Concurrency 4 = CCS (2/4)

Scoping, weak and strong bisimulation

Pierre-Louis Curien (CNRS – Université Paris 7)

MPRI concurrency course 2004/2005 with :
Jean-Jacques Lévy (INRIA-Rocquencourt)
Eric Goubault (CEA)
James Leifer (INRIA - Rocq)
Catuscia Palamidessi (INRIA - Futurs)

(http://pauillac.inria.fr/~leifer/teaching/mpri-concurrency-2004)

Scale and recursion (1/4)

Consider (example of Frank Valencia) (we write \( \mu \) for \( \mu \cdot 0 \)) :

\[
P_1 = (\nu x) P = (\nu x) \left( (a \cdot \tau_{\alpha}) | P \right) \ldots K
\]

Applying the rules, we have (two unfoldings) :

\[
\begin{align*}
(\nu x) (a \cdot \tau_{\alpha}) &= (\nu x) (\nu y) (a \cdot \tau_{\alpha}) | (a \cdot \tau_{\alpha}) | K) \\
&\Rightarrow (\nu x) (0) (a \cdot \tau_{\alpha}) | (a \cdot \tau_{\alpha}) | K)
\end{align*}
\]

What about \( P_2 = (\nu x) K = (\nu y) (b \cdot \tau_{\alpha}) | (b \cdot \tau_{\alpha}) | K) \ldots K \) : the double enfolding yields \( (\nu y) (b \cdot \tau_{\alpha}) | (b \cdot \tau_{\alpha}) | (b \cdot \tau_{\alpha}) | K) \), which is deadlocked, while the first definition of \( K \) allows to perform \( \tau_{\alpha} \) (notice the capture of \( \pi \)).

Scope and recursion (2/4)

\[
P_1 = (\nu x) P = (\nu x) \left( (a \cdot \tau_{\alpha}) | K \right) \ldots K
\]

\[
P_2 = (\nu x) P = (\nu x) \left( (b \cdot \tau_{\alpha}) | K \right) \ldots K
\]

There is a tension :
- These two definitions have a different behaviour.
- The identity of bounded names should be irrelevant (\( \alpha \)-conversion).

So let us rename \( a \) in the first definition :

\[
P_3 = (\nu x) P = (\nu x) \left( (b \cdot \tau_{\alpha}) | K[a \leftarrow b] \right) \ldots K
\]

But what is \( K[a \leftarrow b] \) ? Well, we argue that it is not \( K \), it is a substitution or (explicit) relabelling which is delayed until \( K \) is replaced by its actual definition (cf. e.g. \( \lambda \)-calculus with term metavariables and explicit substitutions)

So, all is well, we maintain both \( \alpha \)-conversion \( (P_1 = P_3) \) and the difference of behaviour \( (P_1 \neq P_2) \), and the tension is resolved . . .

Scope and recursion (3/4)

In an \( \alpha \)-conversion \( (\nu x) P = (\nu y) P[x \leftarrow y] \), \( y \) should be chosen free in \( P \). BUT when substitution arrives on \( K \), how do I know whether \( y \) is free in \( K \) ? For example, in

\[
P_4 = (\nu x) P = (\nu x) \left( (a \cdot \tau_{\alpha}) | K \right) \ldots K
\]

\( b \) is free in \( K \), but I cannot know it from just looking at the subterm \( (\nu y) (a \cdot \tau_{\alpha}) | (a \cdot \tau_{\alpha}) | K) \).

Clean solution (definitions with parameters) : maintain the list of free variables of a constant \( K \), and hence write constants always in the form \( K[\beta] \) and make sure that in a definition \( \nu x K[\beta] = P \ldots Q \) we have \( \forall \nu \beta (P) \subseteq \beta \). (cf. syntax adopted in Milner’s \( \pi \)-calculus book).

And now, relabelling can be omitted from syntax, i.e. left implicit, since, e.g. \( K[\alpha \leftarrow \beta] = K[\alpha \leftarrow \beta] \).
Scope and recursion (4/4)

A “real” example: Consider the following linking operation:
\[ P \triangleleft Q = (\nu i', z', d')(P[i, z, d \rightarrow i', z', d']|Q[\text{inc}, \text{zero}, \text{dec} \leftarrow i', z', d']) \]

In particular
\[ C(\text{inc}, \text{zero}, \text{dec}, z, d) \sim C(\text{inc}, \text{zero}, \text{dec}, z, d) = (\nu i', z', d')(C(\text{inc}, \text{zero}, \text{dec}, z, d'))(C(i', z', d', z, d)) \]

A (unbounded) counter:
\[ C = \text{inc} \cdot (C \triangleleft C) + \text{dec} \cdot D \quad D = \overline{D} \cdot C + \tau \cdot B \quad B = \text{inc} \cdot (C \triangleleft B) + \text{zero} \cdot B \]

An example of execution:
\[ B \overset{\text{zero}}{\rightarrow} B \overset{\text{inc}}{\rightarrow} (C \triangleleft B) \overset{\text{inc}}{\rightarrow} ((C \triangleleft C) \triangleleft B) \overset{\text{dec}}{\rightarrow} ((D \triangleleft C) \triangleleft B) \]
\[ \vdash ((C \triangleleft D) \triangleleft B) \overset{\text{dec}}{\rightarrow} ((D \triangleleft D) \triangleleft B) \vdash ((D \triangleleft C) \triangleleft B) \]
\[ \vdash ((B \triangleleft B) \triangleleft B) \overset{\text{inc}}{\rightarrow} ((C \triangleleft B) \triangleleft B) \ldots \]

**Exercice 1** Show that there is no derivation \( B \overset{\tau^* \text{inc} \tau^* \text{dec} \tau^* \text{dec}}{\rightarrow} \).

---

Bisimilarity is not trace equivalence

As automata \( P = a \cdot (b + c) \) and \( Q = a \cdot b + a \cdot c \) recognize the same language \( \{ab, ac\} \) of traces.

As processes, they are not bisimilar \( (Q \text{ does not even simulate } P) \). \( P \) keeps the choice after performing \( a, Q \) not.

Think of \( a \) as inserting 40 cents, \( b \) as getting tea and \( c \) as getting coffee. Imagine a vending machine with a slot for \( a \) and two buttons for \( b \) and \( c \). The machine allows you to press \( b \) (resp. \( c \)) only if action \( b \) (resp. \( c \)) can be performed. As a customer you will prefer \( P \).

---

Variations on bisimilarity (1/3)

A bisimulation up to \( \sim \) is a relation \( R \) such that for all \( P, Q \):

\[ PRQ \Rightarrow \forall i. P_i \overset{R}{\Rightarrow} i \exists Q'_i \overset{R}{\Rightarrow} i' \quad \text{and} \quad P_i \overset{\tau}{\Rightarrow} i \quad \text{and} \quad Q'_i \overset{\tau}{\Rightarrow} i' \quad \text{and} \quad P_i \overset{R}{\Rightarrow} i \sim R \sim i' \]

If \( R \) is strong bisimulation up to \( \sim \), then \( R \sqsubseteq \sim \).

**Exercice 3** Prove it.

Hence, to show \( P \sim Q \), it is enough to find a bisimulation up to \( \sim \) such that \( P \mathrel{R} Q \).

---

Structural equivalence

**Exercice 2** Show that structural equivalence \( \equiv \) is included in (strong) bisimulation \( \sim \).
Variations on bisimilarity (2/3)

As an example, take

\[
\begin{align*}
\text{Sem} &= \text{P} \cdot \text{Sem}' \\
\text{Sem}' &= \nu \cdot \text{Sem}
\end{align*}
\]

Then a (strong) bisimulation up-to witnessing that \((\text{Sem}|\text{Sem}|\text{Sem}) \sim \text{Sem}^\dagger\) is, say :

\[
\{ ((\text{Sem}|\text{Sem}|\text{Sem}), \text{Sem}^\dagger) \}
\]

Weak† = Weak⋆.

Variations on bisimilarity (3/3)

For any LTS, one can change \(\sim\) to \(\sim^\star\) (words of actions), setting

\[
P \sim Q \text{ if } \begin{cases}
s = \mu_1 \ldots \nu_n \text{ and} \\
(\exists P_1, \ldots, P_n \ (P_n = Q \text{ and } P \sim P_1 \ldots \sim P_n))
\end{cases}
\]

This yields a new LTS, call it LTS∗ (the path LTS) . Then the notions of LTS and of LTS∗ bisimulation coincide.

From strong to weak bisimulation (1/2)

Take the LTS of CCS, with \(\sim = L \cup T \cup \{\tau\}\), call it Strong. The bisimulation for this system is called strong bisimulation.

Take Strong∗ (its path LTS).

Consider the following LTS, call it Weak†, with the same set of actions as Strong∗ :

\[
P \setsto{\tau} Q \text{ if and only if } (\exists t \ P \setsto{\tau} Q \text{ and } \hat{s} = \hat{t})
\]

where the function \(s \mapsto \hat{s}\) is defined as follows :

\[
\hat{\nu} = \nu \quad \hat{\tau} = \epsilon \quad \hat{\alpha} = \alpha \quad \hat{s\mu} = \hat{s}\hat{\mu}
\]

The idea is that weak bisimulation is bisimulation with possibly \(\tau\) actions interspersed.

Let Weak be the LTS on Act whose transitions are \(P \setsto{\tau} Q\), that is :

\[
P \setsto{\tau} Q \text{ if and only if } P \setsto{\tau} Q \quad P \setsto{\tau} Q \text{ if and only if } P \setsto{\nu_1 \ldots \nu_n} Q
\]

Then one has Weak† = Weak∗.

From strong to weak bisimulation (2/2)

None of the three equivalent definition of weak bisimulation (Weak, Weak†, Weak∗) is practical. The following is a fourth, equivalent, and more tractable version :

A weak bisimulation is a relation \(R\) such that

\[
P R Q \Rightarrow \forall P', P''. \ (P' \setsto{\tau} P'' \Rightarrow \exists Q' Q'' \ (Q' \setsto{\tau} Q'' \text{ and } P' R Q'))\] and conversely

Two processes are weakly bisimilar if (notation \(P \approx Q\)) if there exists a weak bisimulation \(R\) such that \(P R Q\).
Bisimulation is a congruence (1/6)

We define $\approx^*$ inductively by the following rules:

\[
\begin{align*}
    P & \approx Q & P \approx^* Q & \quad P \approx^* Q &Q \approx^* R \\
    P \approx^* Q & & Q \approx^* P & & P \approx^* R
\end{align*}
\]

\[
\forall i \in I, P_i \approx^* Q_i, \quad \Sigma_{i \in I} \mu_i \cdot P_i \approx^* \Sigma_{i \in I} \mu_i \cdot Q_i, \quad P_1 | P_2 \approx^* Q_1 | Q_2, \quad \nu \cdot P \approx^* (\nu a)Q
\]

Clearly $\approx \subseteq \approx^*$ and $\approx^*$ is a congruence, by construction. It is enough to show that $\approx^*$ is a bisimulation (since then $\approx = \approx^*$ is a congruence).

Bisimulation is a congruence (2/6)

Proof by rule induction. We look at case $P_1 | P_2 \approx Q_1 | Q_2$:

1. **(backward) decomposition phase**: if $P_1|P_2 \not\approx P'$, then $P' = P'_1|P'_2$ and three cases may occur, corresponding to the three rules for parallel composition in the labelled operational semantics. We only consider the synchronisation case. If $P_1 \not\approx P'_1$ and $P_2 \not\approx P'_2$, then

2. **by induction** there exists $Q'_1$ such that $Q_1 \not\approx Q'_1$ and $P'_1 \not\approx Q'_1$, and there exists $Q'_2$ such that $Q_2 \not\approx Q'_2$ and $P'_2 \not\approx Q'_2$.

3. Hence **(forward phase)** we have $Q_1 | Q_2 \not\approx Q'_1 | Q'_2$ and $P'_1 | P'_2 \not\approx Q'_1 | Q'_2$.

Bisimulation is a congruence (3/6)

$\approx$ is also a congruence (for our choice of language with guarded sums).

Same proof technique: define $\approx^*$. For the forward phase, we use the following properties, which are true:

\[
\begin{align*}
    (P \not\approx P') & \Rightarrow (\nu a)P \not\approx (\nu a)Q' \\
    (Q_1 \not\approx Q'_1) & \Rightarrow (Q_1 | Q_2 \not\approx Q'_1 | Q_2) \\
    (Q_1 \not\approx Q'_1 \text{ and } Q_2 \not\approx Q'_2) & \Rightarrow (Q_1 | Q_2 \not\approx Q'_1 | Q'_2)
\end{align*}
\]

Bisimulation is a congruence (4/6)

Consider CCS with prefix and sums instead of guarded sums, i.e., replace $\Sigma_{i \in I} \mu_i \cdot P_i$ by two constructs $\Sigma_{i \in I} P_i$ and $a \cdot P$, with rules:

\[
\begin{align*}
    P_i \not\approx P'_i & \Rightarrow \Sigma_{i \in I} P_i \not\approx P'_i \\
    \mu \cdot P \not\approx P
\end{align*}
\]

Then strong bisimulation is a congruence, and weak bisimulation is not a congruence.

The problem does not arise because more processes (like $P + (Q|R)$) are allowed.
Bisimulation is a congruence (5/6)

What goes wrong is the sum rule? For the forward phase, we would need the property:

\[(Q_1 \lessgtr Q'_1) \Rightarrow (Q_1 + Q_2 \lessgtr Q'_1)\]

which does not hold (take \(\mu = \tau\) and \(Q'_1 = Q_1\)).

Counter-example: \(\tau \cdot a \cdot 0 + b \cdot 0 \not\approx a \cdot 0 + b \cdot 0\)

Bisimulation is a congruence (6/6)

We have left out recursion, but even so we have:

**Proposition:** For any process \(S\) (possibly with recursive definitions) with free variables in \(\vec{K}\):

\[\forall \vec{Q}, \vec{Q}' (\vec{Q} \approx \vec{Q}' \Rightarrow S[\vec{K} \leftarrow \vec{Q}] \approx S[\vec{K} \leftarrow \vec{Q}'])\]

The proof is by induction on the size of \(S\). The non-recursion cases follow by congruence. For the recursive definition case \(S = \text{let} \vec{L} = \vec{P} \vdots \text{L}_j\), the trick is to unfold:

\[
S[\vec{K} \leftarrow \vec{Q}] \overset{\text{def}}{=} \text{let} \vec{L} = \vec{P}[\vec{K} \leftarrow \vec{Q}] \vdots \text{L}_j \\
\approx \text{P}_j[\vec{K} \leftarrow \vec{Q}][\vec{L} \leftarrow \text{let} \vec{L} = \vec{P} \vdots \text{L}_j] \\
\overset{\text{ind}}{=} \text{P}_j[\vec{K} \leftarrow \vec{Q}'][\vec{L} \leftarrow \text{let} \vec{L} = \vec{P} \vdots \text{L}_j] \\
\approx \overset{\text{ind}}{S[\vec{K} \leftarrow \vec{Q}']}\]

Counter-example: \(\tau \cdot a \cdot 0 + b \cdot 0 \not\approx a \cdot 0 + b \cdot 0\)