Process Algebras and Concurrent Systems

Roberto Bruni\textsuperscript{1}  Rocco De Nicola\textsuperscript{2}

\textsuperscript{1}Dipartimento di Informatica
Università di Pisa

\textsuperscript{2}Dipartimento di Sistemi ed Informatica
Università di Firenze

Process Algebras and Concurrent Systems
May–June 2006

Outline

- Basic Models for Concurrent Programs and Behavioral Equivalences
- Operators for Concurrency and Process Algebras
- CCS and Bisimulations
- Axiomatizations of Equivalences
- Modal and Temporal Logics and Bisimulations
- Pi-Calcolo
- Join - Djoin
- Spi-Calcolo
- ???
- Klaim and X-klaim
A problem

Write a program that terminates if and only if the total function \( f \) has a (positive or negative) zero and proceeds indefinitely otherwise.

Assume to have a program that looks for positive zero:

\[
S_1 = 
\begin{align*}
\text{found} & := \text{false}; x := 0; \\
\text{while} \ (\text{not found}) \\
& \quad \text{do } x := x + 1; \text{ found } := (f(x) = 0) \ \text{od}
\end{align*}
\]

Starting from this we can build the program looking for a negative zero.

\[
S_2 = 
\begin{align*}
\text{found} & := \text{false}; y := 0; \\
\text{while} \ (\text{not found}) \\
& \quad \text{do } y := y - 1; \text{ found } := (f(y) = 0) \ \text{od}
\end{align*}
\]
An obvious solution would be running $S_1$ and $S_2$ in parallel:

$$S_1 \parallel S_2$$

If $f$ has a positive zero and not a negative one, and $S_1$ terminates before $S_2$ starts, the latter sets found to false and starts looking indefinitely for the nonexisting zero.

The problem is due to the fact that found is initialized to false twice.

**LESSON 1**

**USING SHARED VARIABLES MAY LEAD TO PROBLEM**

Let us consider a solution that initializes found only once.

```plaintext
found := false; (R1 || R2) where
R1 = x := 0; while (not found)
    do x := x+1; found := (f(x) = 0) od
R2 = y := 0 while (not found)
    do y := y-1; found := (f(y) = 0) od
```

If $f$ has (again) only a positive zero assume that:

- R2 proceeds just up to the while body (after the do) and is preempted
- R1 computes till a zero is found
- R2 gets the CPU back

When R2 restarts it executes the while body and may set found to false - found := (f(y) = 0). The program then would not terminate because it would look for a non existing negative zero.
The problem is due to the fact that found is set to false (by means of 
found := f(y) = 0) after it has already got the value true.

LESSON 2
NO ASSUMPTION ON THE RELATIVE SPEED OF PROGRAMS CAN
BE MADE

Let us see what happens if we do not perform ”unnecessary” assignments
and only assign true when we find a x or a y such that f(x) = 0 or f(y) = 0.

found := false; (T1 || T2) where
T1 = x := 0; while not found
   do x:= x+1; if f(x) = 0 then found := true fi od
T2 = y := 0; while not found
   do y:= y-1; if f(y) = 0 then found := true fi od

Assume that f has only a positive zero and that T2 gets the CPU to keep
it until it terminates. Since this will never happen, T1 will never get the
chance to find its zero.
This problem is due to the considered scheduler of the CPU, to avoid problems we would need a non fair scheduler; but this is a too strong assumptions.

**LESSON 3**

NO ASSUMPTION CAN BE MADE ON THE SCHEDULING POLICY OF THE CPU.

**Attempt 4**

To avoid assumptions on the scheduler, we could think of adding control to the programs and let them "pass the baton" once they have got their "chance" to execute for a while.

\[
\begin{align*}
\text{turn} := 1; \text{found} := \text{false}; (P1 \parallel P2) \text{ where} \\
P1 &= x := 0; \text{while not found do wait turn} := 1 \text{ then} \\
& \hspace{1cm} \text{turn} := 2; x := x+1; \text{if } f(x) = 0 \text{ then found} := \text{true} \text{ fi od} \\
P2 &= y := 0; \text{while not found do wait turn} := 2 \text{ then} \\
& \hspace{1cm} \text{turn} := 1; y := y-1; \text{if } f(y) = 0 \text{ then found} := \text{true} \text{ fi od}
\end{align*}
\]

If P1 finds a zero and stops when P2 has already set turn:= 1, P2 would be blocked by the wait command because nobody can change the value of turn.
The program does not terminate because of the waiting of an impossible event.

LESSON 4
ON TERMINATION CARE IS NEEDED FOR OTHER PROCESSES.

A CORRECT Solution!

Pass (again) the baton before terminating.

```plaintext
turn:= 1; found := false; (P1; turn:= 2 || P2; turn:= 1)
where
P1 = x := 0; while not found do
    wait turn:= 1 then
    turn:= 2; x:= x+1;
    if f(x) = 0 then found := true fi
od
P2 = y := 0; while not found do
    wait turn:= 2 then
    turn:= 1; y:= y-1;
    if f(y) = 0 then found := true fi
od
```
Operational Semantics for Concurrent Processes

Systems behaviour is described by associating to each program a behaviour represented as a transition graph.

We have two possibilities

**Kripke Structures**

State Labelled Graphs: States are labelled with the properties that are considered relevant (e.g. the value of - the relation between - some variables)

**Labelled Transition Systems**

Transition Labelled Graph: Transition between states are labelled the action that induces the transition to move from one state to another.

We shall mainly rely on **Labelled Transition Systems** and **actions** will play an important role
Finite State automata

**Definition**

A *finite state automaton* $M$ is a 5-tuple $M = (Q, A, \rightarrow, q_0, F)$ where
- $Q$ is a finite set of states
- $A$ is the alphabet
- $\rightarrow \subseteq Q \times (A \cup \{\varepsilon\}) \times Q$ is the transition relation
- $q_0 \in Q$ is a special state called initial state,
- $F \subseteq Q$ is a set of states (final states)

![Finite state automaton](Bruni, De Nicola (UNIPI, DSI-UNIFI) Process Algebras and Concurrent Systems IMT Lucca 15 / 84)

Labelled Transition Systems

**Definition**

A Labelled Transition System $S$ is a 4-tuple $S = (Q, A, \rightarrow, q_0)$ where
- $Q$ is a set of states
- $A$ is a finite set of actions
- $\rightarrow \subseteq Q \times A \times Q$ is a ternary relation called transition relation it is often written $q \xrightarrow{a} q'$ instead of $(q, a, q') \in \rightarrow$
- $q_0 \in Q$ is a special state called initial state.

![Labelled Transition System](Bruni, De Nicola (UNIPI, DSI-UNIFI) Process Algebras and Concurrent Systems IMT Lucca 16 / 84)
A Simple Example

Example (Bill-Ben)

\[ S = (Q, A, \rightarrow) \] where:
- \( Q = \{ q_0, q_1, q_2, q_3, q_4 \} \)
- \( A = \{ \text{play}, \text{work}, \tau \} \)
- \( \rightarrow = \{(q_0, \text{play}, q_1), (q_0, \text{work}, q_2), (q_1, \text{work}, q_3), (q_2, \text{play}, q_3), (q_3, \tau, q_4)\} \)

Internal and External Actions

An elementary action of a system represents the atomic (non-interruptible) abstract step of a computation that is performed by a system to move from one state to the other.

Actions represent various activities of concurrent systems:
- Sending a message
- Receiving a message
- Performing some activity
- Synchronizing with other processes
- ...

We have two main types of atomic actions:
- Visible Actions
- Internal Actions
Operators for Concurrency and Process Algebras

Process Algebras and Concurrent Systems

Operators for Processes Modelling

Processes are composed via a number of basic operators

- Basic Processes
- Action Prefixing
- Sequentialization
- Choice
- Parallel Composition
- Abstraction
- Infinite Behaviours
Regular Expressions

\[ E ::= 0 \mid 1 \mid a \mid E + E \mid E; E \mid E^* \]

R. E. have a denotational semantics that associates to each expression the language (set of strings) generated by it.

\[
\begin{align*}
\mathcal{L}[0] &= \{\} \\
\mathcal{L}[1] &= \{\epsilon\} \\
\mathcal{L}[a] &= \{a\} \quad \text{(per } a \in A) \\
\mathcal{L}[e + f] &= \mathcal{L}[e] \cup \mathcal{L}[f] \\
\mathcal{L}[e \cdot f] &= \mathcal{L}[e] \cdot \mathcal{L}[f] \\
\mathcal{L}[e^*] &= (\mathcal{L}[e])^* 
\end{align*}
\]

Operational Semantics of Regular Expressions

\[ E ::= 0 \mid 1 \mid a \mid E + E \mid E; E \mid E^* \]

\[
\begin{align*}
\text{(Tic)} & \quad 1 \xrightarrow{\epsilon} 1 \\
\text{(Sum)} & \quad e \xrightarrow{\mu} e' \\
& \quad e + f \xrightarrow{\mu} e' \\
\text{(Seq)} & \quad e \xrightarrow{\mu} e' \\
& \quad e; f \xrightarrow{\mu} e'; f \\
\text{(Star)} & \quad e^* \xrightarrow{\epsilon} 1 \\
& \quad e^* \xrightarrow{\mu} e'; e^*
\end{align*}
\]

Table: We assume \( a \in A \) and \( \mu \in A \cup \{\epsilon\} \).
Why Introducing Operators

How would you describe a very large automaton / LTS?

**As a table?**
Rows and columns are labelled by states, entries are either empty or marked with a set of actions.

**As a listing of triples?**
\[ \rightarrow = \{ (q_0, a, q_1), (q_0, a, q_2), (q_1, b, q_3), (q_1, c, q_4), (q_2, \tau, q_3), (q_2, \tau, q_4) \}. \]

**As a more compact listing of triples?**
\[ \rightarrow = \{ (q_0, a, \{q_1, q_2\}), (q_1, b, q_3), (q_1, c, q_4), (q_2, \tau, \{q_3, q_4\}) \}. \]

**As XML?**
\[
\begin{align*}
\text{<lts><ar><st>q0</st><lab>a</lab><st>q1</st></ar>...<lts>}
\end{align*}
\]

---

Why Introducing Operators - ctd

**Linguistic aspects are important!**
The previous solutions are ok for machines... but not much for human beings.

Are prefix and sum enough?

**To describe (finite) systems maybe yes**
\[ p = a.b.(c + d), \quad q = a.(b.c + b.d), \quad r = a.b.c + a.c.d \]

**But having more operators can be useful!**
- to design systems in a structured way (e.g. \( p|q \))
- to understand systems
- to implement systems
- to reason about systems
Operational Semantics

To each process built using the above operators we associate an LTS by relying on structural induction to define the meaning of each operator.

**Definition (Inference Systems)**

An inference system is a set of inference rule of the form

\[ \frac{p_1, \ldots, p_n}{q} \]

In our case for a generic operator \( \text{op} \) we would have:

\[ \frac{E_{i_1} \xrightarrow{\alpha_1} E'_{i_1} \ldots E_{i_m} \xrightarrow{\alpha_m} E'_{i_m}}{\text{op}(E_1, \ldots, E_n) \xrightarrow{\alpha} \text{op}(E'_1, \ldots, E'_n)} \]

where \( \{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\} \).

The Elegance of Operational Semantics

**Automata as expressions**

Few SOS rules define all the automata that can ever be specified with the operators at hand: Given any expression, the rules are used to derive the corresponding automata. Note that the set of rules is fixed once for all.

**Structural induction**

The interaction of complex systems is defined in terms of the behavior of their components.

**A remark**

The LTS is the least one satifying the inference rules.

**Rule induction**

A property is true for the whole LTS if whenever it holds for the premises of each rule, then it holds for the conclusion.
Some Examples on Regular Expressions

\[(a + b)^* \xrightarrow{a} 1; (a + b)^*\]

\[
\begin{align*}
(a + b)^* & \xrightarrow{a} 1 \\
(a + b)^* & \xrightarrow{a} 1 \\
(a + b)^* & \xrightarrow{a} 1 \\
\end{align*}
\]

(Atom)

(Star₂)

\[(1; (a + b)^*) \xrightarrow{\varepsilon} (a + b)^*\]

\[
\begin{align*}
1 & \xrightarrow{\varepsilon} 1 \\
1 & \xrightarrow{\varepsilon} 1 \\
1 & \xrightarrow{\varepsilon} 1 \\
\end{align*}
\]

(Tic)

(Seq₂)

Another Example On Regular Expressions

\[(a^* + b^*)^* \xrightarrow{b} 1; b^*; (a^* + b^*)^*\]

\[
\begin{align*}
(a^* + b^*)^* & \xrightarrow{b} 1 \\
(a^* + b^*)^* & \xrightarrow{b} 1 \\
(a^* + b^*)^* & \xrightarrow{b} 1 \\
\end{align*}
\]

(Atom)

(Star₂)

(Star₂)

\[(a^* + b^*)^* \xrightarrow{c} 1; b^*; (a^* + b^*)^*\]

A remark

\((a^* + b^*)^* \xrightarrow{c} 1\) would not contradict any rule, but it cannot be in the least LTS, because it cannot be inferred.
Some advanced proof methods

- **Proof by obviousness:** so evident it need not to be mentioned
- **Proof by general agreement:** all in favor?
- **Proof by majority:** when general agreement fails
- **Proof by plausability:** it sounds good
- **Proof by intimidation:** who is saying that is false!?
- **Proof by terror:** when intimidation fails
- **Proof by lost reference:** I saw it somewhere
- **Proof by authority:** Don Knuth said it was true
- **Proof by divine word:** Lord said *let it be true*
- **Proof by intuition:** I have this feeling...
- **Proof by deception:** everybody please turn their backs...
- **Proof by logic:** it is on the textbook, hence it must be true

Basic Processes

Inactive Process

Is usually denoted by

- `nil`
- `0`
- `stop`

Its semantics is characterized by the fact that there is no rule to define its transition.

A broken vending machine

```
nil
```

Does not accept coins and does not gives drinks.
Basic Processes ctd

Termination

Termination is sometimes denoted by

- `exit`
- `skip`

that can only perform the special action $\sqrt{}$ ("tick") to indicate termination and become `nil`

\[
\text{exit} \xrightarrow{\sqrt{}} \text{stop}
\]

A gentle broken vending machine

\[
\text{exit}
\]

Does not accept coins, does not gives drinks but says everything is ok.

Action Prefixing

Prefixing

For each action $\mu$ there is a unary operator

- $\mu.$
- $\mu \rightarrow .$

that builds from process $E$ a new process $\mu.E$ that performs action $\mu$ and then behaves like $E$.

\[
\mu.E \xrightarrow{\mu} E
\]

A "one shot" vending machine

\[
\text{coin} \rightarrow \text{choc} \rightarrow \text{stop}
\]

Accepts a coin and gives a chocolate then stops.
Action Prefixing ctd

Action as processes

Instead of prefixing, some calculi rely on considering actions as basic processes.

\[ a \xrightarrow{a} \text{stop} \]

A gentle dishonest vending machine

\[ \text{coin} \]

Accepts a coin and says it’s ok.

Sequential Composition

Sequentialization

The binary operator for sequential composition is denoted by

\[ . ; . \]

\[ . \gg . \]

If \( E \) and \( F \) are processes \( E; F \) executes \( E \) and then \( F \)

\[
\begin{align*}
E \xrightarrow{\mu} E' \\
E; F \xrightarrow{\mu} E'; F
\end{align*}
\]

\[ (\mu \neq \sqrt{\cdot}) \]

\[
\begin{align*}
E \xrightarrow{\sqrt{\cdot}} E' \\
E; F \xrightarrow{\tau} F
\end{align*}
\]

Another "one shot" vending machine

\[ \text{coin; choc} \]
Disabling Operator

The disabling binary operator

\[ E \mu \rightarrow E' \]

permits to interrupt some actions when specific events happen.

\[
\begin{align*}
E \triangleright F & \quad \mu \rightarrow E' \triangleright F \\
E \triangleright F & \quad \triangleright E' \\
E \triangleright F & \quad \mu \rightarrow F'
\end{align*}
\]

A cheating customer

\[(coin \rightarrow choc \rightarrow stop) \triangleright (bang \rightarrow choc \rightarrow stop)\]

This describes a vending machine that when "banged" gives away a chocolate without getting the coin.

Choice - 1

Non-deterministic Choice

\[
\begin{align*}
E \mu \rightarrow E' \\
F \mu \rightarrow F'
\end{align*}
\]

\[
\begin{align*}
E + F & \quad \mu \rightarrow E' \\
F + E & \quad \mu \rightarrow F'
\end{align*}
\]

User’s Choice

\[ coin \rightarrow (choc \rightarrow stop + water \rightarrow stop)\]

Machine’s Choice

\[ coin \rightarrow choc \rightarrow stop + coin \rightarrow water \rightarrow stop\]
**Choice - 2**

**Internal Choice**

\[ E \oplus F \xrightarrow{\tau} E \]

\[ E \oplus F \xrightarrow{\tau} F \]

**Machine's Choice**

\[ \text{coin} \rightarrow (\text{choc} \rightarrow \text{stop} \oplus \text{water} \rightarrow \text{stop}) \]

---

**Choice - 3**

**External Choice**

\[ \frac{E \xrightarrow{\alpha} E'}{E \boxdot F \xrightarrow{\alpha} E'} (\alpha \neq \tau) \]

\[ \frac{F \xrightarrow{\alpha} F'}{E \boxdot F \xrightarrow{\alpha} F'} (\alpha \neq \tau) \]

\[ \frac{E \xrightarrow{\tau} E'}{E \boxdot F \xrightarrow{\tau} E' \boxdot F} \]

\[ \frac{F \xrightarrow{\tau} F'}{E \boxdot F \xrightarrow{\tau} E \boxdot F'} \]

**User's Choice**

\[ \text{coin} \rightarrow ((\text{choc} \rightarrow \text{stop} \oplus \text{water} \rightarrow \text{stop}) \boxdot \text{water} \rightarrow \text{stop}) \]
Different Transitions

External Choice

\[ coin \rightarrow ((choc \rightarrow stop \oplus water \rightarrow stop) \oplus water \rightarrow stop) \]
\[ \xrightarrow{\text{coin}} \]
\[ (choc \rightarrow stop \oplus water \rightarrow stop) \oplus water \rightarrow stop \]
\[ \xrightarrow{\tau} \]
\[ (choc \rightarrow stop \oplus water \rightarrow stop) \]

Internal Choice

\[ coin \rightarrow ((choc \rightarrow stop \oplus water \rightarrow stop) \oplus water \rightarrow stop) \]
\[ \xrightarrow{\text{coin}} \]
\[ (choc \rightarrow stop \oplus water \rightarrow stop) \oplus water \rightarrow stop \]
\[ \xrightarrow{\tau} \]
\[ choc \rightarrow stop \oplus water \rightarrow stop \]
\[ \xrightarrow{\tau} \]
\[ choc \rightarrow stop \]

Parallel Composition - 1

Milner’s Parallel

\[ \frac{E \xrightarrow{\mu} E'}{E|F \xrightarrow{\mu} E'|F} \quad \frac{F \xrightarrow{\mu} F'}{E|F \xrightarrow{\mu} E'|F'} \quad \frac{E \xrightarrow{\alpha} E' \quad F \xrightarrow{\overline{\alpha}} F'}{E|F \xrightarrow{\tau} E'|F'} \] \((\alpha \neq \tau)\)

User-Machine interaction

\[ (\overline{coin} \rightarrow (\overline{choc} \rightarrow stop \oplus \overline{water} \rightarrow stop)) \mid (\overline{coin} \rightarrow choc \rightarrow stop) \]
We can have different interactions

### Appropriate Interaction

\[
\begin{align*}
(coin \rightarrow (choc \rightarrow stop \oplus water \rightarrow stop)) & \mid (coin \rightarrow choc \rightarrow stop) \\
\rightarrow & \\
(choc \rightarrow stop \oplus water \rightarrow stop) & \mid (choc \rightarrow stop) \\
\rightarrow & \\
(choc \rightarrow stop) & \mid (choc \rightarrow stop)
\end{align*}
\]

### Inappropriate Interaction - Coin thrown away

\[
\begin{align*}
(coin \rightarrow (choc \rightarrow stop \oplus water \rightarrow stop)) & \mid (coin \rightarrow choc \rightarrow stop) \\
\rightarrow & \\
(choc \rightarrow stop \oplus water \rightarrow stop) & \mid (choc \rightarrow stop) \\
\rightarrow & \\
(water \rightarrow stop) & \mid (choc \rightarrow stop)
\end{align*}
\]

---

### Parallel Composition - 2

---

### Merge Operator with Synchronization Function

\[
\begin{align*}
E \xrightarrow{\mu} E' & \quad F \xrightarrow{\mu} F' \\
E \parallel F \xrightarrow{\mu} E' \parallel F & \quad E \parallel F \xrightarrow{\mu} E \parallel F'
\end{align*}
\]

\[
\begin{align*}
E \xrightarrow{a} E' & \quad F \xrightarrow{b} F' \\
E \parallel F \xrightarrow{\gamma(a,b)} E' \parallel F'
\end{align*}
\]

with \( \mu \in \Lambda \cup \{\tau\} \)

---

### Another interaction

\[
\begin{align*}
geticn.(giveChoc.nil + giveWater.nil) & \parallel putCoin.getChoc.nil
\end{align*}
\]

with \( \gamma(getCoin, putCoin) = ok \) e \( \gamma(giveChoc, getChoc) = ok \).
Parallel Composition - 3

Communication Merge

\[
\begin{align*}
E \xrightarrow{a} E' & \quad F \xrightarrow{b} F' \\
E \parallel F & \xrightarrow{\gamma(a,b)} E' \parallel F'
\end{align*}
\]

Left Merge

\[
\begin{align*}
E & \xrightarrow{\mu} E' \\
E \parallel F \xrightarrow{\mu} E' \parallel F
\end{align*}
\]

Interleaving

\[
\begin{align*}
\begin{align*}
E & \xrightarrow{\mu} E' \\
E \parallel F \xrightarrow{\mu} E' \parallel F
\end{align*}
\quad
\begin{align*}
F & \xrightarrow{\mu} F' \\
E \parallel F \xrightarrow{\mu} E \parallel F'
\end{align*}
\end{align*}
\]

Parallel Composition - 4

Hoare’s Parallel

\[
\begin{align*}
\begin{align*}
E & \xrightarrow{\mu} E' \\
E \parallel [L] \parallel F \xrightarrow{\mu} E' \parallel [L] \parallel F
\end{align*}
\quad
\begin{align*}
F & \xrightarrow{\mu} F' \\
E \parallel [L] \parallel F \xrightarrow{\mu} E \parallel [L] \parallel F'
\end{align*}
\end{align*}
\]

The operator \([L]\) is strongly related with some of the operators seen before.

- \([L]\) and \(\parallel\) are equivalent if \(\gamma(a,a) = a \forall a \in L\),
- \([L]\) and \(\parallel\) are equivalent if \(L = \emptyset\),
Synchronization Algebra

Most operators for parallel composition can be expressed in terms of suitable synchronization algebras (assume $E \rightarrow E$ for all $E$).

**Definition**

A Synchronization Algebra is a 4-tuple $\langle \Lambda, \cdot, 0, \alpha \rangle$ where

- $\Lambda$ is a set of labels containing the special labels $\cdot \in 0$,
- $\cdot$ is an associative and commutative binary operation $\Lambda \times \Lambda \rightarrow \Lambda$ that satisfies:
  - $a \cdot 0 = 0$ for all $a \in \Lambda$,
  - $\cdot \cdot \cdot = \cdot$,
  - $a \cdot b = \cdot$ implies $a = b = \cdot$, for all $a, b \in \Lambda$.

$$
\begin{array}{c|c|c|c}
\cdot & \cdot & \alpha & 0 \\
\hline
\cdot & \cdot & \alpha & 0 \\
\hline
\alpha & \cdot & 0 & 0 \\
\hline
0 & 0 & 0 & 0 \\
\end{array}
$$

Value Passing

**Single Evolutions**

$$
\begin{array}{c}
a(x).E \xrightarrow{a(v)} E\{v/x\}
\end{array}
$$

**Interaction**

$$
\begin{array}{c}
E \xrightarrow{a(v)} E' \\
F \xrightarrow{a(v)} F' \\
E|F \xrightarrow{\tau} E'|F'
\end{array}
$$
Conditional Execution

\[
\text{val}(e) = \text{true} \quad E \xrightarrow{\mu} E' \\
\text{if } e \text{ then } E \text{ else } F \xrightarrow{\mu} E' \\
\text{val}(e) = \text{false} \quad F \xrightarrow{\mu} F'
\]

Let us consider a vending machine that accept 20 cents coins (or higher) and offers a chocolate:

\[
\text{coin}(x). \text{if } x \geq 20 \text{ then } \text{choc.nil} \text{ else } \text{nil}
\]

The user interacts with the machine as follows:

\[
\text{coin}(x). \text{if } x \geq 20 \text{ then } \text{choc.nil} \text{ else } \text{nil} \quad | \quad \text{coin}40.\text{choc.nil}
\]
\[
\text{if } 40 \geq 20 \text{ then } \text{choc.nil} \text{ else } \text{nil} \quad | \quad \text{choc.nil}
\]
\[
\text{nil} \quad | \quad \text{nil}
\]

Abstraction - 1

Restriction

\[
E \xrightarrow{\alpha} E' \\
E \setminus L \xrightarrow{\alpha} E' \setminus L \quad (\alpha, \overline{\alpha} \not\in L)
\]

Forcing Interaction

\[
\left( (\text{coin.ok.nil} \mid \text{ok.}(\text{choc.nil} + \text{water.nil})) \right) \setminus \text{ok} \quad | \quad \text{coin.choc.nil}
\]
\[
\left( (\text{ok.nil} \mid \text{ok.}(\text{choc.nil} + \text{water.nil})) \right) \setminus \text{ok} \quad | \quad \text{choc.nil}
\]
\[
\left( \text{nil} \mid (\text{choc.nil} + \text{water.nil}) \right) \setminus \text{ok} \quad | \quad \text{choc.nil}
\]
\[
\left( \text{nil} \mid \text{nil} \right) \setminus \text{ok} \quad | \quad \text{nil}
\]

A malicious user executing \text{ok.choc.nil} would be stopped.
Abstraction - 2

Hiding

\[
\begin{align*}
E & \xrightarrow{\alpha} E' \\
E/L & \xrightarrow{\alpha} E'/L \\
E/L & \xrightarrow{\tau} E'/L
\end{align*}
\]

\((\alpha \not\in L)\)  \hspace{1cm}  \((\alpha \in L)\)

Avoiding Interaction

\((coin.ok.nil) \ | [ok] \ | ok.(choc.nil + water.nil) \) / ok

The ok signal is internalized thus it cannot be used by a dishonest user.

Abstraction - 3

Renaming

\[
\begin{align*}
E & \xrightarrow{\mu} E' \\
E[f] & \xrightarrow{f(\mu)} E'[f]
\end{align*}
\]

Multilingual Interaction

An Italian user

\underline{soldo. acqua. nil}

can interact with the machine with English indication by applying:

\((\underline{soldo. acqua. nil}) \ [\text{coin/soldo, water/acqua}]\)
Recursion

\[
\frac{E \{ \text{rec } X.E \}/X \xrightarrow{\mu} E'}{\text{rec } X.E \xrightarrow{\mu} E'}
\]

Long Lasting Vending Machine

\[
\text{rec } D. \text{coin.}(\overline{\text{choc. } D} + \overline{\text{water. } D})
\]

\[
\frac{\text{rec } D. \text{coin.}(\overline{\text{choc. } D} + \overline{\text{water. } D})}{\text{coin}}
\]

\[
\frac{\overline{\text{choc.}} \text{rec } D. \text{coin.}(\overline{\text{choc. } D} + \overline{\text{water. } D}) + \overline{\text{water.}} \text{rec } D. \text{coin.}(\overline{\text{choc. } D} + \overline{\text{water. } D})}{\text{choc}}
\]

\[
\frac{\text{rec } D. \text{coin.}(\overline{\text{choc. } D} + \overline{\text{water. } D})}{\text{coin}} \ldots
\]

Infinite Behaviour

Replication

\[
\frac{E \xrightarrow{\mu} E'}{! E \xrightarrow{\mu} ! E | ! E}
\]

or, equivalently

\[
\frac{E \vdash ! E \xrightarrow{\mu} E'}{! E \xrightarrow{\mu} ! E}
\]

The replication operator can be defined by the following equation

\[
! E \triangleq ! E | ! E
\]

that can be expressed in terms of rec as follows: \( \text{rec } X.(E|X) \)

Chocolate ad libitum

\[
\begin{align*}
&! \text{coin. } \overline{\text{choc. } \text{nil}} \\
&\overline{\text{choc. } \text{nil}} \mid ! \text{coin. } \overline{\text{choc. } \text{nil}} \\
&\overline{\text{choc. } \text{nil}} \mid \overline{\text{choc. } \text{nil}} \mid ! \text{coin. } \overline{\text{choc. } \text{nil}} \\
&\overline{\text{nil}} \mid \overline{\text{choc. } \text{nil}} \mid ! \text{coin. } \overline{\text{choc. } \text{nil}} \\
&\overline{\text{nil}} \mid \overline{\text{nil}} \mid ! \text{coin. } \overline{\text{choc. } \text{nil}}
\end{align*}
\]
Infinite Behaviour - 3

Iteration

\[
E^* \xrightarrow{\epsilon} \sqrt{} \quad \text{and} \quad E \xrightarrow{\mu} E' \\
E^* \xrightarrow{\mu} E', E^*
\]

This iteration operator is the classical one of regular expressions.

Puzzle Time: 50 Prisoners Problem

Solve the problem and write down the solution as a process

50 prisoners kept in separate cells got a chance to be released: From time to time one of them will be carried in a special room (in no particular order, possibly multiple times consecutively, but with a fair schedule to avoid infinite wait) and then back to the cell.

The room is completely empty except for a switch that can turn the light either on or off (the light is not visible from outside and cannot be broken). At any time, if any of them says that all the prisoners have already entered the room at least once and this is true, then all prisoners will be released (but if it is false, then the chance ends and they will never be released). Before the challenge starts, the prisoners have the possibility to discuss together some “protocol” to follow.

Can you find a winning strategy for the prisoners?

Note that the initial state of the light in the room is not known.
Problem: Are these three systems equivalent?
Traces/Language Equivalence

Let \( Q, A, \rightarrow \) be an LTS, with \( q \in Q \) and \( s \in A^* \).

### Traces

- \( s \) is a *trace* of \( q \) if there exists \( q' \in Q \) s.t. \( q \xrightarrow{s} q' \).
- \( T(q) \) represents the set of all traces of \( q \).

### Traces Equivalence

Two states \( p, q \) are *trace equivalent*, written \( p =_T q \), if \( T(p) = T(q) \).

---

**Two Traces Equivalent Systems**

![Diagram of two systems with traces labeled as steps: tea, coffee]
**Bisimulation Relation**

**Strong Bisimulation**

A relation $R \subseteq Q \times Q$ is **strong bisimulation** if, for any pair of states $p \in Q$ such that $(p, q) \in R$, the following holds:

- for all $a \in A$ e $p' \in Q$, if $p \xrightarrow{a} p'$ then $q \xrightarrow{a} q'$ for some $q' \in Q$ such that $(p', q') \in R$;
- for all $a \in A$ e $q' \in Q$, if $q \xrightarrow{a} q'$ then $p \xrightarrow{a} p'$ for some $p' \in Q$ such that $(p', q') \in R$.

**Bisimilarity**

Two states $p, q \in Q$ are strongly **bisimilar**, written $p \sim q$, if there exists a strong bisimulation $R$ such that $(p, q) \in R$.

**Two Systems that are not bisimilar**

- States $p_0$ and $q_0$ are not strongly bisimilar.
- If they were equivalent, also states $p_1$ e $q_1$, had to be so.
- There is no strong bisimulation $R$ that contains $(p_1, q_1)$.
- The $c$-transition from $p_1$ cannot be simulated by $q_1$.
These three systems are not bisimulation equivalent

Two bisimilar Systems

\[ R \triangleq \{ \langle p_0, q_0 \rangle, \langle p_0, q_2 \rangle, \langle p_1, q_1 \rangle, \langle p_2, q_1 \rangle \} \] is a strong bisimulation
A General Observational Approach

When defining behavioural equivalences, we do not want to distinguish between systems that cannot be taken apart by external observers. Because of this we introduce three notions:

- Observers
- Observations
- Successful Observations

Those systems that satisfy (lead to successful observations) the same observers are considered equivalent.

An observer is an LTS having actions from $A_w \triangleq A \cup \{w\}$, with $w \notin A$;

To determine whether a state $q$ satisfies an observer $o$ the set $OBS(q, o)$ of all computations from $\langle q, o \rangle$ is considered.

A process may satisfy an observer always or sometimes.

Observations

Given two LTS $\langle Q, A, \rightarrow \rangle$ and $\langle O, A_w, \rightarrow \rangle$, and two states $q \in Q$ and $o \in O$, an observation $c$ from $\langle q, o \rangle$ is a sequence of pairs $\langle q_i, o_i \rangle$, such that

- $\langle q_0, o_0 \rangle = \langle q, o \rangle$;
- the transition $\langle q_i, o_i \rangle \xrightarrow{a} \langle q_{i+1}, o_{i+1} \rangle$ can be proved using:

$$
\begin{array}{ccc}
E & \xrightarrow{a} & E' \\
F & \xrightarrow{a} & F'
\end{array}
\quad a \in A
$$

- the last element of the sequence, say $\langle q_k, o_k \rangle$, is such that for no configuration $\langle q', o' \rangle$, with $q' \in Q$ and $o' \in O$, there exists $a \in A$ such that $\langle q_k, o_k \rangle \xrightarrow{a} \langle q', o' \rangle$ via the above rule.

$OBS(q, o)$ is the set of all observations from the initial configuration $\langle q, o \rangle$. 
Experimentations

Successful Experiments
An observation $c \in OBS(q, o)$ is *successful* if there exists a configuration $\langle q_n, o_n \rangle \in c$, with $n \geq 0$, such that $o_n \xrightarrow{w} .$

Satisfaction of Observers

- **q MAY SATISFY o** if there exists an observation $c \in OBS(q, o)$ that is successful;
- **q MUST SATISFY o** if all observations $c \in OBS(q, o)$ are successful.

May, Must and Testing Equivalences

May Equivalence

$p$ is *may* equivalent to $q$, $p \simeq_m q$, if for all observers $o \in \mathcal{O}$ we have $p$ MAY SATISFY $o$ if and only if $q$ MAY SATISFY $o$;

Must Equivalence

$p$ is *must* equivalent to $q$, $p \simeq_M q$, if for all observers $o \in \mathcal{O}$ we have $p$ MUST SATISFY $o$ if and only if $q$ MUST SATISFY $o$.

Testing Equivalence

$p$ is *testing* equivalent to $q$, $p \simeq_{test} q$, if $p \simeq_m q$ and $p \simeq_M q$. 
Examples for may, must and testing

\[ p \cong_m q \]
\[ \text{NOT } p \cong_M q \]
\[ q \cong_M r \]
\[ q \cong_{\text{test}} r \]

Weak Equivalences

Is it right to consider different from a user point of view the three machines below, if
- grinding is an internal action?
- \( \tau \) is an invisible action?
Weak Traces Equivalence

Let \( \langle Q, A, \rightarrow \rangle \) be an LTS, with \( q \in Q \) and \( s \in A^* \) and

Let \( q \xrightarrow{s} q' \) denote that \( q \) reduces to \( q' \) by performing the sequence \( s \) of
visible actions each of which can be preceded or followed by internal
actions \( \tau \).

**Traces**

- \( s \) is a weak trace of \( q \) if there exists \( q' \in Q \) s.t. \( q \xrightarrow{s} q' \).
- \( L(q) \) represents the set of all weak traces of \( q \)

**Traces Equivalence**

Two states \( p \) and \( q \) are trace equivalent, written \( p \approx_L q \), if \( L(p) = L(p) \).

**Weak Observations**

To define the weak variants of may, must and testing equivalences it
suffices to change experiments so that processes and observers can freely
perform silent actions

Given two LTS \( \langle Q, A, \rightarrow \rangle \) and \( \langle O, A_w, \rightarrow \rangle \), and two states \( q \in Q \) e
\( o \in O \), a weak experiment \( c \) from \( \langle q, o \rangle \) is a sequence of pairs \( \langle q_i, o_i \rangle \), s.t.

- \( \langle q_0, o_0 \rangle = \langle q, o \rangle \);
- the transition \( \langle q_i, o_i \rangle \xrightarrow{a} \langle q_{i+1}, o_{i+1} \rangle \) can be proved using:

\[
\begin{align*}
E \xrightarrow{\tau} E' & \quad E \xrightarrow{\tau} E' & \quad E \xrightarrow{\tau} E', F \xrightarrow{a} F' \\
\langle E, F \rangle \xrightarrow{\tau} \langle E', F \rangle & \quad \langle E, F \rangle \xrightarrow{\tau} \langle E', F \rangle & \quad \langle E, F \rangle \xrightarrow{\tau} \langle E', F' \rangle \\
\end{align*}
\]

the last element of the sequence, say \( \langle q_k, o_k \rangle \), is such that for no
configuration \( \langle q', o' \rangle \), with \( q' \in Q \) e \( o' \in O \), there exists \( a \in A \) such
that \( \langle q_k, o_k \rangle \xrightarrow{a} \langle q', o' \rangle \) via the above rule.
Weak Bisimulation Relation: An immediate generalization

Weak Bisimulation

A relation $R \subseteq Q \times Q$ is weak bisimulation if, for any pair of states $p \in q$ such that $\langle p, q \rangle \in R$, for any $s \in A^*$, the following holds:

- for all $a \in A$ and $p' \in Q$, if $p \xrightarrow{s} p'$ then $q \xrightarrow{s} q'$ for some $q' \in Q$ such that $\langle p', q' \rangle \in R$;
- for all $a \in A$ and $q' \in Q$, if $q \xrightarrow{s} q'$ then $p \xrightarrow{s} p'$ for some $p' \in Q$ such that $\langle p', q' \rangle \in R$.

Weak Bisimilarity

Two states $p, q \in Q$ of an LTS $\langle Q, A, \rightarrow \rangle$ are weakly bisimilar, written $p \approx q$, if there exists a weak bisimulation $R$ such that $\langle p, q \rangle \in R$.

Weak Bisimulation Relation: A simpler definition

Weak Bisimulation

A relation $R \subseteq Q \times Q$ is weak bisimulation if, for any pair of states $p \in q$ such that $\langle p, q \rangle \in R$, the following holds:

- for all $a \in A$ and $p' \in Q$, if $p \xrightarrow{\mu} p'$ then $q \xrightarrow{\hat{\mu}} q'$ for some $q' \in Q$ such that $\langle p', q' \rangle \in R$;
- for all $a \in A$ and $q' \in Q$, if $q \xrightarrow{\mu} q'$ then $p \xrightarrow{\hat{\mu}} p'$ for some $p' \in Q$ such that $\langle p', q' \rangle \in R$.

where

$$
\hat{\mu} = \begin{cases} 
\epsilon & \text{se } \mu = \tau \\
\mu & \text{se } \mu \neq \tau
\end{cases}
$$

Weak Bisimilarity

Two states $p, q \in Q$ of an LTS $\langle Q, A, \rightarrow \rangle$ are weakly bisimilar, written $p \approx q$, if there exists a weak bisimulation $R$ such that $\langle p, q \rangle \in R$. 
Two Pairs of Weakly Bisimilar Systems

Ignoring Tau’s

```
  a  τ  b
  ↑    ↓    ↑
  p0   p1   p2
  τ    c  a
  b     ↓
  p3   p4
```

Ignoring Tau’s and Branching

```
  a  τ  b
  ↑    ↓    ↑
  q0   q1   q2
  τ    c  a
  b     ↓
  q3   q4
```

An Alternative to Weak Bisimulation

Branching Bisimulation

A symmetric relation $R \subseteq Q \times Q$ is weak bisimulation if, for any pair of states $p \in Q$ and $q \in Q$ such that $\langle p, q \rangle \in R$, if $p \xrightarrow{\mu} p'$, with $\mu \in A_\tau$ and $p' \in Q$, at least one of the following conditions holds:

1. $\mu = \tau \text{ e } \langle p', q \rangle \in R$
2. $q \Rightarrow q'' \xrightarrow{\mu} q'$ per qualche $q', q'' \in Q$ tali che $\langle p, q'' \rangle \in R$ e $\langle p', q' \rangle \in R$.

Branching Bisimilarity

Two states $p, q \in Q$ of an LTS $\langle Q, A_\tau, \rightarrow \rangle$ are Branching bisimilar, written $p \approx_b q$, if there exists a branching bisimulation $R$ such that $\langle p, q \rangle \in R$. 
The systems above are weakly testing equivalent but NOT weakly (nor branching) bisimilar
The systems above are NOT testing equivalent but are weakly (and branching) bisimilar.

Equivalences Hierarchies

For strongly convergent systems we have:

$\equiv_C T \equiv T$
CCS: Calculus of Communicating Processes

Milner 1980

\[ E ::= \text{nil} \mid \mu.E \mid E\setminus L \mid E[f] \mid E_1 + E_2 \mid E_1\mid E_2 \mid \text{rec}X.E \]

The set of actions \( \text{Act}_\tau \) consists of a set of labels \( \Lambda \), of the set \( \overline{\Lambda} \) of complementary labels and by the distinct action \( \tau \). Moreover we have \( \mu \in \text{Act}_\tau, L \subseteq \Lambda, f : \text{Act}_\tau \rightarrow \text{Act}_\tau \) and with \( f(\bar{\alpha}) = \overline{f(\alpha)} \) and \( f(\tau) = \tau \).

CCS has been studied with Bisimulation and Testing Semantics
SCCS: Synchronous Calculus of Communicating Processes

Milner 1983

L’insieme delle azioni $Act$ un gruppo abeliano costituito da un insieme di etichette $\Lambda$, dall’insieme $\overline{\Lambda}$ delle etichette soprasegnate e dall’elemento neutro 1, la sintassi

$$E ::= \text{nil} \mid \mu : E \mid E \uparrow L \mid E_1 + E_2 \mid E_1 \times E_2 \mid \text{rec} X . E$$

dove $\mu \in Act$, $L \subseteq \Lambda$, l’operatore $:$ denota l’action prefixing. L’operatore di rietichettatura assente perché può essere espresso per mezzo degli altri operatori.

SCCS has been studied with Bisimulation Semantics

TCSP: Theoretical Communicating Sequential Processes

Brookes-Hoare-Roscoe 1984

L’insieme delle azioni un insieme $\Lambda$, la sintassi

$$E ::= \text{stop} \mid \text{skip} \mid a \rightarrow E \mid E \setminus L \mid E[f] \mid E_1 ; E_2 \mid E_1 \sqcap E_2$$
$$\mid E_1 \sqcup E_2 \mid E_1 \parallel E_2 \mid E_1 \parallel E_2 \mid E_1 \parallel [L] E_2 \mid A$$

dove $a \in \Lambda$, $L \subseteq \Lambda$, $f : \Lambda \rightarrow \Lambda$, l’operatore $\sqcap$ denota la scelta interna, l’operatore $\rightarrow$ denota l’action prefixing e $A$ una costante di processo.

CSP has been studied with Failure Semantics - a variant of Testing Semantics
ACP: Algebra of Communicating Processes

Bergstra-Klop 1984)

L’insieme delle azioni $\Lambda_\tau$ e costituito da un insieme finito di etichette $\Lambda$ e dall’azione speciale $\tau$, la sintassi

$$E ::= \sqrt{\cdot} \mid a \mid E \setminus L \mid E/L \mid E[f] \mid E_1 \cdot E_2 \mid E_1 + E_2$$

$$\mid E_1 \parallel E_2 \mid E_1 \| E_2 \mid E_1 \varsigma E_2 \mid A$$

dove $a \in \Lambda_\tau$, $L \subseteq \Lambda$, $f : \Lambda \rightarrow \Lambda$, l’operatore $\cdot$ denota la composizione sequenziale e $A$ una costante di processo. (Le notazioni originali degli operatori $\setminus L$, $\cdot/L$ e $\cdot[f]$ sono rispettivamente $\delta_L(\cdot)$, $\tau_L(\cdot)$ e $\rho_f(\cdot)$).

ACP has been studied with Bisimulation and Branching Bisimulation Semantics

LOTOS: Language of Temporal Order Specification

Standard ISO 1988

L’insieme delle azioni $\Lambda_i$ e costituito da un insieme finito di etichette $\Lambda$ e dall’azione speciale $i$, la sintassi

$$E ::= \text{stop} \mid \text{exit} \mid \mu; E \mid E/L \mid E[f] \mid E_1 \gg E_2 \mid E_1 [> E_2$$

$$\mid E_1 + E_2 \mid E_1 \parallel E_2 \mid E_1 \| E_2 \mid E_1 \| [L] E_2 \mid A$$

dove $\mu \in \Lambda_i$, $L \subseteq \Lambda$, $f : \Lambda \rightarrow \Lambda$, l’operatore $\cdot$ denota l’action prefixing, l’operatore $\gg$ denota la composizione sequenziale e $A$ una costante di processo.

LOTOS has been studied with Bisimulation and Testing Semantics