



# Concepts of Concurrent Computation

Bertrand Meyer  
Sebastian Nanz  
Chris Poskitt

Lecture 13: CCS Advanced Concepts  
& the  $\pi$ -calculus



# CCS: Weak Bisimulations



- Further use of bisimulations: refinement of systems
- We would like to state that two processes **Spec** and **Imp** behave the same, where **Imp** specifies the computation in greater detail
- This is not possible with strong bisimulations, as every action needs to be matched in equivalent processes
- Key to a weaker notion of equivalence: abstract from internal actions
- **Idea:** an external observer who focuses on visible actions but ignores all internal behavior

# Weak bisimulation (1)



- We write  $P \xRightarrow{\alpha} Q$  if  $P$  can reach  $Q$  via an  $\alpha$ -transition, preceded and followed by zero or more  $\tau$ -transitions

$$P \xrightarrow{\tau^*} P' \xrightarrow{\alpha} P'' \xrightarrow{\tau^*} Q$$

Furthermore,  $P \xRightarrow{\tau} Q$  holds if  $P = Q$

- This definition allows us to "erase" sequences of  $\tau$ -transitions in a new definition of behavioral equivalence: weak bisimulation

# Weak bisimulation (2)



Let  $(Proc, Act, \{\xrightarrow{\alpha} \mid \alpha \in Act\})$  be an LTS.

## Weak bisimulation

A binary relation  $R \subseteq Proc \times Proc$  is a **weak bisimulation** if  $(P, Q) \in R$  implies for all  $\alpha \in Act$

- if  $P \xrightarrow{\alpha} P'$  then  $Q \xRightarrow{\alpha} Q'$  for some  $Q'$  such that  $(P', Q') \in R$
- if  $Q \xrightarrow{\alpha} Q'$  then  $P \xRightarrow{\alpha} P'$  for some  $P'$  such that  $(P', Q') \in R$

## Weak bisimilarity

Two processes  $P$  and  $Q$  are **weakly bisimilar**,  $P \approx Q$ , if there is a weak bisimulation  $R$  such that  $(P, Q) \in R$

# Example: Weak bisimulation

---



Consider the following CCS processes:

$$P_0 = a.P_0 + b.P_1 + \tau.P_1$$

$$P_1 = a.P_1 + \tau.P_2$$

$$P_2 = b.P_0$$

$$Q_1 = a.Q_1 + \tau.Q_2$$

$$Q_2 = b.Q_1$$

Is  $P_0 \approx Q_1$  ?

Yes, since  $\{(P_0, Q_1), (P_1, Q_1), (P_2, Q_2)\}$  is a weak bisimulation.



# CCS: Value Passing

- For modeling, it is often helpful to be able to express that values can be passed when processes are synchronizing
- For example, a buffer of size one can be modeled as follows:

$$\text{Buffer} = \text{append}(x).\overline{\text{remove}}(x).\text{Buffer}$$

- The value transmitted over channel `append` is bound to variable `x`
- For example, if the value is `d` then we get in the next step:

$$\overline{\text{remove}}(d).\text{Buffer}$$



# Example: Producers-consumers in CCS



- Buffer of size two:

Buffer = append(x).Buffer1(x)

Buffer1(x) =  $\overline{\text{remove}}$ (x).Buffer + append(y).Buffer2(x, y)

Buffer2(x, y) =  $\overline{\text{remove}}$ (x).Buffer1(y)

- Producers and Consumers:

Producer(x) =  $\overline{\text{append}}$ (x).Producer(x + 1)

Consumer = remove(x).Consumer

- Full system:

(Producer(0) | Buffer | Consumer) \ {append, remove}

# Superfluity of value-passing



- It can be shown that the original calculus is just as expressive as the value-passing calculus
- We demonstrate the main argument of this proof by a simple example: we translate a process with value-passing into one without

$$\text{Buffer} = \text{append}(x).\text{Buffer1}(x)$$

$$\text{Buffer1}(x) = \overline{\text{remove}}(x).\text{Buffer}$$

Fix a set of values, e.g. booleans, to be stored in the buffer, then the following process is equivalent:

$$\text{Buffer} = \text{append}_0.\text{Buffer1}_0 + \text{append}_1.\text{Buffer1}_1$$

$$\text{Buffer1}_0 = \overline{\text{remove}}_0.\text{Buffer}$$

$$\text{Buffer1}_1 = \overline{\text{remove}}_1.\text{Buffer}$$

In general, this requires infinite summations and infinitely many equations



# The $\pi$ -calculus



- In *CCS* all communication links are static
- This leads to problems when trying to model dynamically changing systems
- Example: a server  $S$  increments every value it receives
$$S = a(x).\bar{a}(x + 1).0 \mid S$$
- If processes try to access the server, the responses may not be correctly matched
$$\bar{a}(3).a(x).P(x) \mid \bar{a}(5).a(y).Q(y) \mid S \rightarrow \dots \rightarrow P(6) \mid Q(4) \mid \dots$$



- To remove the limitation of *CCS*, the  $\pi$ -calculus allows values to include channel names
- The incrementation server can be reprogrammed as

$$S = a(x, b).\bar{b}\langle x + 1 \rangle.0 \mid S$$

- Note the use of angle brackets  $\langle \dots \rangle$  to denote the output tuple



- The restriction operator  $P \setminus L$  of CCS is overly restrictive
- We would also like that channels can be passed outside their original scope
- In the  $\pi$ -calculus, the restriction (or creation) operator is written

$(\text{new } x) P$

and creates a new name  $x$  with scope  $P$

- The name can however be communicated outside its original scope (**scope extrusion**), changing the scope of the binder:

$(\text{new } y)( \bar{x}\langle y \rangle \mid y(v).P(v) ) \mid x(u).\bar{u}\langle 2 \rangle$

$\rightarrow (\text{new } y) ( y(v).P(v) \mid \bar{y}\langle 2 \rangle )$

$\rightarrow (\text{new } y)( P(2) )$

## Action prefixes

$\pi ::= x(y)$	receive $y$ along $x$
$\bar{x}\langle y \rangle$	send $y$ along $x$
$\tau$	unobservable action

## Process syntax

$P ::= \sum \pi_i.P_i$	summation
$P_1 \mid P_2$	parallel
$(\text{new } x) P$	new name creation
$!P$	replication

# Structural congruence



- Two expressions are **structurally congruent**, written  $P \equiv Q$ , if they can be transformed into the other using the following rules:
  1. Renaming of bound variables (alpha-conversion)
  2. Reordering of terms in a summation
  3. Associativity and commutativity of parallel;  $P \mid 0 \equiv P$
  4.  $(\text{new } x) (P \mid Q) \equiv P \mid (\text{new } x) Q$  if  $x$  not free in  $P$   
 $(\text{new } x) 0 \equiv 0$ ,  $(\text{new } x) (\text{new } y) P \equiv (\text{new } y) (\text{new } x) P$
  5.  $!P \equiv P \mid !P$



$$\text{TAU } \tau.P + M \rightarrow P$$

$$\text{REACT } x(y).P + M \mid \bar{x}\langle z \rangle.Q + N \rightarrow P[z/y] \mid Q$$

$$\text{PAR } \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q}$$

$$\text{RES } \frac{P \rightarrow P'}{(\text{new } x) P \rightarrow (\text{new } x) P'}$$

$$\text{STRUCT } \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$



- An appropriate notion of process equivalence  $P \approx Q$  for the  $\pi$ -calculus:
  - preserves the equivalence in all contexts
  - means that we can make the same observations for  $P$  and for  $Q$
  - implies that  $P$  and  $Q$  mimic their reaction steps
- The equivalence can be developed formally as in the case of  $CCS$ , with some complications due to the reaction semantics (other than in the labeled semantics, the observables are not exposed by the transitions)



- Small calculus, but very expressive:
  - encoding of data structures
  - encoding functions as processes
  - encoding higher-order behavior
  - encoding polyadic with monadic communication
  - ...



- Many "fundamental" models of concurrency: *CCS*, *CSP*,  $\pi$ -calculus
- The reason for this is that there are many forms of concurrency one might like to describe
- The  $\pi$ -calculus takes mobility into account, which is not the case for *CCS* and *CSP*
- Process calculi provide models of concurrency, not a programming languages - for "everyday use" too many details are abstracted away
- However, the formal techniques studied in process calculi can help to design better concurrent programming languages as well