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** $(p \text{ seq } q)$

\[
\begin{align*}
  p \xrightarrow{\ell} p' & \quad \Rightarrow \\
(p \text{ seq } q) \xrightarrow{\ell} (p' \text{ seq } q)
\end{align*}
\]

\[
\begin{align*}
  p \rightarrow q \xrightarrow{\ell} q' & \quad \Rightarrow \\
(p \text{ seq } q) \xrightarrow{\ell} (p \text{ seq } q')
\end{align*}
\]
Example: a calculus with sequencing

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

\[
\begin{align*}
    p \xrightarrow{\ell} p' \\
    (p \text{ seq } q) \xrightarrow{\ell} (p' \text{ seq } q)
\end{align*}
\]

\[
\begin{align*}
    p \xrightarrow{\frac{}{}} q \xrightarrow{\ell} q' \\
    (p \text{ seq } q) \xrightarrow{\ell} (p \text{ seq } q')
\end{align*}
\]

This looks **independent from the syntax of** $p$, $q$ and $\ell$
Example: a calculus with sequencing

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

\[
\begin{align*}
p \xrightarrow{\ell} p' \\
(p \text{ seq } q) \xrightarrow{\ell} (p' \text{ seq } q)
\end{align*}
\]

\[
\begin{align*}
p &\not\rightarrow q \xrightarrow{\ell} q' \\
(p \text{ seq } q) &\not\rightarrow (p \text{ seq } q')
\end{align*}
\]

This looks **independent from the syntax of** \(p, q\) **and** \(\ell\)

- what if \(p, q\) 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** \((p \text{ seq } q)\)

\[
\begin{align*}
p \xrightarrow{\ell} p' \\
(p \text{ seq } q) \xrightarrow{\ell} (p' \text{ seq } q)
\end{align*}
\]

\[
\begin{align*}
p \not\xrightarrow{\ell} q \\
(p \text{ seq } q) \not\xrightarrow{\ell} (p \text{ seq } q')
\end{align*}
\]

This looks **independent from the syntax of** \(p, q\) **and** \(\ell\)

▶ what if \(p, q\) come e.g. from **execution logs**?

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

An LTS is a triple $(\Sigma, \Lambda, R)$ where:

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

A **process** is a pair \((\mathcal{L}, p)\)

where \(\mathcal{L}\) 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**
- \(\mathcal{R} \subseteq (\Sigma \times (\Lambda \times \Sigma))\) is the **transition relation**

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

The **process transition** \((L, p) \xrightarrow{\ell} (L, p')\) holds iff \((p, (\ell, p'))\) is in the transition relation of \(L\)
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 \((\mathcal{L}, p)\)
where \(\mathcal{L}\) is an LTS and \(p\) is one of its states

The **process transition** \((\mathcal{L}, p) \stackrel{\ell}{\rightarrow} (\mathcal{L}, p')\) holds
iff \((p, (\ell, p'))\) is in the transition relation of \(\mathcal{L}\)

Let \(\mathcal{R} \subseteq \Delta \times \Gamma\). Then, \(\mathcal{R}(\delta) := \{\gamma \mid (\delta, \gamma) \in \mathcal{R}\}\)
Definitions

An **LTS** is a triple \((\Sigma, \Lambda, R)\) where:

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

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

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

Let \(R \subseteq \Delta \times \Gamma\). Then, \(R(\delta) := \{\gamma \mid (\delta, \gamma) \in R\}\)

\((L, p)(\ell) := \{(L, p') \mid (L, p) \xrightarrow{\ell} (L, p')\}\)
LTS operators: a (boring) example

The union of LTSs $L_1 = (\Sigma_1, \Lambda_1, \mathcal{R}_1)$ and $L_2 = (\Sigma_2, \Lambda_2, \mathcal{R}_2)$ is:

$$L_1 \cup L_2 := \left( \Sigma_1 \cup \Sigma_2, \; \Lambda_1 \cup \Lambda_2, \; \mathcal{R}_1 \cup \mathcal{R}_2 \right)$$
Sequencing of relations

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

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

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

inductively defined by the rules:

$$(p, (\ell, p')) \in \mathcal{R}_1 \quad \Rightarrow \quad ((p, q), (\ell, (p', q))) \in \mathcal{R}_1 ; \mathcal{R}_2$$

$$(\mathcal{R}_1(p) = \emptyset) \quad \Rightarrow \quad ((q, (\ell, q')) \in \mathcal{R}_2 \quad \Rightarrow \quad ((p, q), (\ell, (p, q'))) \in \mathcal{R}_1 ; \mathcal{R}_2$$
Sequencing of relations

Let \( R_1 \subseteq (\Sigma_1 \times (\Lambda_1 \times \Sigma'_1)) \) and \( R_2 \subseteq (\Sigma_2 \times (\Lambda_2 \times \Sigma'_2)) \)

The sequencing of \( R_1 \) and \( R_2 \) is the relation

\[
R_1 ; R_2 \subseteq \left( (\Sigma_1 \times \Sigma_2) \times ( (\Lambda_1 \cup \Lambda_2) \times (\Sigma'_1 \times \Sigma'_2)) \right)
\]

inductively defined by the rules:

\[
\begin{align*}
(p, (\ell, p')) \in R_1 & \quad \Rightarrow \quad ((p, q), (\ell, (p', q))) \in R_1 ; R_2 \\
R_1(p) = \emptyset & \quad \Rightarrow \quad ((q, (\ell, q')), \emptyset) \in R_2
\end{align*}
\]

Equivalently:

\[
(R_1 ; R_2)((p, q)) = \begin{cases} 
\{ (\ell, (p', q)) \mid (\ell, p') \in R_1(p) \} & \text{if } R_1(p) \neq \emptyset \\
\{ (\ell, (p, q')) \mid (\ell, q') \in R_2(q) \} & \text{otherwise}
\end{cases}
\]
Sequencing of LTSs and processes

Let \( L_1 = (\Sigma_1, \Lambda_1, R_1) \) and \( L_2 = (\Sigma_2, \Lambda_2, R_2) \)

The sequencing of \( L_1 \) and \( L_2 \) is:

\[
L_1 ; L_2 := \left( \Sigma_1 \times \Sigma_2, \; \Lambda_1 \cup \Lambda_2, \; R_1 ; R_2 \right)
\]
**Sequencing of LTSs and processes**

Let $L_1 = (\Sigma_1, \Lambda_1, \mathcal{R}_1)$ and $L_2 = (\Sigma_2, \Lambda_2, \mathcal{R}_2)$.

The sequencing of $L_1$ and $L_2$ is:

$$L_1; L_2 := (\Sigma_1 \times \Sigma_2, \Lambda_1 \cup \Lambda_2, \mathcal{R}_1 ; \mathcal{R}_2)$$

The sequencing of processes $(L_1, p)$ and $(L_2, q)$ is:

$$(L_1, p) ; (L_2, q) := (L_1 ; L_2 , (p, q))$$
Sequencing of LTSs and processes

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

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

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

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

$$(\mathcal{L}_1, p) ; (\mathcal{L}_2, q) := \left( \mathcal{L}_1 ; \mathcal{L}_2, (p, q) \right)$$

i.e., $(\mathcal{L}_1, p) ; (\mathcal{L}_2, q)$ observationally behaves as $p$ in $\mathcal{L}_1$, and then as $q$ in $\mathcal{L}_2$
From semantic to syntactic sequencing

Back to our calculus $C$, with sequential composition $(p \text{ seq } q)$
From semantic to syntactic sequencing

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

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

Desideratum: $(L_C, (p \text{ seq } q)) \cong (L_C; L_C, (p, q))$
From semantic to syntactic sequencing

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

Its LTS is $\mathbb{L}_\mathcal{C} = (\Sigma_\mathcal{C}, \Lambda_\mathcal{C}, \mathcal{R}_\mathcal{C})$

**Desideratum:** $(\mathbb{L}_\mathcal{C}, (p \text{ seq } q)) \cong (\mathbb{L}_\mathcal{C}; \mathbb{L}_\mathcal{C}, (p, q))$

We can define the LTS $\mathbb{L}_\mathcal{C}$ so that:

$$(\mathbb{L}_\mathcal{C}, (p \text{ seq } q))(\ell) = \left\{ (\mathbb{L}_\mathcal{C}, (p' \text{ seq } q')) \mid (p', q') \in (\mathbb{L}_\mathcal{C}; \mathbb{L}_\mathcal{C}, (p, q))(\ell) \right\}$$
From semantic to syntactic sequencing

Back to our calculus $C$, with sequential composition $p \seq q$

Its LTS is $L_C = (\Sigma_C, \Lambda_C, R_C)$

Desideratum: $\left( L_C, \left( p \seq q \right) \right) \equiv \left( L_C ; L_C, \left( p, q \right) \right)$

We can define the LTS $L_C$ so that:

$\left( L_C, \left( p \seq q \right) \right)(\ell) = \left\{ \left( L_C, \left( p' \seq q' \right) \right) \mid (p', q') \in \left( L_C ; L_C, \left( p, q \right) \right)(\ell) \right\}$

Which means:

$R_C \left( \left( p \seq q \right) \right) = \left\{ \left( \ell, \left( p' \seq q' \right) \right) \mid \left( \ell, \left( p', q' \right) \right) \in \left( R_C ; R_C \right) \left( \left( p, q \right) \right) \right\}$
Summing up

Syntactic sequencing

Process sequencing

LTS sequencing

Relational sequencing
Summing up

Syntactic operator

Process operator

LTS operator

Relational operator
Summing up

Syntactic operator

↑

Process operator

↑

LTS operator

↑

Relational operator

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

Syntactic **operator**

⇑

Process **operator**

⇑

**LTS operator**

⇑

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 values
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 values**

**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)
Internals

- Set[A] with .contains(x)
  - FiniteSet[A] with .iterator()


Internals

- Set[A] with `.contains(x)`
  - FiniteSet[A] with `.iterator()`

  - FiniteImageRelation[A,B], FiniteRelation[A,B]


Internals

- Set[A] with .contains(x)
  - FiniteSet[A] with .iterator()

  - FiniteImageRelation[A,B], FiniteRelation[A,B]

  - FiniteBranchingRelation3[A,B,C], FiniteRelation3[A,B,C]


Internals

- Set[A] with .contains(x)
  - FiniteSet[A] with .iterator()

  - FiniteImageRelation[A,B], FiniteRelation[A,B]

  - FiniteBranchingRelation3[A,B,C], FiniteRelation3[A,B,C]

  - FiniteBranchingLTS[A,B], FiniteLTS[A,B]

Internals

- Set[A] with .contains(x)
  - FiniteSet[A] with .iterator()

  - FiniteImageRelation[A,B], FiniteRelation[A,B]

  - FiniteBranchingRelation3[A,B,C], FiniteRelation3[A,B,C]

  - FiniteBranchingLTS[A,B], FiniteLTS[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)

\[
\text{val } l1 = \text{LTS(List}((0, "+", 1)), (1, "+", 2)), (2, "+", 3), (2, "-", 1)))
\]

\[
\text{val } l2 = \text{LTS(List}(("p1", "!a", "p2")), ("p2", "?b", "p3")), ("p2", (?c" , "p1")))
\]

\[l1\text{.doDot} \text{ and } l2\text{.toDot} \text{ are:}\]
Defining LTSs (and processes)

```plaintext
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 || l2).toDot is:

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

![Diagram](image)
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**: (De Nicola and Hennessy, 1987; Barbanera and de’ Liguoro, 2010)

  \[ !a \oplus \tau \rightarrow !a \cdot ?c \rightarrow ?c \rightarrow 0 \]

- **Asynchronous**: with an unbounded output queue (Neubauer et al., 2004; Mostrous et al., 2009; . . .)

  \[ !a \oplus !b \tau \rightarrow !a \cdot ?c \tau \rightarrow ?c \cdot !a {\{ !a \} \rightarrow \ldots} \]

  Can we generalise such an “async transformation”?
Synchronous vs. asynchronous semantics

We often work on session types, with two semantics:

- **synchronous:**
  
  \[(\text{De Nicola and Hennessy, 1987; Barbanera and de' Liguoro, 2010})\]

\[!a \oplus !b\]
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)

  \[
  \begin{align*}
  !a \cdot ?c \oplus !b & \xrightarrow{\tau} !a \cdot ?c \xrightarrow{!a} ?c \\
  \end{align*}
  \]
Synchronous vs. asynchronous semantics

We often work on session types, with two semantics:

- **synchronous**: (De Nicola and Hennessy, 1987; Barbanera and de’ Liguoro, 2010)

\[
!a ?c \otimes !b \xrightarrow{\tau} !a ?c \xrightarrow{!a} ?c \xrightarrow{?c} 0
\]
Synchronous vs. asynchronous semantics

We often work on session types, with two semantics:

- **synchronous**:
  
  (De Nicola and Hennessy, 1987; Barbanera and de’ Liguoro, 2010)
  
  \[
  \begin{align*}
  &!a . ?c \oplus !b \xrightarrow{\tau} !a . ?c \\
  &\quad \xrightarrow{!a} ?c \xrightarrow{?c} 0
  \end{align*}
  \]

- **asynchronous**, with an unbounded output queue:
  
  (Neubauer et al., 2004; Mostrous et al., 2009; ...)

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

... and also with async CCS, e.g.:
Synchronous vs. asynchronous semantics

We often work on session types, with two semantics:

- **synchronous:**
  
  (De Nicola and Hennessy, 1987; Barbanera and de’ Liguoro, 2010)

  \[
  !a . ?c \oplus !b \xrightarrow{\tau} !a . ?c \quad !a \quad ?c \quad \xrightarrow{\tau} 0
  \]

- **asynchronous**, with an unbounded output queue:
  
  (Neubauer et al., 2004; Mostrous et al., 2009; …)

  \[
  !a . ?c \oplus !b \quad \xrightarrow{\tau} !a . ?c\
  \]
Synchronous vs. asynchronous semantics

We often work on session types, with two semantics:

- **synchronous:**

  \[
  !a \oplus !b \xrightarrow{\tau} !a \oplus !c \xrightarrow{!a} ?c \xrightarrow{?c} 0
  \]

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

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

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

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

Can we generalise such an “async transformation”?
Synchronous vs. asynchronous semantics

We often work on session types, with two semantics:

- **synchronous:**
  
  (De Nicola and Hennessy, 1987; Barbanera and de Liguoro, 2010)

  \[
  !a \cdot ?c \oplus !b \xrightarrow{\tau} !a \cdot ?c \xrightarrow{!a} ?c \xrightarrow{?c} 0
  \]

- **asynchronous, with an unbounded output queue:**
  
  (Neubauer et al., 2004; Mostrous et al., 2009; ...)

  \[
  !a \cdot ?c \oplus !b \xrightarrow{\tau} !a \cdot ?c \xrightarrow{\tau} ?c[!a]
  \]
Synchronous vs. asynchronous semantics

We often work on session types, with two semantics:

- **synchronous:**
  
  (De Nicola and Hennessy, 1987; Barbanera and de' Liguoro, 2010)
  
  \[
  !a . ?c \oplus !b \overset{\tau}{\rightarrow} !a . ?c \overset{!a}{\rightarrow} ?c \overset{?c}{\rightarrow} 0
  \]

- **asynchronous**, with an unbounded output queue:
  
  (Neubauer et al., 2004; Mostrous et al., 2009; . . .)

  \[
  !a . ?c \oplus !b [\tau] \overset{\tau}{\rightarrow} !a . ?c [\tau] \overset{?c[!a]}{\rightarrow} ?c [!a]
  \]
Synchronous vs. asynchronous semantics

We often work on session types, with two semantics:

- **synchronous**: (De Nicola and Hennessy, 1987; Barbanera and de’ Liguoro, 2010)
  \[
  !a . ?c \oplus !b \rightarrow !a . ?c \rightarrow !a \rightarrow ?c \rightarrow 0
  \]

- **asynchronous**, with an unbounded output queue: (Neubauer et al., 2004; Mostrous et al., 2009; . . .)
  \[
  !a . ?c \oplus !b \rightarrow !a . ?c \rightarrow ?c[!a] \rightarrow !a \rightarrow ?c \rightarrow 0
  \]
**Synchronous vs. asynchronous semantics**

We often work on session types, with two semantics:

- **synchronous:**
  
  (De Nicola and Hennessy, 1987; Barbanera and de’ Liguoro, 2010)

  \[
  \begin{align*}
  & !a . ?c \oplus !b \rightarrow^\tau !a . ?c \rightarrow^!a ?c \rightarrow^?c 0 \\
  & !a . ?c \rightarrow^\tau !a . ?c \rightarrow^?c !a !a \rightarrow^!a ?c \rightarrow^?c 0
  \end{align*}
  \]

- **asynchronous**, with an unbounded output queue:
  
  (Neubauer et al., 2004; Mostrous et al., 2009; . . . )

  \[
  \begin{align*}
  & !a . ?c \oplus !b \rightarrow^\tau !a . ?c \rightarrow^?c !a !a \rightarrow^!a ?c \rightarrow^?c 0
  \end{align*}
  \]

Can we generalise such an "async transformation"?
Synchronous vs. asynchronous semantics

We often work on session types, with two semantics:

- **synchronous**:  
  (De Nicola and Hennessy, 1987; Barbanera and de' Liguoro, 2010)
  \[
  !a . ?c \oplus !b \xrightarrow{\tau} !a . ?c \xrightarrow{!a} ?c \xrightarrow{?c} 0
  \]

- **asynchronous**, with an unbounded output queue:  
  (Neubauer et al., 2004; Mostrous et al., 2009; ...)  
  \[
  !a . ?c \oplus !b \xrightarrow{\tau} !a . ?c \xrightarrow{\tau} ?c[!a] \left\{ \begin{array}{c}
  \xrightarrow{!a} ?c[] \xrightarrow{?c} 0[] \\
  \xrightarrow{?c} 0[!a] \xrightarrow{!a} 0[]
  \end{array} \right.
  \]
Synchronous vs. asynchronous semantics

We often work on session types, with two semantics:

- **synchronous:**
  
  (De Nicola and Hennessy, 1987; Barbanera and de’ Liguoro, 2010)
  
  \[
  !a \cdot ?c \oplus !b \xrightarrow{\tau} !a \cdot ?c \xrightarrow{!a} ?c \xrightarrow{?c} 0
  \]

- **asynchronous**, with an **unbounded output queue**:
  
  (Neubauer et al., 2004; Mostrous et al., 2009; . . . )
  
  \[
  !a \cdot ?c \oplus !b\[\]\xrightarrow{\tau} !a \cdot ?c\[\]\xrightarrow{?c[!a]} \left\{ \begin{array}{c}
  \xrightarrow{!a} ?c\[\]\xrightarrow{?c} 0\[\] \\
  \xrightarrow{?c} 0[!a] \xrightarrow{!a} 0[!a]
  \end{array} \right.
  \]

  . . . and also with **(async) CCS**, e.g.:
  
  \(?a \mid \text{rec}_X \left( ?b. (\tau \cdot !c + ?d. X) \right) ![e. !d]\)
Synchronous vs. asynchronous semantics

We often work on session types, with two semantics:

- **synchronous:**

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

  \[
  !a \oplus !b \xrightarrow{\tau} !a \cdot ?c \xrightarrow{!a} ?c \xrightarrow{?c} 0
  \]

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

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

  \[
  !a \cdot ?c \oplus !b \cdot !c \xrightarrow{\tau} !a \cdot ?c \cdot !c \xrightarrow{?c} \{ \xrightarrow{!a} \cdot ?c \xrightarrow{?c} 0 \cdot !a \xrightarrow{!a} 0 \}
  \]

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

  \[
  ?a | \text{rec}_X \left( ?b \left( \tau \cdot !c + ?d \cdot X \right) \right) [!e \cdot !d]
  \]

Can we generalise such an “async transformation”?
Semantic asynchrony

ccs1.toDot()
Semantic asynchrony

ccs1.toDot()

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

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

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

```scala
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 \( p_1 \) and \( p_2 \) observationally equivalent?

```scala
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

```scala
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

\[
\text{val } p1 = \text{CCS}("!a.!b.rec(X)(?c.?c.X)")
\]
\[
\text{val } p2 = \text{CCS}("!a.!b") \text{ seq } \text{CCS}("\text{rec}(Y)(?c.Y)") \text{ seq } \text{CCS}("!d")
\]

Are \(p1\) and \(p2\) observationally equivalent?

\[
\text{val } b = \text{Bisimulation}.\text{build}(p1, p2)
\]

(Fernandez and Mounier. Verifying Bisimulations “On the Fly”, FORTE 1990)

\(b\) is Either a \textbf{counterexample} or a \textbf{Bisimulation relation}

\[
\text{Right(}\text{Set}(!a.(!b.(\text{rec}(X)(?c.(?c.(X))))), (!a.(!b.(0)),\text{rec}(Y)(?c.(Y))),!d.(0))),

(!b.(\text{rec}(X)(?c.(?c.(X)))), (!b.(0),\text{rec}(Y)(?c.(Y))),!d.(0))),

(\text{rec}(X)(?c.(?c.(X)))), ((0,\text{rec}(Y)(?c.(Y))),!d.(0))),

(?c.(\text{rec}(X)(?c.(?c.(X)))), ((0,\text{rec}(Y)(?c.(Y))),!d.(0))))
\]

...and relations can be \textbf{checked}: \(b.\text{right}.\text{get}.\text{check}()\) is true

Similar machinery for \textbf{simulation}, client/server \textbf{progress},
client/server \textbf{(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 → LTS → process → syntax way
- larger library of process languages and relations
- multiparty interactions via decorations? (see PCCS)
- value-passing and time
- interface with Gephi
Thanks!
(Questions?)