ETH Zürich

Chair of Software Engineering

Concepts of Concurrent Computation

Bertrand Meyer
Sebastian Nanz

Lecture 12: CCS Advanced Concepts & the π-calculus

CCS: Weak Bisimulations

Refinement

- 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 $p \xRightarrow{a} q$ if p can reach q via an a-transition, preceeded and followed by zero or more τ-transitions

$$p \xrightarrow{\tau}{}^{*} p' \xrightarrow{a} q' \xrightarrow{\tau}{}^{*} q$$

Furthermore, $p \xRightarrow{\tau} q$ holds if $p = q$

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

Weak bisimulation (2)

- **Definition** (Weak bisimulation)

Let (Q, A, $\longrightarrow$) be a labeled transition system. R $\subseteq$ Q x Q is a weak bisimulation if (p, q) $\in$ R implies for all a $\in$ A

- if p $\xrightarrow{a}$ p' then q $\overset{a}{\Longrightarrow}$ q' such that (p', q') $\in$ R
- if q $\xrightarrow{a}$ q' then p $\overset{a}{\Longrightarrow}$ p' such that (p', q') $\in$ R

Two states 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

Value-passing CCS

- 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

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

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

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

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

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

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

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

In general, this requires infinite summations and infinitely many equations

The π-calculus

Limitations of CCS

- 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

$$S = a(x).\overline{a}(x + 1).0 \mid S$$

- If processes try to access the server, the responses may not be correctly matched

$$\overline{a}(3).a(x).P(x) \mid \overline{a}(5).a(y).Q(y) \mid S \to \ldots \to P(6) \mid Q(4) \mid \ldots$$

Names

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

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

- Note the use of angle brackets ⟨ ... ⟩ to denote the output tuple

Restriction

- The restriction operator P \ L of CCS is overly restrictive
- We would also like that channels can be passed outside their original scope
- In the π-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:

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

$$\quad \text{->}\ (\text{new } y)\ (\ y(v).P(v) \mid \overline{y}\langle 2\rangle)$$

$$\quad\quad \text{->}\ (\text{new } y)(\ P(2)\ )$$

Syntax of the π-calculus

Action prefixes

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

Process syntax

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

Structural congruence

- Two expressions are *structurally congruent*, written P ≡ 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 | 0 ≡ P
4. (new x) (P | Q) ≡ P | (new x) Q   if x not free in P
   (new x) 0 ≡ 0,   (new x) (new y) P ≡ (new y) (new x) P
5. !P ≡ P | !P

Reaction semantics

TAU $\tau.P + M \rightarrow P$

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

PAR $\dfrac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q}$

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

STRUCT $\dfrac{Q \equiv P \qquad P \rightarrow P' \qquad P' \equiv Q'}{Q \rightarrow Q'}$

Equivalence

- An appropriate notion of process equivalence P $\simeq$ Q for the π-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)

Expressiveness

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

Conclusion

- Many "fundamental" models of concurrency: CCS, CSP, π-calculus
- The reason for this is that there are many forms of concurrency one might like to describe
- The π-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