The LTS WorkBench

Alceste Scalas   Massimo Bartoletti

University of Cagliari
Dept. of Mathematics and Informatics

ICE — Grenoble, June 5th, 2015

Motivation

You are working on **LTS-based** models for concurrency and **observational relations**

You want to **validate** some theory, on LTS generated by **different calculi**, or maybe just **explore the transitions** of a process

Motivation

You are working on **LTS-based** models for concurrency and **observational relations**

You want to **validate** some theory, on LTS generated by **different calculi**, or maybe just **explore the transitions** of a process

You need to:

- represent **LTSs and processes**
- **manipulate** them (**compose** them, let them **synchronise**, **filter out** some parts,...)
- compute and check **relations** between their states

Motivation

You are working on **LTS-based** models for concurrency and **observational relations**

You want to **validate** some theory, on LTS generated by **different calculi**, or maybe just **explore the transitions** of a process

You need to:

- represent **LTSs and processes**
- **manipulate** them (**compose** them, let them **synchronise**, **filter out** some parts,. . .)
- compute and check **relations** between their states

How do you proceed?

Motivation

You are working on **LTS-based** models for concurrency and **observational relations**

You want to **validate** some theory, on LTS generated by **different calculi**, or maybe just **explore the transitions** of a process

You need to:

- represent **LTSs and processes**
- **manipulate** them (**compose** them, let them **synchronise**, **filter out** some parts,. . .)
- compute and check **relations** between their states

How do you proceed?

- you try to **encode your theory** in the process/logic language supported by some tool (e.g. mCRL2, CADP)

Motivation

You are working on **LTS-based** models for concurrency and **observational relations**

You want to **validate** some theory, on LTS generated by **different calculi**, or maybe just **explore the transitions** of a process

You need to:

- represent **LTSs and processes**
- **manipulate** them (**compose** them, let them **synchronise**, **filter out** some parts, . . .)
- compute and check **relations** between their states

How do you proceed?

- you try to **encode your theory** in the process/logic language supported by some tool (e.g. mCRL2, CADP)
- **otherwise**, you implement it **directly**

Motivation (cont'd)

What if you are dealing with **(possibly) infinite-state** LTSs and processes, arising e.g. from **recursion, parallelism, unbounded buffers**?

Motivation (cont'd)

What if you are dealing with **(possibly) infinite-state** LTSs and processes, arising e.g. from **recursion, parallelism, unbounded buffers**?

The encoding **(if possible)** may be **cumbersome**, and then a direct implementation becomes more appealing

Motivation (cont'd)

What if you are dealing with **(possibly) infinite-state** LTSs and processes, arising e.g. from **recursion, parallelism, unbounded buffers**?

The encoding **(if possible)** may be **cumbersome**, and then a direct implementation becomes more appealing

We were in this situation, but wanted to **avoid yet another ad-hoc implementation**

- we ended up with `LTSwb`

A reusable semantic framework

Observation: different process calculi have **similar operators**

- sequential execution
- choice
- parallel composition
- synchronisation
- ...

...and several commonalities with **semantic models** (e.g. CFSMs)

A reusable semantic framework

Observation: different process calculi have **similar operators**

- sequential execution
- choice
- parallel composition
- synchronisation
- ...

...and several commonalities with **semantic models** (e.g. CFSMs)

Idea:

1. define **as many operators as possible** at a **semantic, syntax-independent level**
2. mix & match to **cook your process calculus — if** needed!

Example: a calculus with sequencing

We want to implement and study a process calculus C
with the usual **sequential composition** $(\mathtt{p}\ \mathtt{seq}\ \mathtt{q})$

$$\frac{\mathtt{p} \xrightarrow{\ell} \mathtt{p}'}{(\mathtt{p}\ \mathtt{seq}\ \mathtt{q}) \xrightarrow{\ell} (\mathtt{p}'\ \mathtt{seq}\ \mathtt{q})} \qquad \frac{\mathtt{p} \not\rightarrow \quad \mathtt{q} \xrightarrow{\ell} \mathtt{q}'}{(\mathtt{p}\ \mathtt{seq}\ \mathtt{q}) \xrightarrow{\ell} (\mathtt{p}\ \mathtt{seq}\ \mathtt{q}')}$$

Example: a calculus with sequencing

We want to implement and study a process calculus C
with the usual **sequential composition** $(\mathtt{p\ seq\ q})$

$$\frac{\mathtt{p} \xrightarrow{\ell} \mathtt{p}'}{(\mathtt{p\ seq\ q}) \xrightarrow{\ell} (\mathtt{p}'\ \mathtt{seq\ q})} \qquad \frac{\mathtt{p} \not\rightarrow \quad \mathtt{q} \xrightarrow{\ell} \mathtt{q}'}{(\mathtt{p\ seq\ q}) \xrightarrow{\ell} (\mathtt{p\ seq}\ \mathtt{q}')}$$

This looks **independent from the syntax of** $\mathtt{p}$**,** $\mathtt{q}$ **and** ℓ

Example: a calculus with sequencing

We want to implement and study a process calculus C
with the usual **sequential composition** $(\mathtt{p\ seq\ q})$

$$\frac{\mathtt{p} \xrightarrow{\ell} \mathtt{p}'}{(\mathtt{p\ seq\ q}) \xrightarrow{\ell} (\mathtt{p'\ seq\ q})} \qquad \frac{\mathtt{p} \not\rightarrow \quad \mathtt{q} \xrightarrow{\ell} \mathtt{q}'}{(\mathtt{p\ seq\ q}) \xrightarrow{\ell} (\mathtt{p\ seq\ q'})}$$

This looks **independent from the syntax of** $\mathtt{p}$**,** $\mathtt{q}$ **and** ℓ

- what if $\mathtt{p}$,$\mathtt{q}$ come e.g. from **execution logs**?

Example: a calculus with sequencing

We want to implement and study a process calculus C with the usual **sequential composition** $(\mathtt{p}\ \ \mathtt{seq}\ \ \mathtt{q})$

$$\frac{\mathtt{p} \xrightarrow{\ell} \mathtt{p}'}{(\mathtt{p}\ \ \mathtt{seq}\ \ \mathtt{q}) \xrightarrow{\ell} (\mathtt{p}'\ \ \mathtt{seq}\ \ \mathtt{q})} \qquad \frac{\mathtt{p} \not\rightarrow \quad \mathtt{q} \xrightarrow{\ell} \mathtt{q}'}{(\mathtt{p}\ \ \mathtt{seq}\ \ \mathtt{q}) \xrightarrow{\ell} (\mathtt{p}\ \ \mathtt{seq}\ \ \mathtt{q}')}$$

This looks **independent from the syntax of** $\mathtt{p}$**,** $\mathtt{q}$ **and** ℓ

- what if $\mathtt{p}$,$\mathtt{q}$ come e.g. from **execution logs**?

Can we implement such a composition upon a **reusable syntax-independent** foundation?

Definitions

An **LTS** is a triple $(\Sigma, \Lambda, \mathcal{R})$ where:

- $\Sigma = \{p, q, r, \ldots\}$ is the set of **states**
- $\Lambda = \{\ell_1, \ell_2, \ldots\}$ is the set of **labels**
- $\mathcal{R} \subseteq \big(\Sigma \times (\Lambda \times \Sigma)\big)$ is the **transition relation**

Definitions

An **LTS** is a triple $(\Sigma, \Lambda, \mathcal{R})$ where:

- $\Sigma = \{p, q, r, \ldots\}$ is the set of **states**
- $\Lambda = \{\ell_1, \ell_2, \ldots\}$ is the set of **labels**
- $\mathcal{R} \subseteq \big(\Sigma \times (\Lambda \times \Sigma)\big)$ is the **transition relation**

A **process** is a pair $(\mathbb{L}, p)$
where $\mathbb{L}$ is an LTS and p is one of its states

Definitions

An **LTS** is a triple $(\Sigma, \Lambda, \mathcal{R})$ where:

- $\Sigma = \{p, q, r, \ldots\}$ is the set of **states**
- $\Lambda = \{\ell_1, \ell_2, \ldots\}$ is the set of **labels**
- $\mathcal{R} \subseteq \big(\Sigma \times (\Lambda \times \Sigma)\big)$ is the **transition relation**

A **process** is a pair $(\mathbb{L}, p)$
where $\mathbb{L}$ is an LTS and p is one of its states

The **process transition** $(\mathbb{L}, p) \xrightarrow{\ell} (\mathbb{L}, p')$ holds
iff $\big(p, (\ell, p')\big)$ is in the transition relation of $\mathbb{L}$

Definitions

An **LTS** is a triple $(\Sigma, \Lambda, \mathcal{R})$ where:

- $\Sigma = \{p, q, r, \ldots\}$ is the set of **states**
- $\Lambda = \{\ell_1, \ell_2, \ldots\}$ is the set of **labels**
- $\mathcal{R} \subseteq (\Sigma \times (\Lambda \times \Sigma))$ is the **transition relation**

A **process** is a pair $(\mathbb{L}, p)$
where $\mathbb{L}$ is an LTS and p is one of its states

The **process transition** $(\mathbb{L}, p) \xrightarrow{\ell} (\mathbb{L}, p')$ holds
iff $(p, (\ell, p'))$ is in the transition relation of $\mathbb{L}$

Let $\mathcal{R} \subseteq \Delta \times \Gamma$. Then, $\mathcal{R}(\delta) \coloneqq \{\gamma \mid (\delta, \gamma) \in \mathcal{R}\}$

Definitions

An **LTS** is a triple $(\Sigma, \Lambda, \mathcal{R})$ where:

- $\Sigma = \{p, q, r, \ldots\}$ is the set of **states**
- $\Lambda = \{\ell_1, \ell_2, \ldots\}$ is the set of **labels**
- $\mathcal{R} \subseteq (\Sigma \times (\Lambda \times \Sigma))$ is the **transition relation**

A **process** is a pair $(\mathbb{L}, p)$
where $\mathbb{L}$ is an LTS and p is one of its states

The **process transition** $(\mathbb{L}, p) \xrightarrow{\ell} (\mathbb{L}, p')$ holds
iff $(p, (\ell, p'))$ is in the transition relation of $\mathbb{L}$

Let $\mathcal{R} \subseteq \Delta \times \Gamma$. Then, $\mathcal{R}(\delta) \coloneqq \{\gamma \mid (\delta, \gamma) \in \mathcal{R}\}$

$$(\mathbb{L}, p)(\ell) \coloneqq \left\{ (\mathbb{L}, p') \;\middle|\; (\mathbb{L}, p) \xrightarrow{\ell} (\mathbb{L}, p') \right\}$$

LTS operators: a (boring) example

The **union of LTSs** $\mathbb{L}_1 = (\Sigma_1, \Lambda_1, \mathcal{R}_1)$ and $\mathbb{L}_2 = (\Sigma_2, \Lambda_2, \mathcal{R}_2)$ is:

$$\mathbb{L}_1 \cup \mathbb{L}_2 \;\coloneqq\; \Big(\Sigma_1 \cup \Sigma_2,\;\; \Lambda_1 \cup \Lambda_2,\;\; \mathcal{R}_1 \cup \mathcal{R}_2 \Big)$$

Sequencing of relations

Let $\mathcal{R}_1 \subseteq \big(\Sigma_1 \times (\Lambda_1 \times \Sigma_1')\big)$ and $\mathcal{R}_2 \subseteq \big(\Sigma_2 \times (\Lambda_2 \times \Sigma_2')\big)$

The **sequencing of** $\mathcal{R}_1$ **and** $\mathcal{R}_2$ is the relation

$$\mathcal{R}_1 \,;\mathcal{R}_2 \subseteq \Big((\Sigma_1 \times \Sigma_2) \times \big((\Lambda_1 \cup \Lambda_2) \times (\Sigma_1' \times \Sigma_2')\big)\Big)$$

inductively defined by the rules:

$$\frac{\big(p, (\ell, p')\big) \in \mathcal{R}_1}{\Big((p, q),\ \big(\ell, (p', q)\big)\Big) \in \mathcal{R}_1 \,;\mathcal{R}_2} \qquad \frac{\mathcal{R}_1(p) = \emptyset \quad \big((q, (\ell, q')\big) \in \mathcal{R}_2}{\Big((p, q),\ \big(\ell, (p, q')\big)\Big) \in \mathcal{R}_1 \,;\mathcal{R}_2}$$

Sequencing of relations

Let $\mathcal{R}_1 \subseteq \big(\Sigma_1 \times (\Lambda_1 \times \Sigma_1')\big)$ and $\mathcal{R}_2 \subseteq \big(\Sigma_2 \times (\Lambda_2 \times \Sigma_2')\big)$

The **sequencing of** $\mathcal{R}_1$ **and** $\mathcal{R}_2$ is the relation

$$\mathcal{R}_1 ; \mathcal{R}_2 \subseteq \Big((\Sigma_1 \times \Sigma_2) \times \big((\Lambda_1 \cup \Lambda_2) \times (\Sigma_1' \times \Sigma_2')\big)\Big)$$

inductively defined by the rules:

$$\frac{\big(p, (\ell, p')\big) \in \mathcal{R}_1}{\Big((p, q), \big(\ell, (p', q)\big)\Big) \in \mathcal{R}_1 ; \mathcal{R}_2} \qquad \frac{\mathcal{R}_1(p) = \emptyset \quad \big((q, (\ell, q')\big) \in \mathcal{R}_2}{\Big((p, q), \big(\ell, (p, q')\big)\Big) \in \mathcal{R}_1 ; \mathcal{R}_2}$$

Equivalently:

$$\Big(\mathcal{R}_1 ; \mathcal{R}_2\Big)\big((p, q)\big) = \begin{cases} \big\{\big(\ell, (p', q)\big) \mid \big(\ell, p'\big) \in \mathcal{R}_1(p)\big\} & \text{if } \mathcal{R}_1(p) \neq \emptyset \\ \big\{\big(\ell, (p, q')\big) \mid \big(\ell, q'\big) \in \mathcal{R}_2(q)\big\} & \text{otherwise} \end{cases}$$

Sequencing of LTSs and processes

Let $\mathbb{L}_1 = (\Sigma_1, \Lambda_1, \mathcal{R}_1)$ and $\mathbb{L}_2 = (\Sigma_2, \Lambda_2, \mathcal{R}_2)$

The **sequencing of** $\mathbb{L}_1$ **and** $\mathbb{L}_2$ is:

$$\mathbb{L}_1 ; \mathbb{L}_2 \ \coloneqq\ \left(\Sigma_1 \times \Sigma_2,\ \ \Lambda_1 \cup \Lambda_2,\ \ \mathcal{R}_1 ; \mathcal{R}_2 \right)$$

Sequencing of LTSs and processes

Let $\mathbb{L}_1 = (\Sigma_1, \Lambda_1, \mathcal{R}_1)$ and $\mathbb{L}_2 = (\Sigma_2, \Lambda_2, \mathcal{R}_2)$

The **sequencing of** $\mathbb{L}_1$ **and** $\mathbb{L}_2$ is:

$$\mathbb{L}_1 ; \mathbb{L}_2 \;\coloneqq\; \left(\Sigma_1 \times \Sigma_2,\;\; \Lambda_1 \cup \Lambda_2,\;\; \mathcal{R}_1 ; \mathcal{R}_2 \right)$$

The **sequencing of processes** $(\mathbb{L}_1, p)$ **and** $(\mathbb{L}_2, q)$ is:

$$(\mathbb{L}_1, p) \;;(\mathbb{L}_2, q) \;\coloneqq\; \Big(\mathbb{L}_1 ; \mathbb{L}_2 \,,\; (p, q)\Big)$$

Sequencing of LTSs and processes

Let $\mathbb{L}_1 = (\Sigma_1, \Lambda_1, \mathcal{R}_1)$ and $\mathbb{L}_2 = (\Sigma_2, \Lambda_2, \mathcal{R}_2)$

The **sequencing of $\mathbb{L}_1$ and $\mathbb{L}_2$** is:

$$\mathbb{L}_1 ; \mathbb{L}_2 \quad \coloneqq \quad \Big(\Sigma_1 \times \Sigma_2, \;\; \Lambda_1 \cup \Lambda_2, \;\; \mathcal{R}_1 ; \mathcal{R}_2 \Big)$$

The **sequencing of processes** $(\mathbb{L}_1, \mathrm{p})$ **and** $(\mathbb{L}_2, \mathrm{q})$ is:

$$(\mathbb{L}_1, \mathrm{p}) \; ; (\mathbb{L}_2, \mathrm{q}) \quad \coloneqq \quad \Big(\mathbb{L}_1 ; \mathbb{L}_2 \, , \, (\mathrm{p}, \mathrm{q}) \Big)$$

I.e., $(\mathbb{L}_1, \mathrm{p}) \; ; (\mathbb{L}_2, \mathrm{q})$ **observationally behaves** as p in $\mathbb{L}_1$, *and then* as q in $\mathbb{L}_2$

From semantic to syntactic sequencing

Back to our calculus C, with sequential composition `(p seq q)`

From semantic to syntactic sequencing

Back to our calculus C, with sequential composition `(p seq q)`

Its LTS is $\mathbb{L}_C = (\Sigma_C, \Lambda_C, \mathcal{R}_C)$

Desideratum: $\left(\mathbb{L}_C, \texttt{(p seq q)}\right) \cong \left(\mathbb{L}_C ; \mathbb{L}_C, (\mathtt{p}, \mathtt{q})\right)$

From semantic to syntactic sequencing

Back to our calculus C, with sequential composition $\texttt{(p seq q)}$

Its LTS is $\mathbb{L}_C = (\Sigma_C, \Lambda_C, \mathcal{R}_C)$

Desideratum: $\Big(\mathbb{L}_C, \texttt{(p seq q)}\Big) \cong \Big(\mathbb{L}_C \,;\, \mathbb{L}_C, (\mathrm{p}, \mathrm{q})\Big)$

We can define the LTS $\mathbb{L}_C$ so that:

$$\Big(\mathbb{L}_C, \texttt{(p seq q)}\Big)(\ell) = \Big\{\Big(\mathbb{L}_C, \texttt{(p}' \texttt{ seq q}'\texttt{)}\Big) \;\Big|\; (\mathrm{p}', \mathrm{q}') \in \Big(\mathbb{L}_C \,;\, \mathbb{L}_C, (\mathrm{p}, \mathrm{q})\Big)(\ell)\Big\}$$

From semantic to syntactic sequencing

Back to our calculus C, with sequential composition $\texttt{(p seq q)}$

Its LTS is $\mathbb{L}_C = (\Sigma_C, \Lambda_C, \mathcal{R}_C)$

Desideratum: $\left(\mathbb{L}_C, \texttt{(p seq q)}\right) \cong \left(\mathbb{L}_C \,;\, \mathbb{L}_C, (p, q)\right)$

We can define the LTS $\mathbb{L}_C$ so that:

$$\left(\mathbb{L}_C, \texttt{(p seq q)}\right)(\ell) = \left\{\left(\mathbb{L}_C, \texttt{(p}' \texttt{ seq q}'\texttt{)}\right) \,\middle|\, (p', q') \in \left(\mathbb{L}_C \,;\, \mathbb{L}_C, (p, q)\right)(\ell)\right\}$$

Which means:

$$\mathcal{R}_C(\texttt{(p seq q)}) = \left\{\left(\ell, \texttt{(p}' \texttt{ seq q}'\texttt{)}\right) \,\middle|\, (\ell, (p', q')) \in \left(\mathcal{R}_C \,;\, \mathcal{R}_C\right)((p, q))\right\}$$

Summing up

Syntactic sequencing

$\Uparrow$

Process sequencing

$\Uparrow$

LTS sequencing

$\Uparrow$

Relational sequencing

Summing up

Syntactic **operator**

$\Uparrow$

Process **operator**

$\Uparrow$

LTS **operator**

$\Uparrow$

Relational **operator**

Summing up

Syntactic **operator**

$\Uparrow$

Process **operator**

$\Uparrow$

LTS **operator**

$\Uparrow$

Relational **operator**

where "operator" may be
sequencing, **parallel composition**, **state/label filtering**, ...

Summing up

Syntactic **operator**

$\Uparrow$

Process **operator**

$\Uparrow$

LTS **operator**

$\Uparrow$

Relational **operator**

where “operator” may be
sequencing, **parallel composition**, **state/label filtering**, . . .

. . . and this is **how our tool works**

Introducing LTSwb

`LTSwb` is a **Labelled Transition System (LTS) toolbox**, allowing to **define LTSs and processes**, **manipulate** them, and **compute relations** between their states

Introducing `LTSwb`

`LTSwb` is a **Labelled Transition System (LTS) toolbox**, allowing to **define LTSs and processes**, **manipulate** them, and **compute relations** between their states

`LTSwb` is a **Scala** library, usable from the Scala REPL

Introducing `LTSwb`

`LTSwb` is a **Labelled Transition System (LTS) toolbox**, allowing to **define LTSs and processes**, **manipulate** them, and **compute relations** between their states

`LTSwb` is a **Scala** library, usable from the Scala REPL

Why Scala?

- advanced **type system**
- access to **JVM libraries**
- **eager** language with `lazy val`ues

Introducing `LTSwb`

`LTSwb` is a **Labelled Transition System (LTS) toolbox**, allowing to **define LTSs and processes**, **manipulate** them, and **compute relations** between their states

`LTSwb` is a **Scala** library, usable from the Scala REPL

Why Scala?

- advanced **type system**
- access to **JVM libraries**
- **eager** language with `lazy val`ues

`LTSwb` **features**:

- **purely semantic**: no privileged language for processes
- **generic**: parametric on state/label types and synchronisation
- **lazy**: only generates states and transitions when needed

Internals

- `Set[A]` with `.contains(x)`
- `Relation[A,B]` with `.apply(x:A): Set[B]`
- `Relation3[A,B,C]` with `.apply(x:A): Relation[B,C]`
- `LTS[A,B]` with `.process(s:A): Process[A,B]`
- `Process[A,B]` with `.apply(l:B): Set[Process[A,B]]`

Internals

- `Set[A]` with `.contains(x)`
  - `FiniteSet[A]` with `.iterator()`
- `Relation[A,B]` with `.apply(x:A): Set[B]`
- `Relation3[A,B,C]` with `.apply(x:A): Relation[B,C]`
- `LTS[A,B]` with `.process(s:A): Process[A,B]`
- `Process[A,B]` with `.apply(l:B): Set[Process[A,B]]`

Internals

- `Set[A]` with `.contains(x)`
  - `FiniteSet[A]` with `.iterator()`
- `Relation[A,B]` with `.apply(x:A): Set[B]`
  - `FiniteImageRelation[A,B]`, `FiniteRelation[A,B]`
- `Relation3[A,B,C]` with `.apply(x:A): Relation[B,C]`

- `LTS[A,B]` with `.process(s:A): Process[A,B]`

- `Process[A,B]` with `.apply(l:B): Set[Process[A,B]]`

Internals

- `Set[A]` with `.contains(x)`
  - `FiniteSet[A]` with `.iterator()`
- `Relation[A,B]` with `.apply(x:A): Set[B]`
  - `FiniteImageRelation[A,B]`, `FiniteRelation[A,B]`
- `Relation3[A,B,C]` with `.apply(x:A): Relation[B,C]`
  - `FiniteBranchingRelation3[A,B,C]`, `FiniteRelation3[A,B,C]`
- `LTS[A,B]` with `.process(s:A): Process[A,B]`
- `Process[A,B]` with `.apply(l:B): Set[Process[A,B]]`

Internals

- Set[A] with .contains(x)
  - FiniteSet[A] with .iterator()
- Relation[A,B] with .apply(x:A): Set[B]
  - FiniteImageRelation[A,B], FiniteRelation[A,B]
- Relation3[A,B,C] with .apply(x:A): Relation[B,C]
  - FiniteBranchingRelation3[A,B,C], FiniteRelation3[A,B,C]
- LTS[A,B] with .process(s:A): Process[A,B]
  - FiniteBranchingLTS[A,B], FiniteLTS[A,B]
- Process[A,B] with .apply(l:B): Set[Process[A,B]]

Internals

- `Set[A]` with `.contains(x)`
  - `FiniteSet[A]` with `.iterator()`
- `Relation[A,B]` with `.apply(x:A): Set[B]`
  - `FiniteImageRelation[A,B]`, `FiniteRelation[A,B]`
- `Relation3[A,B,C]` with `.apply(x:A): Relation[B,C]`
  - `FiniteBranchingRelation3[A,B,C]`, `FiniteRelation3[A,B,C]`
- `LTS[A,B]` with `.process(s:A): Process[A,B]`
  - `FiniteBranchingLTS[A,B]`, `FiniteLTS[A,B]`
- `Process[A,B]` with `.apply(l:B): Set[Process[A,B]]`
  - `FiniteBranchingProcess[A,B]`, `FiniteProcess[A,B]`

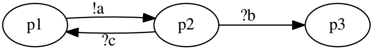
Defining LTSs (and processes)

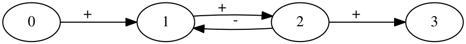
```
val l1 = LTS(List((0, ("+", 1)), (1, ("+", 2)),
                  (2, ("+", 3)), (2, ("-", 1))))
val l2 = LTS(List(("p1", ("!a", "p2")),
                  ("p2", ("?b", "p3")),
                  ("p2", ("?c", "p1"))))
```

Defining LTSs (and processes)

```
val l1 = LTS(List((0, ("+", 1)), (1, ("+", 2)),
                  (2, ("+", 3)), (2, ("-", 1))))
val l2 = LTS(List(("p1", ("!a", "p2")),
                  ("p2", ("?b", "p3")),
                  ("p2", ("?c", "p1"))))
```

`l1.doDot` and `l2.toDot` are:

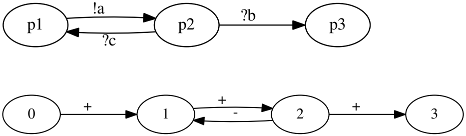




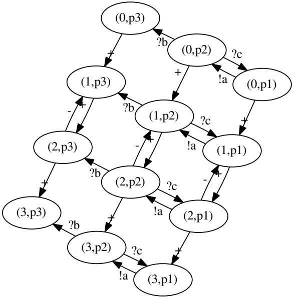
Defining LTSs (and processes)

```
val l1 = LTS(List((0, ("+", 1)), (1, ("+", 2)),
                  (2, ("+", 3)), (2, ("-", 1))))
val l2 = LTS(List(("p1", ("!a", "p2")),
                  ("p2", ("?b", "p3")),
                  ("p2", ("?c", "p1"))))
```

`l1.doDot` and `l2.toDot` are:



`(l1 ||| l2).toDot` is:

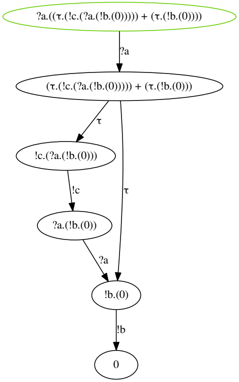


CCS processes

```
// Parses the CCSTerm from String
val ccs1 = CCS.process("rec(X)(!a.(?b + ?c.X))")
// Shorthand. "t" is the internal action
val ccs2 = CCS("?a.(t.!c.?a.!b + t.!b)")
```

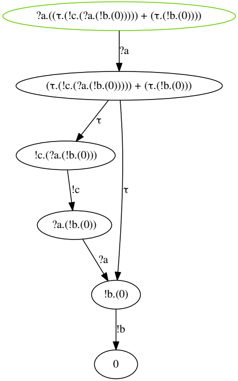
CCS processes

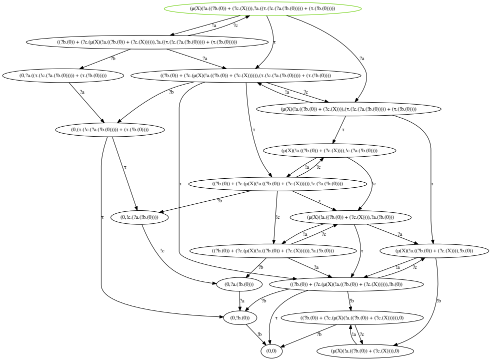
```
// Parses the CCSTerm from String
val ccs1 = CCS.process("rec(X)(!a.(?b + ?c.X))")
// Shorthand. "t" is the internal action
val ccs2 = CCS("?a.(t.!c.?a.!b + t.!b)")
```



CCS processes

```
// Parses the CCSTerm from String
val ccs1 = CCS.process("rec(X)(!a.(?b + ?c.X))")
// Shorthand. "t" is the internal action
val ccs2 = CCS("?a.(t.!c.?a.!b + t.!b)")
```





The CCS semantics

```
object CCSSemantics
extends FiniteBranchingRelation3[CCSTerm, CCSPfx, CCSTerm] {
  override def apply(s: CCSTerm) = s match {
    case CCSNil() => EmptyRelation()
    case CCSSeq(prefix, cont) => Relation(List((prefix, cont)))
    case CCSPlus(term1, term2) => this(term1) | this(term2)
    case CCSPar(term1, term2) => {
      (CCS ||| CCS).relation((term1, term2)).iso(
        (t:Tuple2[CCSTerm,CCSTerm]) => CCSPar(t._1, t._2),
        (t:CCSPar) => (t.term1, t.term2) )
    }
    case CCSRec(_,_) => this(s.unfold)
    case CCSVar(_) => EmptyRelation() // Free rec variable
    case CCSDel(n, b) => {
      CCS.del(CCSInPfx(n)).del(CCSOutPfx(n)).relation(b).iso(
        (t:CCSTerm) => CCSDel(n, t),  (t:CCSDel) => t.body )
    }
  }
}
```

Synchronous vs. asynchronous semantics

We often work on **session types**, with two semantics:

Synchronous vs. asynchronous semantics

We often work on **session types**, with two semantics:

- **synchronous**:

  (De Nicola and Hennessy, 1987; Barbanera and de' Liguoro, 2010)

  $!a.?c \oplus !b$

Synchronous vs. asynchronous semantics

We often work on **session types**, with two semantics:

- **synchronous**:

  (De Nicola and Hennessy, 1987; Barbanera and de' Liguoro, 2010)

  $!a.?c \oplus !b \xrightarrow{\tau} !a.?c$

Synchronous vs. asynchronous semantics

We often work on **session types**, with two semantics:

- **synchronous**:

  (De Nicola and Hennessy, 1987; Barbanera and de' Liguoro, 2010)

  $!a.?c \oplus !b \xrightarrow{\tau} !a.?c \xrightarrow{!a} ?c$

Synchronous vs. asynchronous semantics

We often work on **session types**, with two semantics:

- **synchronous**:

  (De Nicola and Hennessy, 1987; Barbanera and de' Liguoro, 2010)

  $\mathsf{!a}.\mathsf{?c} \oplus \mathsf{!b} \xrightarrow{\tau} \mathsf{!a}.\mathsf{?c} \xrightarrow{\mathsf{!a}} \mathsf{?c} \xrightarrow{\mathsf{?c}} \mathbf{0}$

Synchronous vs. asynchronous semantics

We often work on **session types**, with two semantics:

- **synchronous**:

  (De Nicola and Hennessy, 1987; Barbanera and de' Liguoro, 2010)

  $!a.?c \oplus !b \xrightarrow{\tau} !a.?c \xrightarrow{!a} ?c \xrightarrow{?c} \mathbf{0}$

- **asynchronous**, with an **unbounded output queue**:

  (Neubauer *et al.*, 2004; Mostrous *et al.*, 2009; ...)

  $!a.?c \oplus !b[]$

Synchronous vs. asynchronous semantics

We often work on **session types**, with two semantics:

- **synchronous**:

  (De Nicola and Hennessy, 1987; Barbanera and de' Liguoro, 2010)

  $!a.?c \oplus !b \xrightarrow{\tau} !a.?c \xrightarrow{!a} ?c \xrightarrow{?c} \mathbf{0}$

- **asynchronous**, with an **unbounded output queue**:

  (Neubauer *et al.*, 2004; Mostrous *et al.*, 2009; ...)

  $!a.?c \oplus !b[] \xrightarrow{\tau} !a.?c[]$

Synchronous vs. asynchronous semantics

We often work on **session types**, with two semantics:

- **synchronous**:

  (De Nicola and Hennessy, 1987; Barbanera and de' Liguoro, 2010)

  $!a.?c \oplus !b \xrightarrow{\tau} !a.?c \xrightarrow{!a} ?c \xrightarrow{?c} \mathbf{0}$

- **asynchronous**, with an **unbounded output queue**:

  (Neubauer *et al.*, 2004; Mostrous *et al.*, 2009; . . .)

  $!a.?c \oplus !b[] \xrightarrow{\tau} !a.?c[] \xrightarrow{\tau} ?c[!a]$

Synchronous vs. asynchronous semantics

We often work on **session types**, with two semantics:

- **synchronous**:

  (De Nicola and Hennessy, 1987; Barbanera and de' Liguoro, 2010)

  $!a.?c \oplus !b \xrightarrow{\tau} !a.?c \xrightarrow{!a} ?c \xrightarrow{?c} \mathbf{0}$

- **asynchronous**, with an **unbounded output queue**:

  (Neubauer *et al.*, 2004; Mostrous *et al.*, 2009; . . .)

  $!a.?c \oplus !b[] \xrightarrow{\tau} !a.?c[] \xrightarrow{\tau} ?c[!a] \Bigg\{$

Synchronous vs. asynchronous semantics

We often work on **session types**, with two semantics:

- **synchronous**:

  (De Nicola and Hennessy, 1987; Barbanera and de' Liguoro, 2010)

  $!a.?c \oplus !b \xrightarrow{\tau} !a.?c \xrightarrow{!a} ?c \xrightarrow{?c} \mathbf{0}$

- **asynchronous**, with an **unbounded output queue**:

  (Neubauer *et al.*, 2004; Mostrous *et al.*, 2009; ...)

  $!a.?c \oplus !b[] \xrightarrow{\tau} !a.?c[] \xrightarrow{\tau} ?c[!a] \begin{cases} \xrightarrow{!a} ?c[] \\ \end{cases}$

Synchronous vs. asynchronous semantics

We often work on **session types**, with two semantics:

- **synchronous**:

  (De Nicola and Hennessy, 1987; Barbanera and de' Liguoro, 2010)

  $!a.?c \oplus !b \xrightarrow{\tau} !a.?c \xrightarrow{!a} ?c \xrightarrow{?c} \mathbf{0}$

- **asynchronous**, with an **unbounded output queue**:

  (Neubauer *et al.*, 2004; Mostrous *et al.*, 2009; . . .)

  $!a.?c \oplus !b[] \xrightarrow{\tau} !a.?c[] \xrightarrow{\tau} ?c[!a] \left\{ \begin{array}{l} \xrightarrow{!a} ?c[] \xrightarrow{?c} \mathbf{0}[] \\ \\ \end{array} \right.$

Synchronous vs. asynchronous semantics

We often work on **session types**, with two semantics:

- **synchronous**:

  (De Nicola and Hennessy, 1987; Barbanera and de' Liguoro, 2010)

  $$!a.?c \oplus !b \xrightarrow{\tau} !a.?c \xrightarrow{!a} ?c \xrightarrow{?c} \mathbf{0}$$

- **asynchronous**, with an **unbounded output queue**:

  (Neubauer *et al.*, 2004; Mostrous *et al.*, 2009; ...)

  $$!a.?c \oplus !b[] \xrightarrow{\tau} !a.?c[] \xrightarrow{\tau} ?c[!a] \begin{cases} \xrightarrow{!a} ?c[] \xrightarrow{?c} \mathbf{0}[] \\ \xrightarrow{?c} \mathbf{0}[!a] \end{cases}$$

Synchronous vs. asynchronous semantics

We often work on **session types**, with two semantics:

- **synchronous**:

  (De Nicola and Hennessy, 1987; Barbanera and de' Liguoro, 2010)

  $!a.?c \oplus !b \xrightarrow{\tau} !a.?c \xrightarrow{!a} ?c \xrightarrow{?c} \mathbf{0}$

- **asynchronous**, with an **unbounded output queue**:

  (Neubauer *et al.*, 2004; Mostrous *et al.*, 2009; . . .)

  $$!a.?c \oplus !b[] \xrightarrow{\tau} !a.?c[] \xrightarrow{\tau} ?c[!a] \begin{cases} \xrightarrow{!a} ?c[] \xrightarrow{?c} \mathbf{0}[] \\ \xrightarrow{?c} \mathbf{0}[!a] \xrightarrow{!a} \mathbf{0}[] \end{cases}$$

Synchronous vs. asynchronous semantics

We often work on **session types**, with two semantics:

- **synchronous**:

  (De Nicola and Hennessy, 1987; Barbanera and de' Liguoro, 2010)

  $!a.?c \oplus !b \xrightarrow{\tau} !a.?c \xrightarrow{!a} ?c \xrightarrow{?c} \mathbf{0}$

- **asynchronous**, with an **unbounded output queue**:

  (Neubauer *et al.*, 2004; Mostrous *et al.*, 2009; ...)

  $$!a.?c \oplus !b[] \xrightarrow{\tau} !a.?c[] \xrightarrow{\tau} ?c[!a] \begin{cases} \xrightarrow{!a} ?c[] \xrightarrow{?c} \mathbf{0}[] \\ \xrightarrow{?c} \mathbf{0}[!a] \xrightarrow{!a} \mathbf{0}[] \end{cases}$$

... and also with **(async) CCS**, e.g.:

$?a \mid rec_X\big(?b.(\tau.!c + ?d.X)\big)[!e.!d]$

Synchronous vs. asynchronous semantics

We often work on **session types**, with two semantics:

- **synchronous**:

  (De Nicola and Hennessy, 1987; Barbanera and de' Liguoro, 2010)

  $!a.?c \oplus !b \xrightarrow{\tau} !a.?c \xrightarrow{!a} ?c \xrightarrow{?c} \mathbf{0}$

- **asynchronous**, with an **unbounded output queue**:

  (Neubauer *et al.*, 2004; Mostrous *et al.*, 2009; . . .)

  $$!a.?c \oplus !b[] \xrightarrow{\tau} !a.?c[] \xrightarrow{\tau} ?c[!a] \begin{cases} \xrightarrow{!a} ?c[] \xrightarrow{?c} \mathbf{0}[] \\ \xrightarrow{?c} \mathbf{0}[!a] \xrightarrow{!a} \mathbf{0}[] \end{cases}$$

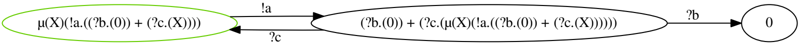
. . . and also with **(async) CCS**, e.g.:

$?a \mid rec_X\Big(?b.\big(\tau.!c + ?d.X\big)\Big)[!e.!d]$

Can we **generalise** such an **"async transformation"**?

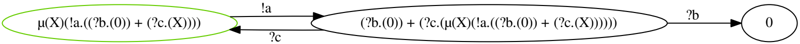
Semantic asynchrony

`ccs1.toDot()`

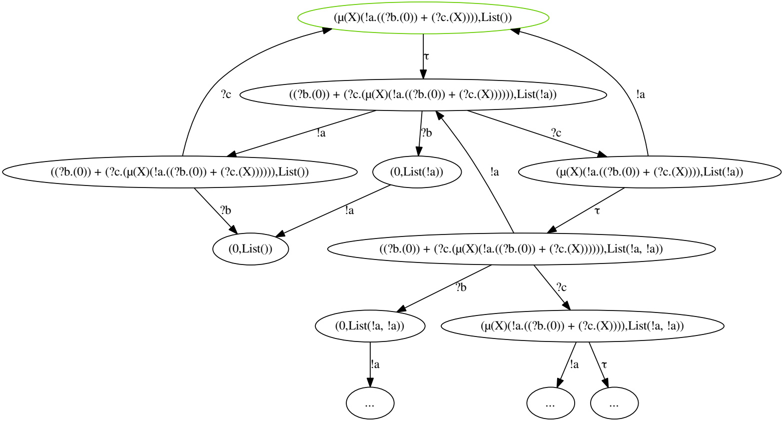


Semantic asynchrony

`ccs1.toDot()`



`ccs1.async.toDot(maxDepth=Finite(4))`



Relations

```
val p1 = CCS("!a.!b.rec(X)(?c.?c.X)")
val p2 = CCS("!a.!b") seq CCS("rec(Y)(?c.Y)") seq CCS("!d")
```

Are `p1` and `p2` **observationally equivalent**?

Relations

```
val p1 = CCS("!a.!b.rec(X)(?c.?c.X)")
val p2 = CCS("!a.!b") seq CCS("rec(Y)(?c.Y)") seq CCS("!d")
```

Are `p1` and `p2` **observationally equivalent**?

```
val b = Bisimulation.build(p1, p2)
```

(Fernandez and Mounier. *Verifying Bisimulations "On the Fly"*, FORTE 1990)

`b` is `Either` a **counterexample** or a `Bisimulation` **relation**

Relations

```
val p1 = CCS("!a.!b.rec(X)(?c.?c.X)")
val p2 = CCS("!a.!b") seq CCS("rec(Y)(?c.Y)") seq CCS("!d")
```

Are `p1` and `p2` **observationally equivalent**?

```
val b = Bisimulation.build(p1, p2)
```

(Fernandez and Mounier. *Verifying Bisimulations "On the Fly"*, FORTE 1990)

`b` is `Either` a **counterexample** or a `Bisimulation` **relation**

```
Right(Set((!a.(!b.(rec(X)(?c.(?c.(X))))), ((!a.(!b.(0)),rec(Y)(?c.(Y))),!d.(0))),
          (!b.(rec(X)(?c.(?c.(X)))), ((!b.(0),rec(Y)(?c.(Y))),!d.(0))),
          (rec(X)(?c.(?c.(X))), ((0,rec(Y)(?c.(Y))),!d.(0))),
          (?c.(rec(X)(?c.(?c.(X)))), ((0,rec(Y)(?c.(Y))),!d.(0)))))
```

Relations

```
val p1 = CCS("!a.!b.rec(X)(?c.?c.X)")
val p2 = CCS("!a.!b") seq CCS("rec(Y)(?c.Y)") seq CCS("!d")
```

Are `p1` and `p2` **observationally equivalent**?

```
val b = Bisimulation.build(p1, p2)
```

(Fernandez and Mounier. *Verifying Bisimulations "On the Fly"*, FORTE 1990)

`b` is `Either` a **counterexample** or a `Bisimulation` **relation**

```
Right(Set((!a.(!b.(rec(X)(?c.(?c.(X))))), ((!a.(!b.(0)),rec(Y)(?c.(Y))),!d.(0))),
          (!b.(rec(X)(?c.(?c.(X)))), ((!b.(0),rec(Y)(?c.(Y))),!d.(0))),
          (rec(X)(?c.(?c.(X))), ((0,rec(Y)(?c.(Y))),!d.(0))),
          (?c.(rec(X)(?c.(?c.(X)))), ((0,rec(Y)(?c.(Y))),!d.(0)))))
```

. . . and relations can be **checked**: `b.right.get.check()` is `true`

Relations

```
val p1 = CCS("!a.!b.rec(X)(?c.?c.X)")
val p2 = CCS("!a.!b") seq CCS("rec(Y)(?c.Y)") seq CCS("!d")
```

Are `p1` and `p2` **observationally equivalent**?

```
val b = Bisimulation.build(p1, p2)
```

(Fernandez and Mounier. *Verifying Bisimulations "On the Fly"*, FORTE 1990)

`b` is `Either` a **counterexample** or a `Bisimulation` **relation**

```
Right(Set((!a.(!b.(rec(X)(?c.(?c.(X))))), ((!a.(!b.(0)),rec(Y)(?c.(Y))),!d.(0))),
          (!b.(rec(X)(?c.(?c.(X)))), ((!b.(0),rec(Y)(?c.(Y))),!d.(0))),
          (rec(X)(?c.(?c.(X))), ((0,rec(Y)(?c.(Y))),!d.(0))),
          (?c.(rec(X)(?c.(?c.(X)))), ((0,rec(Y)(?c.(Y))),!d.(0)))))
```

...and relations can be **checked**: `b.right.get.check()` is `true`

Similar machinery for **simulation**, client/server **progress**, client/server **(I/O) compliance**, ...

Conclusions

`http://tcs.unica.it/software/ltswb`

- initial phases of development
- **praxis-theory-praxis** loop:
  - sticking to theory **reduces code** and **improves reusability**
  - spotting **duplicated code** helps **refining the theory**

Conclusions

`http://tcs.unica.it/software/ltswb`

- initial phases of development
- **praxis-theory-praxis** loop:
  - sticking to theory **reduces code** and **improves reusability**
  - spotting **duplicated code** helps **refining the theory**

Ongoing and future work

- **formalise** the relational $\to$ LTS $\to$ process $\to$ syntax way
- **larger library** of process languages and relations
- **multiparty** interactions via **decorations**? (see PCCS)
- **value-passing** and **time**
- interface with **Gephi**

Thanks!

(Questions?)