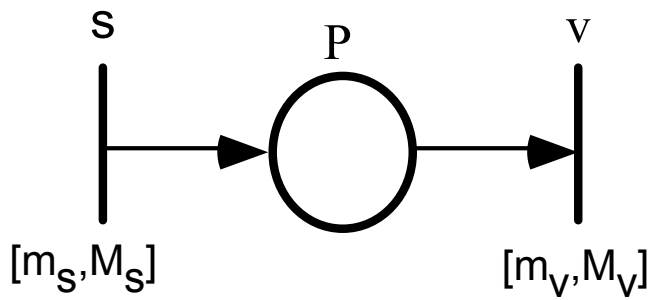


Figure (a)

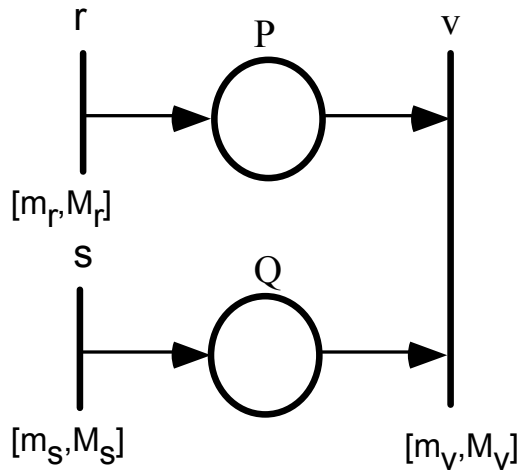
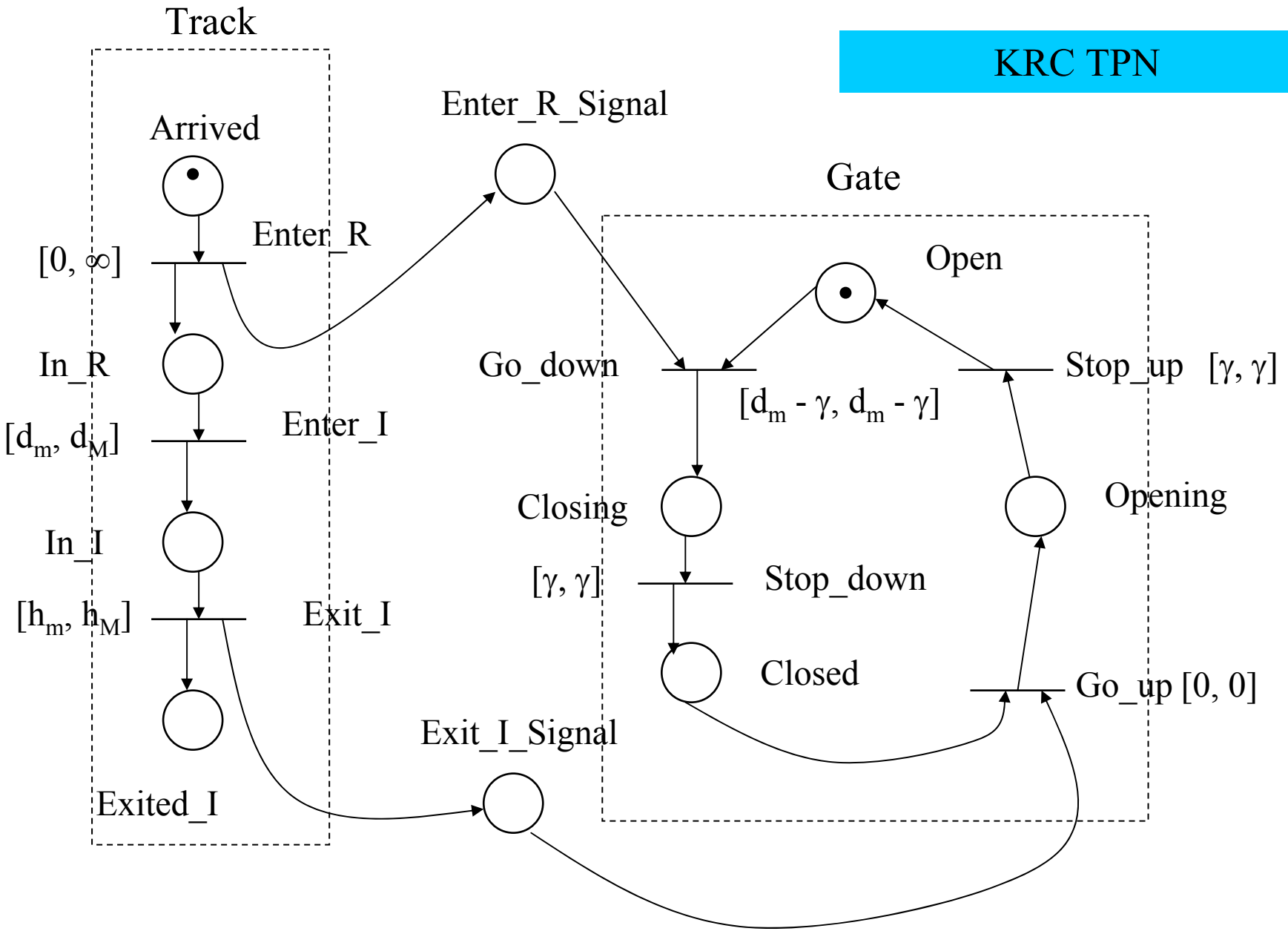


Figure (b)

They are enough to formalize, e.g., the KRC system:

KRC TPN


Track
Arrived
Enter_R_Signal
Gate
Enter_R
[0, ∞]
Open
In_R
Go_down
Stop_up [γ, γ]
[d_m - γ, d_m - γ]
Enter_I
[d_m, d_M]
Closing
Opening
In_I
[γ, γ]
Stop_down
Exit_I
[h_m, h_M]
Closed
Go_up [0, 0]
Exit_I_Signal
Exited_I

- A first axiom states that the predicate fire is istantaneous:
  (Ist): $\text{fire}(r) \rightarrow \text{UpToNow}(\neg \text{fire}(r))$
  (Another way of saying that it is an event)
- Another axiom relates the property of being a place marked to the firing of its input and output transitions: if s is the (unique) input transition of a place P and v is its (unique) output transition, then
  (Mar): $\text{marked}(P) \leftrightarrow (\text{Since}_{ie}(\neg\, \text{fire}(v), \text{fire}(s)) \vee (\text{fire}(s) \wedge \neg\, \text{fire}(v))$,
  i.e., place P is marked iff at some time in the past its input transition fired and since then its output transition did not fire yet; or just now the input transition fires and the output transition does not fire.
  Notice that *marked* is a predicate that can hold only in left-closed , right-open intervals. This implies that we assume the convention that a place is not marked if a token flows through it in zero time.

- Furthermore, the following axioms formalize the firing relations for the transitions belonging to the fragments of Figures (a) and (b):
  - (LB.(a)) $fire(v) \rightarrow Lasted_{ie}(marked(P), m_v)$
  - (UB.(a)) $Lasted_{ie}(marked(P), M_v) \rightarrow fire(v)$
  - (LB.(b)) $fire(v) \rightarrow Lasted_{ie}(marked(P) \wedge marked(Q), m_v)$
  - (UB.(b)) $Lasted_{ie}(marked(P) \wedge marked(Q), M_v) \rightarrow fire(v)$
  - LB.(a) formalizes the lowerbound for the firing of transition v in the fragment (a): v can fire (at the current instant) only if s fired at least m_v instants ago. UB.(a) states that if s fires, then v must fire within M_v time units. Similar explanations hold for the remaining axioms.

- The following Lemmas are straightforward consequences of the axioms. They illustrate how the above axioms formalize the behavior of timed PNs and will be useful in the proof of the properties of the KRC system.

- Lemma 1: With reference to Figure (a):
  - (A) $\text{marked}(P) \rightarrow \exists d\ \text{past}(\text{fire}(s),d) \wedge d \leq M_v$
  - (B) $\text{fire}(v) \rightarrow \exists d\ \text{past}(\text{fire}(s),d) \wedge m_v \leq d \leq M_v$
  - (C) $\text{fire}(s) \rightarrow \exists d\ \text{futr}(\text{fire}(v),d) \wedge m_v \leq d \leq M_v$

- Lemma 2: With reference to Figure (b):
  - (A) $\text{fire}(s) \wedge \text{UpToNow}(\text{marked}(P)) \rightarrow \exists d\ \text{futr}(\text{fire}(v),d) \wedge m_v \leq d \leq M_v$
    If s fires at the current time and P is already marked, then v will fire not earlier than m_v and not later than M_v. A symmetric lemma obviously hold w.r.t. place Q and transition r.
  - (B) $\text{fire}(v) \rightarrow (\text{SomeP}_i(\text{fire}(r)) \wedge \text{SomeP}_i(\text{fire}(s)))$

- We are now ready for the safety proof of the KRC system:

- Let us first formalize the safety property as a TRIO formula with reference to the KRC TPN figure: quite intuitively, this is the following:

  (saf) SomP((marked(Arrived) $\wedge$ marked(Open)) $\rightarrow$

  (marked(In_I) $\rightarrow$ marked(Closed))

  i.e., if initially the train is out of R and the gate is open, then, whenever the train will be in region I, the gate will be closed.
- Our job is therefore to prove that (saf) is a theorem in the above axiom system.
- The proof of (saf) will be derived by using the previous lemmas and the following one which states, without proof, an obvious property of the net of the KRC TPN figure(Actually, the hypothesis marked(Arrived) will not be exploited but it has been stated to stay close to physical intuition):

- Lemma 3: Every transition of the net of Figure KRC TPN can fire at most once.

Safety proof of the KRC TPN

- Assume that at the current instant marked(In_I) holds. Then,
- marked(In_I) $\rightarrow$ (by Lemma 1A)
- $\exists t$ Past(fire(Enter_I),t) $\wedge$ $t \leq h_M$, $\rightarrow$ (by Lemma 1B)
- $\exists t$ Past(fire(Enter_I) $\wedge$ $\exists p$ Past(fire(Enter_R), p), t) $\wedge$ $d_m \leq p \leq d_M \wedge t \leq h_M \rightarrow$ (by properties of Past operator)
- $\exists q$ Past(fire(Enter_R) , q) $\wedge$ $d_m \leq q \leq h_M + d_M$.
- Since, by Lemma 2.B, Go_Down could not fire before a firing of Enter_R, then when Enter_R fired q instants in the past, the place Open was still marked (axioms for initial marking have been omitted). By Lemma 2A, this implies:
- Past(fire(Go_Down),q- (d_m - γ))), that is, by Lemma 1C:
- (*) Past(fire(Stop_Down), q-d_m).
- Notice that q-$d_m \geq 0$, that is Stop_Down fired in the past.
- By definition, marked(In_I) $\leftrightarrow$ (fire(Enter_I) $\wedge$ $\neg$ fire(Exit_I)) $\vee$ $Since_{ie}$($\neg$ fire(Exit_I), fire(Enter_I)). Since Past(fire(Enter_I), t), with $t \leq h_M$,, then $SomF_i$(fire(Exit_I)), and by Lemma 3 AlwP($\neg$ fire(Exit_I)) holds. Thus, by Lemma 1.B:
- (**) AlwP($\neg$ fire(Go_Up))
- By (*) and (**) it follows that, in case q - d_m = 0, fire(Stop_Down) holds, in case q - d_m > 0 $Since_{ie}$($\neg$ fire(Go_Up), fire(Stop_Down)) holds, that is marked(Closed) holds.

Comments

- The (highly simplified and a little dirty …) proof is similar to the “pure TRIO” proof, but closer to operational intuition (?)

- (Advanced comment)
  - Our proof system is well suited to prove properties such as “whatever it will happen, such a requirement -e-g-. invariant- will be *guaranteed*”.
  - In the case of nondeterministic systems this properties -and related proofs- sharply differ from properties such as “it *may happen* that the system behaves in such a way to achieve a given -positive or negative- result”.
  - It is hard to prove this type of properties with our proof system
  - Branching logic (not pursued here) has been proposed to overcome such a difficulty

A deeper insight into a critical issue: the case of 0-time transitions

- *** see:
- PisaSeminar-03

The (formal) refinement issue

A short historical perspective of refinement

- The term *stepwise refinement* was proposed in the early 70s in the context of structured programming
- In general, it means moving gradually from early formulations -of anything- to more and more precise ones by adding details -pieces of information.
- Its generality encompasses several interpretations -not all of them being equivalent:
  - Specifications can be refined towards implementation
  - Specifications -or any other sort of documentation- can be refined just to make them more precise and complete
- Often the term is used rather ambiguously
- In the context of formal methods it has received precise definitions (e.g. in B method) but often at the expenses of flexibility and applicability
- Next, ***.

Case study: **B**: a fairly complete Formal *method*

- Formalisms can be used for:
  - specifying (defining)
  - implementing (designing, coding)
  - verifying
    - correctness proofs,
    - model checking,
    - but also support to testing, e.g. producing test cases (semi)automatically from specs and/or code)
- As such they are not *methods*
- A method teaches "how to do things":
  - stepwise refinement
  - structured analysis
  - several OO A/Ds, …
- A method can be
  - informal
  - fully formal (every action is based on one or more formalisms)
  - partially formalized
  - open to formalization
  - pure or mixed/integrated
  - ….

- B is a fairly complete formal *method*
- It is based on *predicate transformers* (Extensions of Hoare/Dijkstra/and others approaches to "program specification/construction/proof"
- It is based on stepwise refinement(we will need to go fairly deep into the refinement issue)
- It is based on correctness proofs as a major approach to verification
- In some sense it can be seen as one *method* rooted in Hoare's et al. Formal *approaches*.
- It has some major drawbacks (in my opinion mainly due to some lack of flexibility). We will critically evaluate it.
- However it is one of the most noticeable formal methods from the point of view of industrial "success". Such a success may be explained by:
  - A well established state and documentation based on a long-standing experience
  - The adoption of natural evolution of well studied techniques rather than reinventing "the wheel"
  - The availability of a well engineered (not free!) toolkit
  - Political support
  - (Not by chance "successful" FMs are B in France, Z (and to a lesser extent VDM) in UK and Danmark)

1. The B "language"

- A unifying notation rooted in predicate logic (a` la Hoare/Dijkstra) but extended to cover typical programming language features.
- The B language claims assertions about the state of an abstract machine
- The state is an assignment of values to variables
- The predicate transformer:
  [S]R :
  the *weakest precondition* that must by satisfied before execution of *operation* S in order to guarantee that *after* its execution, the postcondition R holds
- The very basic operation: the substitution:
- xx := E {xx's value in the post condition is replaced by E's value}
  formally: let R[E/xx] denote the result of substituting in R each free occurrence of xx by E. Then
  [xx := E]R <==> R[E/xx]
- Notice the similarity with Hoare's approach

Other basic substitutions
(elementary assignments of values to variables)

- Skip [skip]R <==> R {skip has no effect}
- Choice from set xx : ∈ S
  [xx : ∈ S]R <==> [@xx'. xx' ∈ S ==> xx := xx']R
  To be read as:
  xx receives any value in S. Thus, [xx : ∈ S]R holds if and only if for any xx' in [S] R holds by substituting xx by xx'.
- Choice by predicate xx : P
  [xx : P]R <==> [@xx'. [xx := xx'] P ==> xx := xx']R
  {xx receives any value satisfying P}
- Multiple (parallel) substitution)
  xx, yy := E, F
  [xx, yy := E, F] R <==> R[E,F/xx,yy]

Basic compositions of substitutions

- Sequential composition: G; H
  [G; H] R <==> [G][H] R
- Preconditioned P|G
  [P|G]R <==> P ∧ [G] R
- Guarded P ===> G
  [P ==>G]R <==> P ==> [G] R
- Alternate G [] H
  [G[]H]R <==> [G]R ∧ [H] R
  {the precondition guarantees that whether I apply G or H, I obtain R}
- Unbounded choice @zz.(G)
  [@zz.(G)]R <==> ∀ zz.([G] R)
  {for any choice of zz [G] R is the weakest precondition of substitution G to obtain R}

- We now move a first step towards a more "programming oriented" notation. The following extended notation can be seen as a set of abbreviations for the corresponding compositions
- BEGIN G END — G
- IF P THEN G ELSE H END — (P ==> G) [] (¬ P ==> H)
- IF P THEN G END — IF P THEN G ELSE skip END
- CHOICE G OR H END — G [] H
- SELECT P THEN G WHEN ... WHEN Q THEN H END — P ==> G [] ... Q ==> H
- VAR z IN G END — @ z . (G)
- ANY z WHERE P THEN G END — @ z . (P ==> G)
- LET x BE x = E IN G END — @ x . (x = E ==> G), where x\E
  {x does not occur free in E}
  {x denotes any value of expression E satisfying G}

- Better understanding will be gained through the first following examples

The while-loop substitution
(definitely towards operational programming ...)

- Let
  - P be a "controlling predicate"
  - S: a substitution
  - V: an arithmetic *variant* expression, which is a function of the machine state
  - I: an *invariant* predicate on the machine state
- Then the semantics of the **while-loop** substitution
  WHILE P DO S VARIANT V INVARIANT I END
  is defined by the following inference rule

$$\frac{I \wedge P \Rightarrow [S]I,\ I \Rightarrow V \in N,\ I \wedge P \Rightarrow [n := V][S](V < n),\ \neg P \wedge I \Rightarrow R}{I \Rightarrow [WHILE\ P\ DO\ S\ VARIANT\ V\ INVARIANT\ I\ END]R}$$

Notice the similarity with the traditional **while-loop** proof rule with the inclusion of the termination condition

B type system (hints)

- B is strongly typed but not through the traditional type/variable declaration mechanism of classical programming languages.
  Rather set theory is exploited to assign a *type* (set) to any variable
  The correct use of a variable according to its type must be proved with the same logic deduction mechanisms as other specifications
- Type assignment is performed through the CONSTRAINTS section of machine definition.
  Such constraints are of the form
  $x \in S$, $x \subset S$, $x \subseteq S$, $x = E$, where x is a variable symbol, S a set symbol, E an expression symbol and $x \setminus S$, $x \setminus E$, i.e., x does not occur free in S or E.

A first example of B *specification*

- **MACHINE** SquareRoot
  OPERATIONS
  - sqrt ← SquareRoot (xx) $=_{def}$ {is defined as}
    - **PRE** $xx \in N$
      THEN
      - **ANY** yy
      - **WHERE** $yy \in N \wedge square(yy) \leq xx \wedge xx < square(yy+1)$
        THEN sqrt := yy
        END
    - **END**
  - **DEFINITIONS**
    - $square(x) =_{def}$ x*x
  - **END**

A few explanations/comments

- The name of the machine is SquareRoot
- The DEFINITIONS sections defines auxiliary operations as "macros"
- For the sake of clarity machine variable identifiers are long (2 characters);
  1-character identifiers are reserved for dummy variables -such as in the definition of auxiliary operations- and represent arbitrary expressions
  Notice the similarity with the previous convention (overscored identifiers) in Hoare's formalism
- The machine has a single operation SquareRoot. The operation has a single parameter xx and returns the result sqrt (to the left of ←)
- Type assignment arises from set belonging
- := has a semantics *similar* to the traditional assignment but is *not -yet!-* assignment. Think in terms of *substitution*!
- In top level machine specification **WHILE-LOOP** and sequential composition of substitution are not allowed. This emphasizes that we are talking of specifications not of implementation, though we will use the same language for both.

A second example

- **MACHINE** UniqueID (maxids)
  CONSTRAINTS maxids ∈ N
  SETS IDS
  PROPERTIES card(IDS) = maxids
  VARIABLES usedIDS
  INVARIANT usedIDS ⊆IDS
  INITIALIZATION usedIDS :={}
  OPERATIONS
  - newid ←allocID $=_{def}$
    - **PRE** userIDS ≠ IDS
      THEN
      - **ANY** nid
      - **WHERE** nid ∈ IDS - usedIDS)
        THEN newid := nid || usedIDS :=usedIDS ∪ {nid}
        END
    - **END**
  - nids ← FreeIDS $=_{def}$
    - **BEGIN**
      - nids :=card(IDS - usedIDS)
    - **END**

Further explanations/comments

- The machine has a parameter: the maximum number of identifiers
- The CONSTRAINTS section specifies the type of the parameter
- The "abstract set" IDS specifies the "universe" of identifiers. At this time it is not -yet- defined. It will have to be instantiated during refinement (e.g. integers, strings, ...)
- PROPERTIES section specifies constraints on the abstract set IDS. card is a built-in function with the obvious meaning
- The machine state consists of the single variable usedIDS. The INVARIANT section specifies that userIDS must be a subset of the universe IDS. usedIDS denotes the identifiers already allocated. Hence the natural initialization.
- The machine has two operations, allocID which returns a new (not previously used) identifier; FreeIDS which returns the number of still available identifiers.
- Notice that both specifications use the nondeterministic operator **ANY**. However. SquareRoot is such that the formula specifying yy defines a unique value. Instead, allocID is allowed to return several different values satisfying the specification; thus, its result is truly nondeterministic. This allows for different possible implementations that satisfy the (top-level) specification.

Some initial summarizing remarks

- Despite the "programming flavor" of the B notation we have built some true specifications, i.e., *definitions*, as opposed to *implementations*.
- The distinction between definition and implementation, however, is more a matter of style and relative implications than a formal "definition": a Turing machine "computes" but also "defines" a function; a logic formula -say, specifying the sorting of a list- may look as a definition but may become a true implementation if the logic language -e.g., PROLOG- is interpreted by a machine.
- Next, we go into the core of the B method:
  coming to implementation through refinement.

A short historical perspective of refinement

- The term *stepwise refinement* was proposed in the early 70s in the context of structured programming
- In general, it means moving gradually from early formulations -of anything- to more and more precise ones by adding details -pieces of information.
- Its generality encompasses several interpretations -not all of them being equivalent:
  - Specifications can be refined towards implementation
  - Specifications -or any other sort of documentation- can be refined just to make them more precise and complete
- Often the term is used rather ambiguously
- In the context of formal methods it has received precise definitions (such as B's one) but often at the expenses of flexibility and applicability
- Next, we examine the issue of refinement in B; later we will critically evaluate it.

Refinement in B

- Intuitively, P is refined by Q if Q may satisfactorily *substitute* P. Satisfactorily does not mean that P and Q are equivalent. Rather, in some sense, that Q "implies" P.
- In B it is convenient to distinguish two types of refinement, though there is no formal definition of such a difference:
  - procedural or algorithmic refinement:
    only the "computational" part of the specification is refined but variables are not changed
  - data refinement:
    the machine state too is changed

A first example of procedural refinement: the SquareRoot

- **REFINEMENT** SquareRootR
  REFINES SquareRoot
  OPERATIONS
  - sqrt ← SquareRoot (xx) $=_{def}$
    - **ANY** yy, zz
      WHERE yy ∈ N ∧ zz ∈ N ∧
      sqinv (xx, yy, zz) ∧ zz = yy + 1
      THEN sqrt := yy
      END
  - **DEFINITIONS**
    - square(x) $=_{def}$ x*x;
      sqinv(x,y,z) $=_{def}$ y < z ∧ square(y) ≤ x ∧ x < square (z)
  - **END**

Usual explanations/comments

- Since the machine has no state (xx is a parameter, yy and zz are *values* used to *denote* a result) the refinement is pure procedural refinement
- It is "clear" -*but we will have to prove it*- that
  sqinv (xx, yy, zz) $\wedge$ zz = yy + 1 implies
  square(yy) $\leq$ xx $\wedge$ xx < square (yy+1)
- sqinv in some sense "suggests" an iterative way to get the desired result. In fact, not by chance its identifier "sounds" close to the term *invariant*.
- However, so far, this definition is still in a declarative style, though it is more suggesting an implementation algorithm than the top-level one.
- Let us clearly distinguish between
  - Formal facts:
    - Still while loops and sequential compositions are not used
    - sqinv is just a predicate
    - "assignment" is used only to define the returned result
  - Feelings, flavors, and intuitions

Data refinement example: the unique identifier machine

- **REFINEMENT** UniqueIDR
  REFINES UniqueID
  PROPERTIES IDS = 1 .. maxids
  VARIABLES lastID
  INVARIANT lastID $\in$ N $\wedge$ usedIDS = 1 .. lastID
  INITIALIZATION lastID := 0
  OPERATIONS
  - newid $\leftarrow$allocID $=_{def}$
    - **BEGIN** newID := lastID + 1|| lastID := lastID + 1
      {parallel substitution!}
      END;
  - nids $\leftarrow$ FreeIDS $=_{def}$
    - **BEGIN**
      - nids := maxids - lastID
    - **END**

Explanations/comments

- This time we have a "decrease of abstraction" both in data and in operations definition
  - The abstract set IDS is replaced by the "concrete" set of natural numbers 1.. maxids
  - We "resolve" nondeterminism by allocating deterministically new identifiers in a sequential way
- As a consequence:
  - The abstract set usedIDS is replaced by 1 .. lastID
  - The state variable is just the number lastID
- The invariant not only states an invariant property of the refining machine, but also "implies" the higher level invariant
  (this too will be a matter of formal proof)

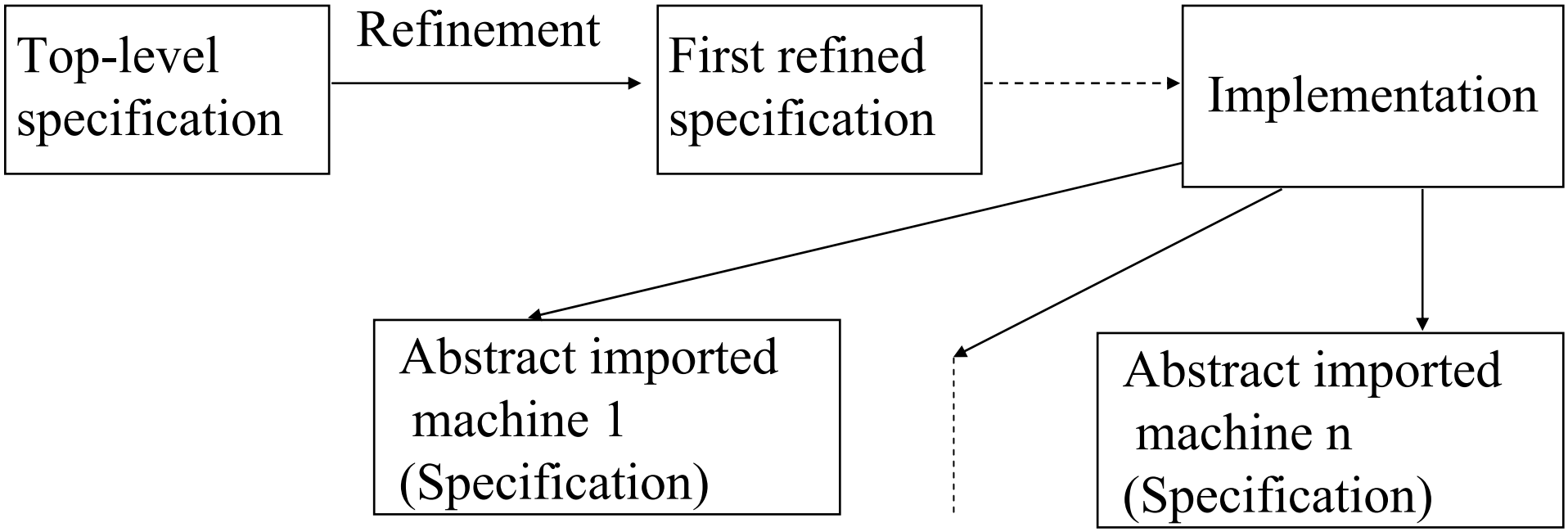
Refinement and nondeterminism

- If a specification is deterministic then its refinement must have the same "semantics".
  E.g., if an operation defines a function, as in the case of the square root, then its refinement must define the same function.
- However, if the specification is nondeterministic, then asking for semantic *equivalence* is too much:
  In fact, the implementation of allocID has not the same meaning as the top-level specification but simply "implies" it according to some intuition.
- This intuition will be made formal and precise through suitable *proof obligations*.

Implementation

- Implementation is a special refinement step: the *last* one. It must satisfy the following strong constraints:
  - The implementation machine has no state of its own: it "uses" the state of *imported machines*;
  - To implement the operations the implementation machine must import other (specification) machines;
  - Any parameter of imported machines must be instantiated in the IMPORTS clause;
  - The operations of the implementation machine cannot directly access variables (which belong to imported machines): they can only do that by invoking operations of the imported machines
  - Implementation machines cannot use"abstract substitutions" such as nondeterministic choice and parallel composition;
  - On the contrary they can use sequential composition and the **while-loop**.

The typical B refinement process



Comments

- The notion of implementation follows the principles of encapsulation, information hiding, ADT, ...:e.g. depending only on the interface (specification) of the imported "modules".
- Despite the fact that the language is unique, some features are typical of specification (and excluded in the implementation) and conversely:
  ======>
  in some sense the language too is refined through the several steps.

First implementation example: the unique identifier machine

- **IMPLEMENTATION** UniqueIDRI
  REFINES UniqueIDR
  IMPORTS ID_Nvar 1(maxids)
  INVARIANT lastID = ID_Nvar
  OPERATIONS
  - newid ← allocID $=_{def}$
    - **BEGIN**
      - ID_INC_NVAR;
        newID ←ID_VAL_NVAR
    - **END;**
  - nids ← FreeIDS $=_{def}$
    - **VAR** ll
      IN
      - ll ← ID_VAL_NVAR;
        nids := maxids -ll
    - **END**
  - **END**

Explanations/Remarks

- The imported machine is a renamed instance of a natural number variable machine. Its specification is omitted here as quite natural.
- (The machine Rename_Nvar is available in the standard library of B toolkit. The prefix ID_ identifies this instance of the renamed machine)
- The parameter of the Nvar machine sets the upperbound for the values of the variable
- ID_INC_NVAR increments the value of ID_Nvar
- ID_VAL_NVAR reads the value of ID_Nvar
- Operations may have local temporary variables (not state variables)
- As usual we will have to prove the correctness of the refinement implementation through suitable proof obligations

Second implementation example: the square root machine

- **IMPLEMENTATION** SquareRootRI
  REFINES SquareRootR
  IMPORTS SquareRootUtils
  OPERATIONS
  - sqrt ← SquareRoot (xx) $=_{def}$
    - **VAR** yy, zz
      IN yy := 0; zz := (xx+1)/2 + 1;
      - **WHILE** yy + 1 /= zz
        DO yy, zz ← ChooseNewApprox (xx, yy, zz)
        INVARIANT yy ∈ N ∧ zz ∈ N ∧ sqinv(xx, yy, zz)
        VARIANT zz - yy
        END;
      - sqrt := yy
  - **DEFINITIONS**
    - square(x) $=_{def}$ x*x;
      sqinv(x,y,z) $=_{def}$ y < z ∧ square(y) <= x ∧ x < square(z)
  - **END**

This time the implementation strongly depends on the imported items.
Thus, we define and implement the imported machine too.

- **MACHINE** SquareRootUtils
  OPERATIONS
  - yy, zz ← ChooseNewApprox (xx, yy0, zz0) $=_{def}$
    - **PRE** xx ∈ N ∧ yy0 ∈ N ∧ zz0 ∈ N ∧
      sqinv (xx, yy0, zz0) ∧ yy0 + 1 < zz0 **THEN**
    - **ANY** ww
      WHERE ww ∈ N ∧ yy0 < ww ∧ ww < zz0 **THEN**
      - **SELECT** sqinv (xx, ww, zz0) **THEN** yy, zz := ww, zz0
        WHEN sqinv (xx,yy0, ww) **THEN** yy, zz := yy0, ww
        END
    - **END**
  - **END**
  - **DEFINITIONS**
    - square(x) $=_{def}$ x*x;
      sqinv(x,y,z) $=_{def}$ y < z ∧ square(y) <= x ∧ x < square(z)
  - **END**

- **REFINEMENT (IMPLEMENTATION)** SquareRootUtilsR
  REFINES SquareRootUtilsR
  OPERATIONS
  - yy, zz ← ChooseNewApprox (xx, yy0, zz0) $=_{def}$
    - **VAR** ww **IN**
      - ww := (yy0 + zz0)/2;
        IF ww * ww <= xx
        THEN yy := ww; zz := zz0
        ELSE yy := yy0; zz := ww
        END
    - **END**
  - **END**

- With a subtle remark we could "implement" this last refinement by replacing ww * ww <= xx with ww <= xx/ww and relying on the PRE ww /= 0: by this way we would avoid the risk of overflow in the operation
  ww*ww. Here we neglect these subtleties and we consider the present one as the final implementation.

Comments

- Of course, this refinement process is overdetailed if compared with the complexity of the example (typical "toy example" criticism)
- However, it serves well several illustration purposes
- Also, it shows pretty well the philosophy of the method which emphasizes moving smoothly from specification towards implementation and uses a notation which parallels the process (as opposed to the so-called "dual language approach".)
- It is also emphasized the usefulness of concentrating on single -or few- issues at any design step.
- All this is further emphasized with the distinguishing feature of the B method, i.e. pairing design with verification through correctness proofs: every step must be accompanied -at least in principle- by a mathematical proof (well supported by B's toolkit.)

Proof obligations in B (hints)

•Proof obligation: what must be proved to guarantee that the given specification is satisfied.

•In the case of Hoare's system

$\{I \wedge c\}\ S\ \{I\}$ is a proof obligation to guarantee $\{I\}$ **while** c **do** $S\ \{I \wedge \neg c\}$

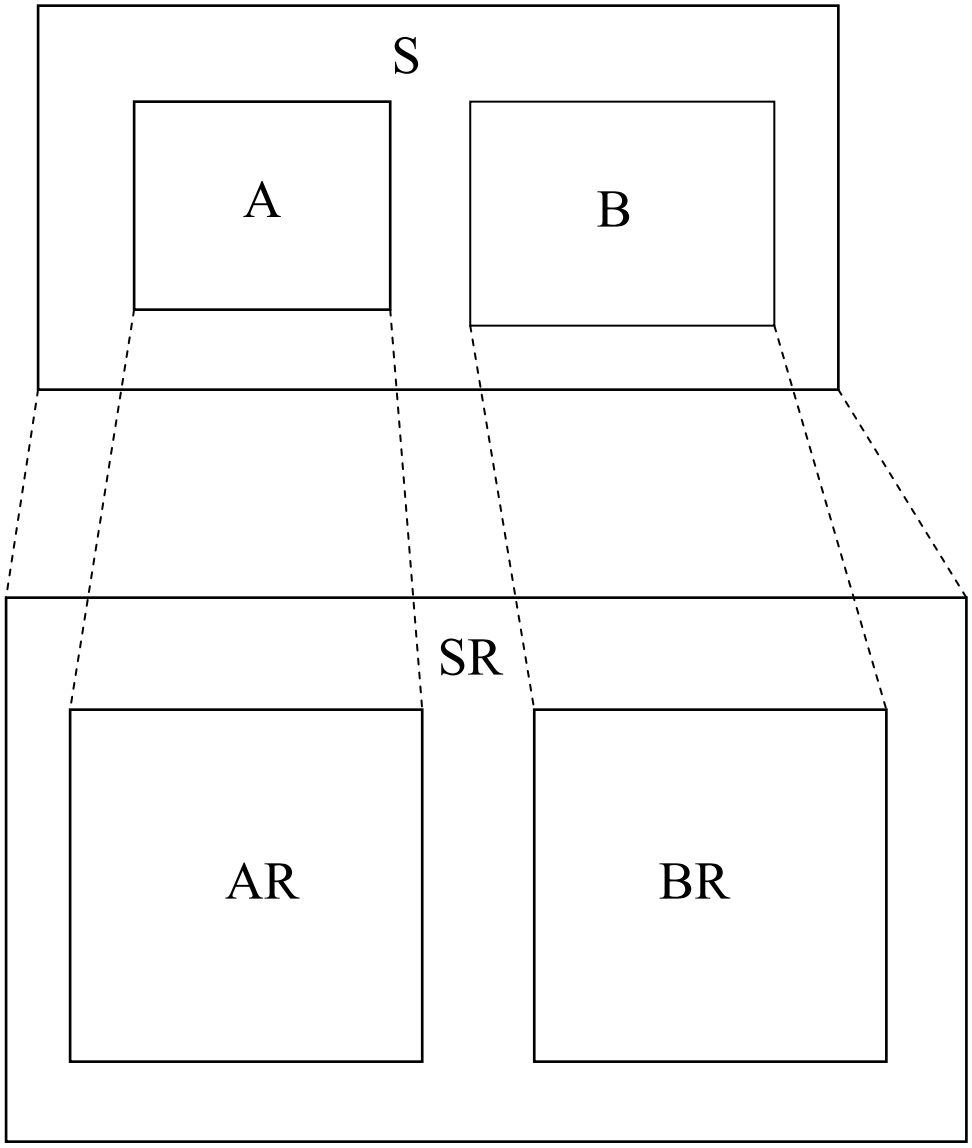
In the case of structured methods such as B, *proofs are structured as well as design*.

In a fairly general perspective, we may distinguish between:

Horizontal proofs

Vertical proofs

: ----->



Horizontal proofs: properties of components A, B imply global properties of system S

Vertical proofs: properties of refining components AR, BR, and refining system SR imply properties of refined components A, B, and system S, respectively

- In B the distinction between the two types of properties is not as sharp (for a deeper clarification see:
  Kolano, P.Z. “Tools and Techniques for the Design and Systematic Analysis of Real-Time Systems”. Ph.D. Thesis, University of California, Santa Barbara, Dec. 1999.)
- However, it is useful to refer to such a distinction when considering any type of proof obligation in structured methods that involve both structuring a complex system into several components and refining them through several layers.
- We will be happy enough to scratch the surface of B’s proof system (without the toolkit …)
  B’s toolkit *automatically* generates all proof obligations (they are quite many!); then it *helps* proving them: the trivial ones are automatically verified, for the other ones human interaction is needed.
  In case of success proofs are completely certified (formalized)

A typical case of horizontal proof obligations

- MACHINE
  - M(X, x)
- CONSTRAINTS
  - C
- SETS
  - S; T = {a,b}
- CONSTANTS
  - c
- PROPERTIES
  - P
- INVARIANT
  - I
- ASSERTIONS
  - J
- INITIALIZATION
  - U
- OPERATIONS
  - $u \leftarrow O(W) =_{def}$ PRE Q THEN V END;
- END

- Let A be an abbreviation for $X \in P(INT)$;
- B for $S \in P(INT) \wedge T \in P(INT) \wedge T = \{a,b\} \wedge a \neq b$
  {A and B state type properties for used sets}
- Then the following are the proof obligations for M:
- $A \wedge B \wedge C \wedge P \Rightarrow [U] I$
  {The initialization (+ the "basic assumptions) guarantees the invariant}
- $A \wedge B \wedge C \wedge P \wedge I \Rightarrow J$
  {actually J is a "corollary" of I; inessential here)}
- $A \wedge B \wedge C \wedge P \wedge I \wedge J \wedge Q \Rightarrow [V] I$
  {operation(s) O, which applies substitution V under precondition Q, preserves the invariant}

Another typical horizontal proof obligation (for refining/implementing machines) is the while-loop obligation:

$$\frac{I \wedge P \Rightarrow [S]I, I \Rightarrow V \in N, I \wedge P \Rightarrow [n := V][S](V < n), \neg P \wedge I \Rightarrow R}{I \Rightarrow [WHILE\ P\ DO\ S\ VARIANT\ V\ INVARIANT\ I\ END]R}$$

We already noticed that it generalizes Hoare's proof rule -obligation by including termination

An example of "vertical" proof obligation

- Let specification
- MACHINE M
- VARIABLES V
- INVARIANT I
- OPERATIONS
  - $R \leftarrow Op =_{def}$
    - PRE P
      THEN S
      END
- END
- be refined as
- MACHINE M_R
- REFINES M
- VARIABLES V_R
- INVARIANT I_R
- OPERATIONS
  - $R \leftarrow Op =_{def}$
    - PRE P_R
      THEN S_R
      END
- END

- Then, we must prove:
- $I \wedge I_R \wedge P \Rightarrow P_R$
  {The upper level (UL) precondition and the two invariants imply the lower level (LL) precondition}
- $I \wedge I_R \wedge P \Rightarrow [S'_R]([S](I_R \wedge R'=R))$
  where $S'_R = [R := R'] S_R$.
  {I.e., the refining operation must produce the same result R as the refined one, under the same assumptions.}
- Notice that the above obligation is a simplified version of a more general one, that applies only when Op is deterministic. Otherwise, we should prove:
- $I \wedge I_R \wedge P \Rightarrow [S'_R](\neg[S](\neg (I_R \wedge R'=R)))$
  {the refining operation does not contradict the refined one}
- As a matter of style this -and other- proof obligations might be criticized as they "mix UL and LL properties" …..

In the “official” B theory proof obligations are classified into:

- *Context*: proofs that sets satisfying constraints and properties exist
- *Maintenance of invariant*: proof that the invariant is established by initialization and kept by the operations
- *Satisfaction of refinement relation*: proof that refinement and implementation machines satisfy the refinement constraints
- *Precondition*: proof that preconditions for any invoked machine are satisfied

Now, let us state the following abbreviations:

- *Cst(mch)*: the CONSTRAINTS clause of machine *mch*
- *ctx (context)*: the PROPERTIES clause of machine *mch*
- *inv(mch)*: the INVARIANT clause of machine *mch*
- *asn(mch)*: the ASSERTIONS clause of machine *mch*

Here are some proof obligations for the UniqueIDR machine

- Initialisation.1
  - cst(UniqueIDR) $\wedge$ ctx(UniqueIDR) $\Rightarrow$ 0 $\in$ N ,i.e.,
    IDS = 1..maxids $\Rightarrow$ 0 $\in$ N (cst is empty for this machine)
- Initialisation.2
  - cst(UniqueIDR) $\wedge$ ctx(UniqueIDR) $\Rightarrow$ {} = 1..0

- allocID.1
  - cst(UniqueIDR) $\wedge$ ctx(UniqueIDR) $\wedge$ inv(UniqueIDR) $\wedge$ asn(UniqueIDR) $\wedge$ pre(allocID) $\Rightarrow$
    lastID + 1 $\in$ IDS - usedIDS, i.e.,
    IDS = 1..maxids $\wedge$ lastID $\in$ N $\wedge$ usedIDS = 1.. lastID $\wedge$ usedIDS $\neq$ IDS $\Rightarrow$
    lastID + 1 $\in$ IDS - usedIDS
- allocID.2
  - cst(UniqueIDR) $\wedge$ ctx(UniqueIDR) $\wedge$ inv(UniqueIDR) $\wedge$ asn(UniqueIDR) $\wedge$ pre(allocID) $\Rightarrow$
    lastID + 1 $\in$ N
- allocID.3
  - cst(UniqueIDR) $\wedge$ ctx(UniqueIDR) $\wedge$ inv(UniqueIDR) $\wedge$ asn(UniqueIDR) $\wedge$ pre(allocID) $\Rightarrow$
    usedIDS $\cup$ {lastID + 1} = 1..lastID + 1
- FreeIDS.1

...

Remarks

- Again: (too?) many details … is B a too restrictive language?
- … but automatically generated
- most proofs are trivial
- all are supported by the interactive tool
- some confusion between different layers (UL assertions mixed with LL ones)
- here is a sample of a fully formal proof:
- ***

The **B** method for reactive systems

- Essentially it consists of describing a collection of communicating abstract machines
- The rationale of the approach is similar to Statechart
- It is not clear (to me) how deep and complete is the formalization of its semantics.
- Presentation omitted in this course. Further reading available in
  Sekerinski & Sere (eds) “Program development by Refinement: case studies using the B method”, Springer, 1999.

Outdated material

Model checking

…

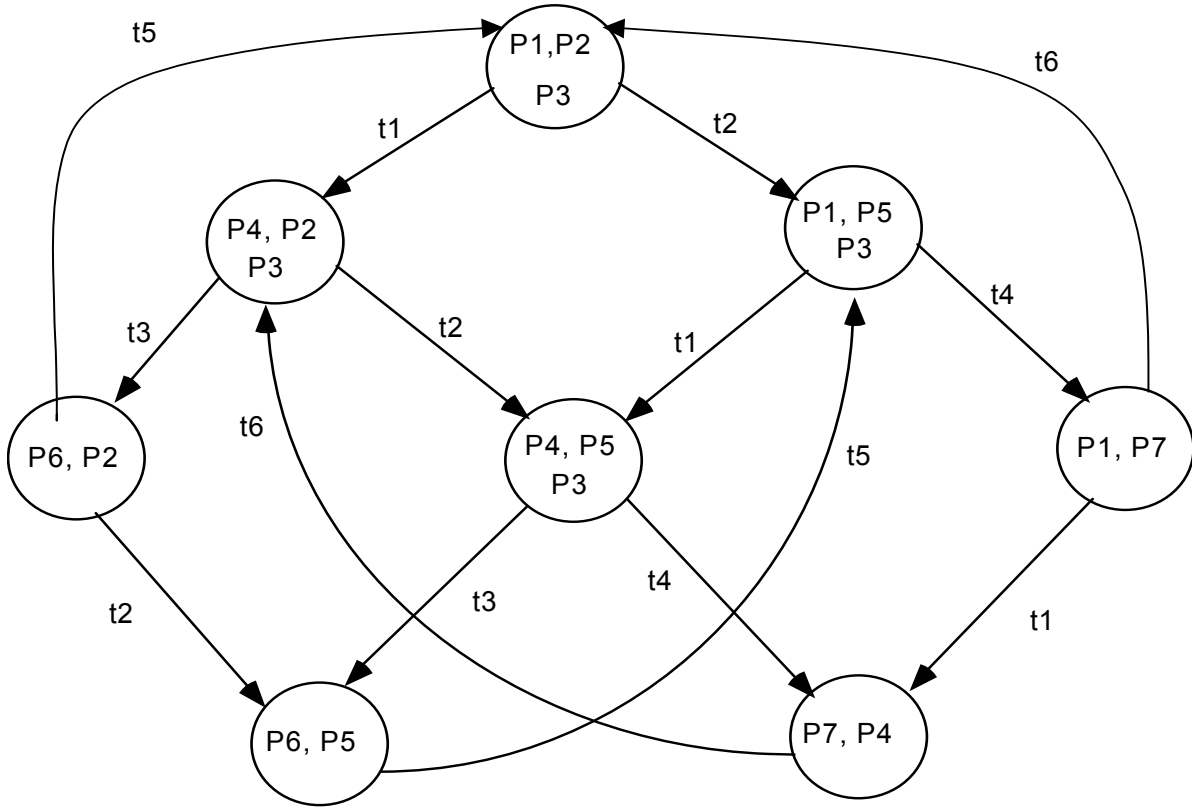
A "new" formal verification technique: Model checking

- Traditionally -and incorrectly- correctness proof are "opposed" to testing as well as theory is "opposed" to practice, academy to industry, etc.
- Actually they are complementary -and potentially mutually beneficial- verification techniques:
  - testing is simpler and more intuitive
  - but it does not provide any certainty
  - correctness proofs, in principle, provide guaranteed correctness
  - but they are often complex and error prone themselves
  - …
- These -and more- remarks should be taken into account when designing real (verification) *methods* (we will go back to the issue)
- Next we review a fairly new verification technique, which, under some circumstances, aims at joining pros of both testing and mathematical proof, i.e., verifying "by running the system -better, a model thereof" and obtaining mathematical certainty: model checking

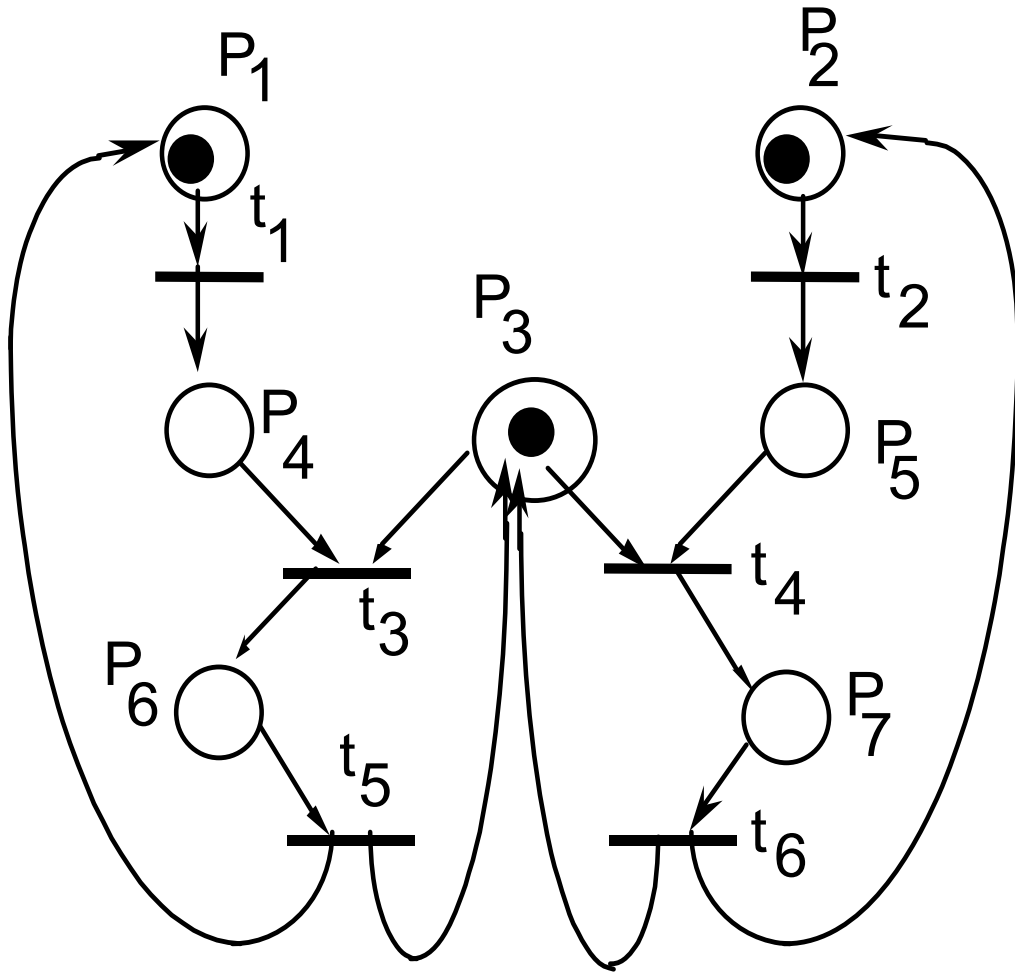
- The very idea of model checking:
  exploiting the fact that correctness, i.e., guaranteeing system requirements, though undecidable in general, is indeed decidable when the system is (modeled by) a finite state machine (FSM) [though, often of exponential complexity …].
- Original paper by Clarke et al.
  - Reference: Clarke, E.M., Emerson, E. A., Sistla, A.P., Automatic verification of finite state concurrent systems using temporal logic specifications., ACM TOPLAS Vol. 8, N.2, pp. 244-263, 1986.
  - Became soon highly popular
  - Lot of further research/extensions/tools
  - Significant industrial successes (mainly in the field of HW design)
- System described by a FSM
- Properties (requirements) formalized with a kind of temporal logic
- (Inductive -and somewhat exhaustive) algorithm to verify whether the machine guarantees the properties; if not a counterexample is provided

Let us illustrate the essentials of model checking through a running example

- First, a FSM description of the system is given:



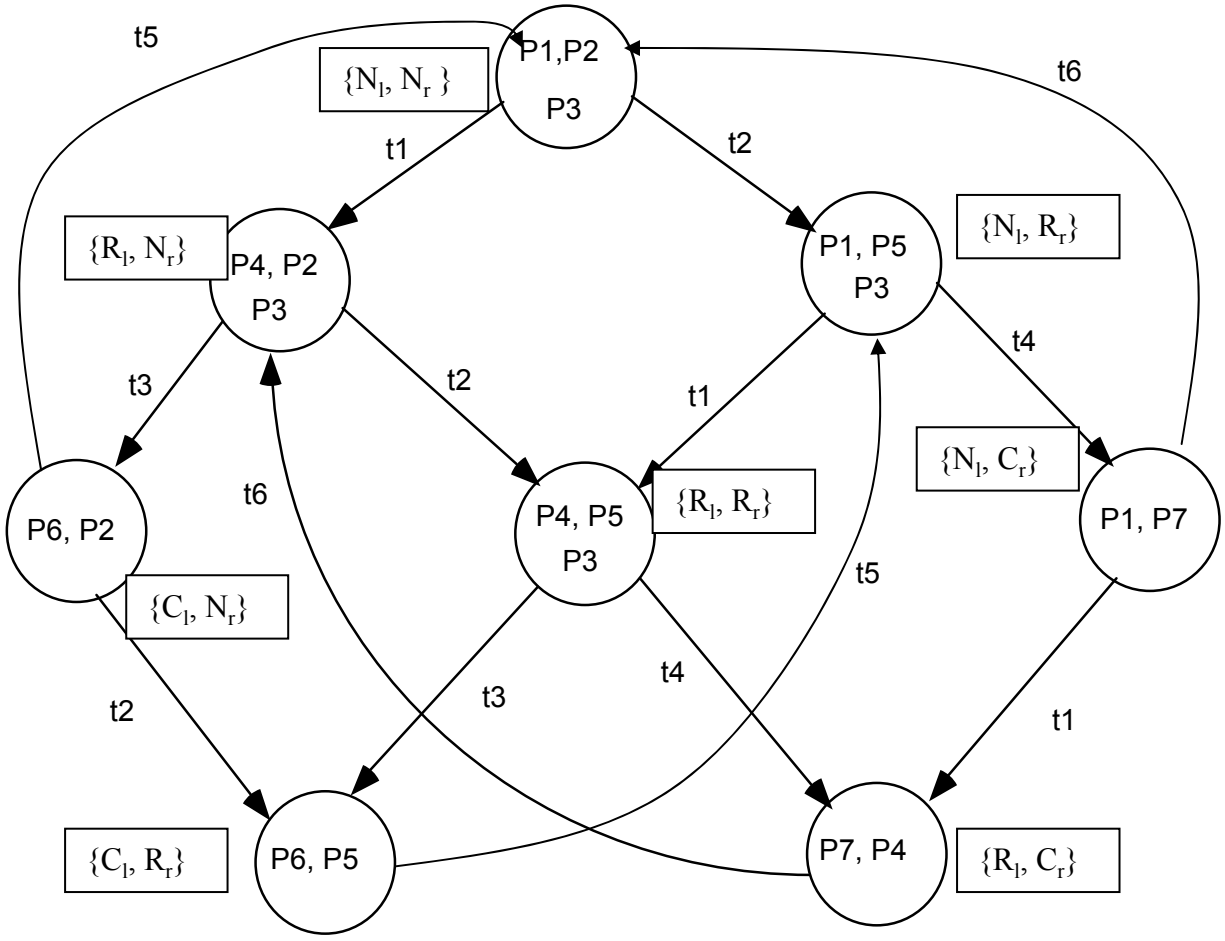
Note: this is a FSM "version" of the more understandable Petri net



- Second step: a desired -or feared- property is formalized.
  The original and traditional syntax for such formulas is of propositional-temporal logic flavor:

  – A finite set AP of *atomic propositions* is defined. For instance:
    - C_l, C_r, mean that the processes described by the left and right portions of the net, respectively, are in the critical region, i.e., "own" the resource modeled by place P_3.
    - N_l, N_r, mean that the two processes are in the non critical region (tokens are in P_1 and P_2, respectively)
    - R_l, R_r, mean that the two processes requested the shared resource (tokens are in P_4 and P_5, respectively).

  – A subset of AP is associated with each state of the FSM. Such an association defines all elementary properties that hold in a given state. For instance, when the system is in state $<P_1, P_2, P_3>$, neither process is in the critical region; thus, we associate the subset $\{N_l, N_r\}$ with state $<P_1, P_2, P_3>$:


t5
P1,P2
P3
t6
{Nl, Nr }
t1
t2
{Rl, Nr}
P4, P2
P3
P1, P5
P3
{Nl, Rr}
t3
t2
t1
t4
{Nl, Cr}
P6, P2
t6
P4, P5
P3
{Rl, Rr}
P1, P7
t5
{Cl, Nr}
t2
t3
t4
t1
{Cl, Rr}
P6, P5
P7, P4
{Rl, Cr}

- Then, more complex formulas are built, inductively, in the typical way:
- Let F_1, F_2 denote two generic formulas. Then the following are also formulas:

  - F_1 **and** F_2, **not** F_1, F_1 **or** F_2 with the usual meaning of the propositional operators **not**, **or**, **and**.
  - $AX(F_1)$: holds when the FSM is in a given state **s** (called the *current state*), if and only if F_1 holds in every state that can be reached from **s** in a single step.
  - $EX(F_1)$, holds when the FSM is in state **s**, if and only if F_1 holds in some state that can be reached from **s** in a single step.
  - $A(F_1 \mathbf{U} F_2)$ holds when the FSM is in state **s**, if and only if for every sequence σ of transitions starting from **s**, σ has a prefix $\sigma 1$ such that F_2 holds in state s' reached by $\sigma 1$ and F_1 holds in all states preceding s' in s1. For this reason **U** is called the *Until* operator)
  - $E(F_1 \mathbf{U} F_2)$ holds when the FSM is in state **s**, if and only if there is some sequence σ of transitions starting from **s** such that σ has a prefix σ_1 such that F_2 holds in state s' reached by σ_1 and F_1 holds in all states preceding s' during the application of σ_1.
  - $AF(F_2)$ (respectively $EF(F_2)$) is an abbreviation $A(TRUE \mathbf{U} F_2)$ (respectively $E(TRUE \mathbf{U} F_2)$; they are read "F_2 is *inevitable*" and "F_2 *potentially* holds", respectively.

- For instance:

- $\mathsf{EF}(C_l)$ holds in every state but $\mathsf{AF}(C_l)$ does not hold in any state that does not include P_6; this means that there is always a computation that allows the left process to enter the critical region, but there is no guarantee that the left process accesses the shared resource unless it already owns it.

- At this point …. We “push the button”!

- In fact such formulas are decidable for FSMs

- This has been the major source of the success of model checking

- We do not give the details of the model checking algorithm(s). However:
  - It exploits the typical finiteness of these machines: there are finite many states, each one with finite many possible elementary properties.
  - Thus, properties that predicate machine behavior even for infinite sequences of transitions can be checked inductively in a finite way thanks to the typical cyclic behavior of FSMs: once more, the *pumping lemma* comes into play.

- There are drawbacks, of course:
  - The system must be modeled by a finite set of states (if a system is finite state, its properties too can be "propositional-based")
  - If the original system is not FS, we can approximate it in such a way (finite integers, …), but …
  - even if the approximation is realistic, we are faced with the typical *combinatorial explosion* (a k-bit register has 2^k states)

- many ad-hoc techniques have been developed to deal with such an enormous number of -potential- states:
  - partitioning, symbolic execution, …
- some of them can be applied even to infinite state spaces (provided we are able to partition an infinite space into a finite set of “equivalence classes”, e.g.:
  - Temperature below -30, temperature between - 30 and + 30, temperature above +30)

- All this stuff is the matter of promising and very active research activity

- So far, as we said, major successes in the HW sector, rather than in the SW:
- HW exhibits more “regularities”