CCS: Syntax and Semantics

Marco Carbone

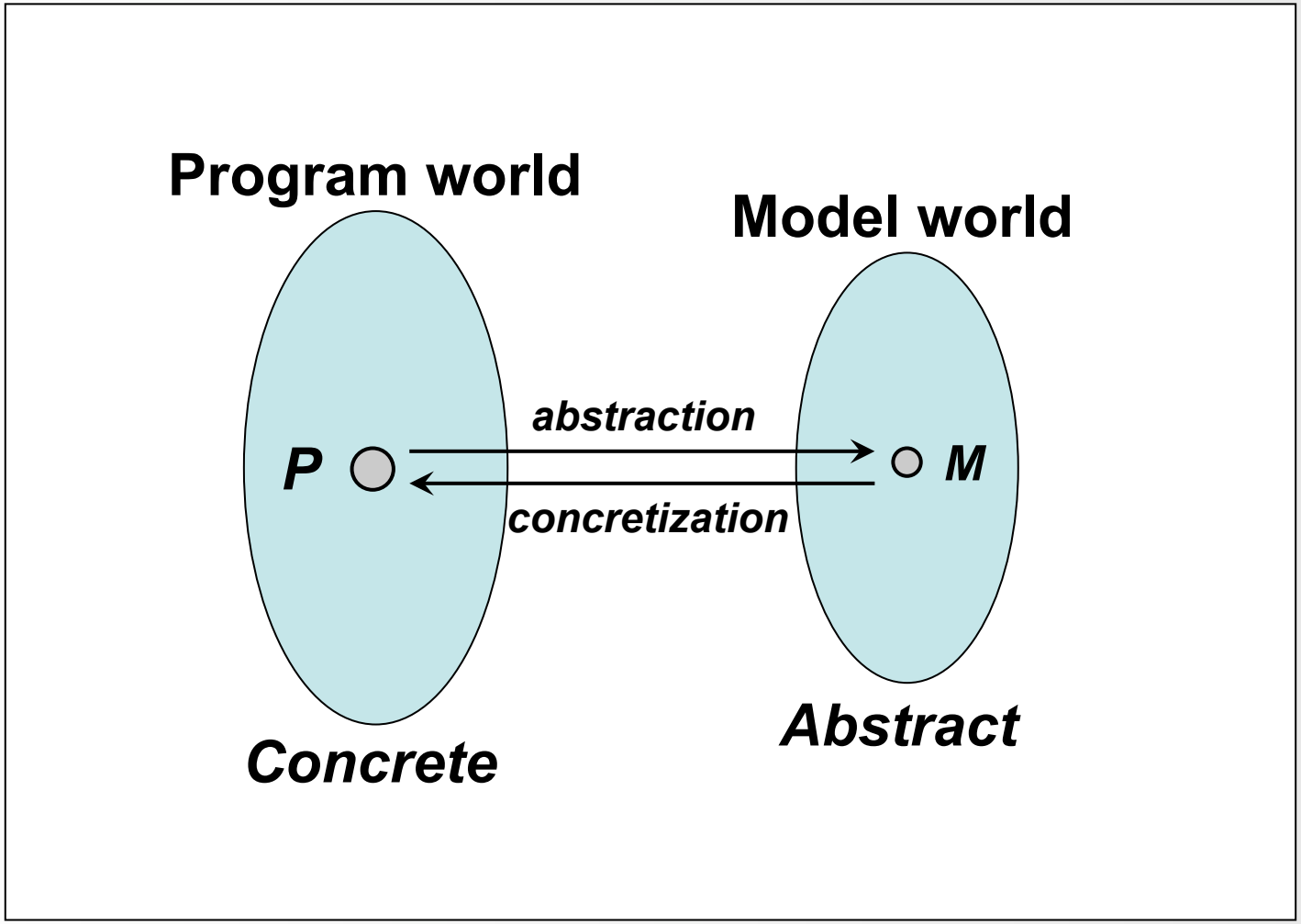
April 2009

Special Thanks

Thanks to *Claus Brabrand* for providing a lot of material for this part

Outline

- Concurrent vs. Sequential Systems
- ***CCS***: *"Calculus of Communicating Systems"*
  - *By-example (one construction at a time)*
- Syntax of ***CCS***
  - 7 linguistic constructions(!)
- Semantics of ***CCS***
  - 1 page(!)
- A Tale of two ***Coca-Cola*** Machines


Program world
Model world
abstraction
P
M
concretization
Concrete
Abstract

CONCURRENCY vs. SEQUENTIALITY

April 2009

Concurrency vs. Sequentiality

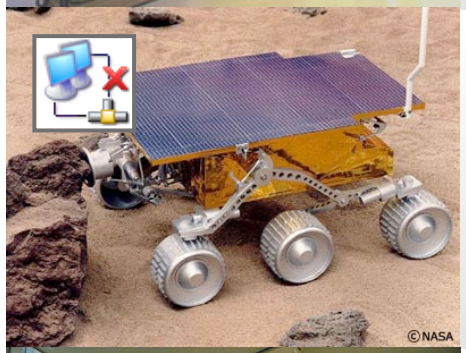
- Sequential programming:
  - Describe computation as a "*reduction*" of expressions to values
  - Inherently *deterministic*
  - *Termination* often desirable
  - Resulting *value* is of primary interest and focus
- Concurrent programming:
  - Describe execution as *"process evolution"*
  - Inherently *non-deterministic*
  - *Non-termination* often desirable (e.g. Op.Sys., Control sys, Cell-phone, …)
  - Describe *possible executions* (aka. *execution traces*)
  - Describe *possible interactions during* execution
  - Describe interaction with an *environment*
  - Resulting *"value"* is not (necessarily) interesting

Concurrency is much *Harder*

- Harder than sequential programming:
  - Huge number of possible executions
  - Inherently non-deterministic
  - Parallelism conceptually harder
- Consequences:
  - Programs are harder to *write!*
  - Programs are harder to *debug!*
  - Errors are not always reproducible
  - New kinds of errors possible:
    - Deadlock, starvation, priority inversion, interference, …

Concurrency Problems

- ***Therac-25 Radiation Therapy***
  - ’85-’87
    - Massive overdoses (*6 deaths / amputations*)!



- ***Mars Pathfinder***
  - July ’97
    - Periodic resets (*on mars*)!

- ***Windows 95/98 w/ Device Drivers***
  - late ’90es
    - Dysfunction (*“blue screen of death”*)!

Concurrency Problems (cont'd)

- ***Mobile Phones***
  - '00-…
    - Freeze and odd behaviors (*really annoying*)!

- ***Cruise Control System Model***
  - '86 [Grady Booch]
    - Accellerated after car ignition (*car crashes*)!

- ***…***

…and what about?

- ***Air Plane Control System***
  - Dysfunction (*plane crash*)!

- ***Nuclear Powerplant Control System***
  - Core melt-down (*"China-syndrome"*)!

Problem: *System Development?*

- In the presence of all these errors:
  - deadlock, starvation, priority inversion, interference, anti-cooperation, un-intended execution traces, un-fairness, …
- How to…:
  - 1. …design a system that *"works"* ?
  - 2. …verify that the system is *"safe"* ?
  - 3. …verify that the system *"meets its specification"* ?

…and: *What does "works", "safe", and "to meet a specification" mean ?!?*

Solution: *Modelling*

- *"Models come to the rescue":*
  - *Create models* (~ *architecture, bridge construction, …*)

Note: *"Errors are much cheaper to commit in models than in implementations"*

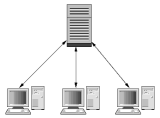
Dictionary: "model"

Webster's("model"):
Main Entry: **¹mod·el** Pronunciation: **'mä-d&l** Function: *noun* Etymology: Middle French *modelle,* from Old Italian *modello,* from (assumed) Vulgar Latin *modellus,* from Latin *modulus* small measure, from *modus*
1 *obsolete* **:** a set of plans for a building **2** *dialect British* **:** COPY**,** IMAGE **3 : structural design** <a home on the *model* of an old farmhouse> **4 : a usually miniature representation of something; *also* : a pattern of something to be made** **5 :** an example for imitation or emulation **6 :** a person or thing that serves as a pattern for an artist; *especially* **:** one who poses for an artist **7 :** ARCHETYPE **8 :** an organism whose appearance a mimic imitates **9 :** one who is employed to display clothes or other merchandise **:** MANNEQUIN **10 a :** a type or design of clothing / **b :** a type or design of product (as a car) **11 :** a description or analogy used to help visualize something (as an atom) that cannot be directly observed **12 : a system of postulates, data, and inferences presented as a mathematical description of an entity or state of affairs** **13 :** VERSION

- In this course (we use):
  - **3+4** : as in "***Model-based design***" (*designing a model of a concurrent system*)
  - **12** : as in "***Model-checking***" (*checking implementation against declarative (logic) specification*)

Modelling: *Level of Abstraction*

- Consider a *client/server system*:



higher level of abstraction

- *Extremely abstract* (*high level of abstraction*):
  - $\texttt{Universe} \stackrel{\text{def}}{=} \overline{\texttt{event}}.\texttt{Universe}$

- *Appropriate* (*level of abstraction*) ***for*** **…** :
  - $\texttt{Server} \stackrel{\text{def}}{=} \underline{\texttt{request}}.\overline{\texttt{process}}.\underline{\texttt{reply}}.\texttt{Server}$
    $\texttt{Client} \stackrel{\text{def}}{=} \texttt{calc}.\overline{\texttt{request}}.\underline{\texttt{wait}}.\underline{\texttt{reply}}.\texttt{Client}$
    $\texttt{Database} \stackrel{\text{def}}{=} \ldots$
- *Extremely concrete* (*low level of abstraction*):
  - $\texttt{NAND\_Gate} \stackrel{\text{def}}{=} \ldots$
    $\texttt{Transistor} \stackrel{\text{def}}{=} \ldots$
    $\texttt{Accumulator} \stackrel{\text{def}}{=} \ldots$
    $\ldots$
    $\texttt{Client} \stackrel{\text{def}}{=} \ldots$

Solution: *Modelling*

- *"Models come to the rescue":*
  - *Create models* (~ *architecture, bridge construction, …*)

Note: *"Errors are much cheaper to commit in models than in implementations"*

- *Formal modelling* (e.g., CCS) permits:
  - (Offline) *Reasoning* → *understanding*
  - (Runtime) *Testing* → *confidence*
  - (C-time) *Property Verification* → *safety*
  - (C-time) *Specification Verification* → *correctness*

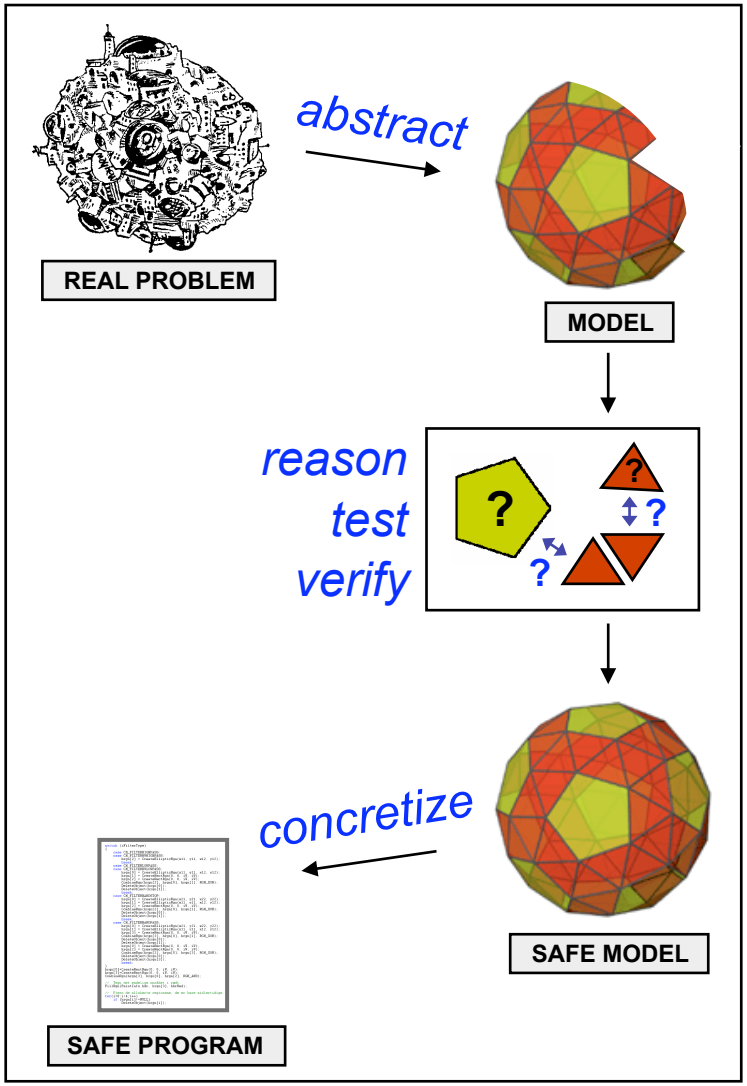
auto-mate

"Never send a human to do a machine's job"

-- A.Smith ('99)

Methodology: Model-based Design

- Design *abstract model*
- *Decompose* model
- Reason/Test/Verify model
  - individual parts and whole
- *Recompose* insights
  - make model safe
- Impl. *concrete program*



CCS: Why a *Calculus* (*pl.* Calculi)

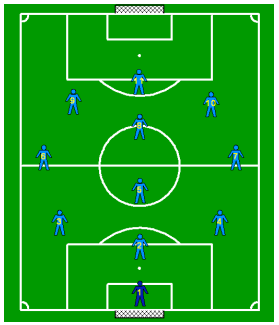
- Compositional:
  - $P \mid Q \approx P \parallel Q$
    - Break big things into (several) smaller things
- Algebraic:
  - $P + Q \equiv Q + P$, $P \mid Q \equiv Q \mid P$, ...
    - Intuitive ideal (also eases automated verification)
- Syntactic:
  - $[\text{PAR}_1]$ $\dfrac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q}$ and $[\text{PAR}_2]$ $\dfrac{Q \rightarrow Q'}{P \mid Q \rightarrow P \mid Q'}$ ...
    - Provide basis for programming languages

Parallel- vs. Concurrent Programming

The Football Match Analogy:

*"An analogy that one can make is with football**;*
- *the coach of the team is a parallel programmer while*
- *the referee is a concurrent programmer"* -- [P.Panangaden, '96]

*/ *interpret appropriately on either side of the Atlantic*



The Trainer (~ **the parallel programmer**):
-- Make sure my agents are performing "optimally"

- *Strategy*:
  - Optimal strategy for a particular goal
  - Use available resources efficiently

The Referee (~ **the concurrent programmer**):
-- Make sure what is happening is a soccer match

- *Safety*:
  - Conceptually *independent* players
  - Control interaction and "rules"

CALCULUS OF COMMUNICATING SYSTEMS

CCS: **C**alculus of **C**ommunicating **S**ystems

[Robin Milner, ’89]

Turing Award 1991

1) LCF (theorem pr.)
2) ML
3) CCS

April 2009

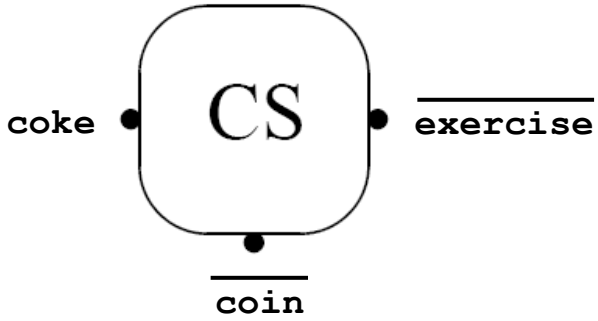
Concurrency and Communication

- Concurrency:
  - *Parallel processes* (construction '`P | Q`')
    - Abstract away (physical) processors
    - Abstract away diff. in real- vs pseudo-parallelism
- Communication:
  - *Process synchronization (aka. hand-shaking)*
    - Abstract away communication protocol
    - Abstract away actual values passed

Process *Interface*

- Example: a process modelling a *CS student:*

Interface:



- *Process name*: **CS**
- *Input action(s)*: { **coke** }
- *Output action(s)*: { $\overline{\texttt{coin}}$, $\overline{\texttt{exercise}}$ }

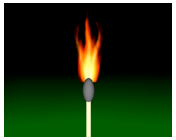
Behavior of the process described by a *CCS program*

The *Inactive* Process: "0"

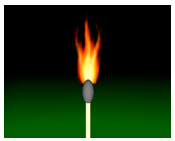
- *The inactive process*: 0
  - (aka. *"the zero process"* or *"the nil process"*)
    - Performs no action whatsoever!
  - Note that it offers:
    - the prototypical behavior of a *deadlocked process* (that cannot proceed any further in its execution)
- Example: 0

Action Prefixing: "α.P"

- *Action Prefixing*: `α.P`
  - Can perform action, α, after which it behaves like process, `P`
- Example(s):
  - Match: `strike.0`



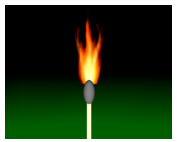
  - Complex match `take.strike.0`



Named Process: "K"

- *Named Process*: K
  - Behaves just like the (statically named) process, K

- Example(s):
  - $Match \stackrel{def}{=} strike.0$
  - $CokeDisp1 \stackrel{def}{=} coin.\overline{coke}.0$



Recursive Processes

- *Recursive Processes*

- Example:
  - $\texttt{Clock} \overset{\text{def}}{=} \texttt{tick.Clock}$

  - *Expanding* the definition we get:
    - $\texttt{Clock}$

$= \texttt{tick.Clock}$

$= \texttt{tick.tick.Clock}$

…

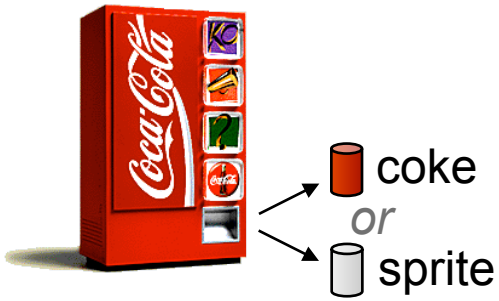
$= \texttt{tick.tick.tick. … .tick.Clock}$

…

Non-deterministic Choice: "P+Q"

- *Non-deterministic choice*: `P+Q`
  - Non-deterministic choice between processes `P` and `Q`
    - Initially has the capabilities of both `P` and `Q`; but performing an action from `P`, say, will pre-empt further execution of `Q`.

- Example:
  - $\texttt{Disp} \stackrel{\text{def}}{=} \texttt{coin.}(\overline{\texttt{coke}}\texttt{.Disp} + \overline{\texttt{sprite}}\texttt{.Disp})$



Parallel Composition: "P | Q"

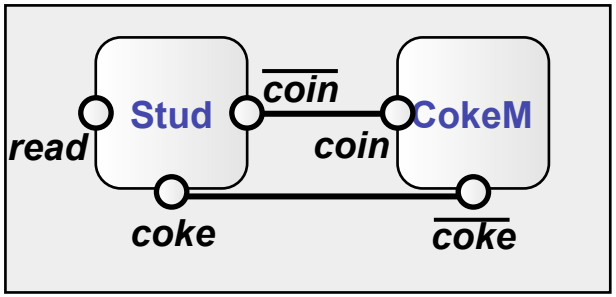
- *Parallel Composition*: `P | Q`
  - Any independent *interleavings* of processes `P` and `Q`
    - **Also:** *may communicate* (*hand-shake*): process `P` using input action, a; process `Q` corresponding output action, $\overline{a}$ (or vice versa)

- Example:
  - Student: $Stud \stackrel{def}{=} \overline{read}.\overline{coin}.coke.Stud$
  - Coke Machine: $CokeM \stackrel{def}{=} coin.\overline{coke}.CokeM$

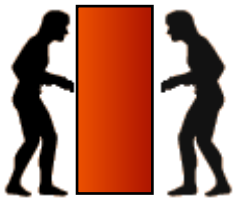
`CokeM | Stud`

Parallel Composition (cont'd)

- `Stud | CokeM`

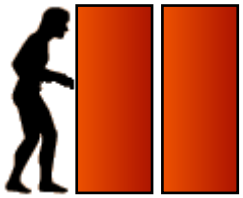


- `(Stud | CokeM) | Stud`



[<< ? >>]

- `(Stud | CokeM) | CokeM`

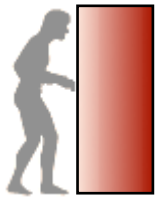


[<< ? >>]

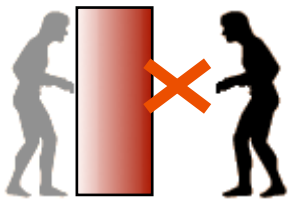
Restriction: "`P\a`"

- *Restriction (private name)*: `P\a`
  - Behaves just like `P`, except cannot make `a` or $\bar{a}$ actions (except within `P`)
    - Reminiscent of *local variables* (in *private scope*)

- Example:
  - `(Stud | CokeM) \ coin \ coke`



  - `((Stud | CokeM) \ coin \ coke) | Stud`



Action Relabelling: "P[f]"

- *Action Relabelling*: `P[f]`
  - Behaves like `P`, except that actions are *renamed* according to *action renaming function*, `f`
    - Permits *parameterized reuse* of processes

Note: *relabel inputs to inputs (and corresponding outputs to outputs)*

- Examples:
  - $\texttt{VendingMachine} \stackrel{\text{def}}{=} \texttt{coin}.\overline{\texttt{item}}.\texttt{VendingMachine}$
  - $\texttt{CokeMachine} \stackrel{\text{def}}{=} \texttt{VendingMachine}[\texttt{coke}/\texttt{item}]$
  - $\texttt{MarsMachine} \stackrel{\text{def}}{=} \texttt{VendingMachine}[\texttt{mars}/\texttt{item}]$

SYNTAX FOR *CCS*

April 2009

Input, output (and internal) action

- Actions:
  - $a \in A$ Set of *Channel Names* (input)
  - $\bar{a} \in \bar{A}$ Set of *Channel Co-Names* (output)
  - τ Special *silent* (invisible/internal) *action* tau
  - Note: inputs and outputs are *complementary*: $\bar{\bar{a}} = a$
    - Communication: *hand-shake* on `a` and $\bar{a}$ only (*no values*)
- Metavariables:
  - $a \in \mathcal{L} = A \cup \bar{A}$
  - $\alpha \in Act = \mathcal{L} \cup \{\tau\}$

CCS Syntax

- CCS Syntax:

$$P ::= 0 \mid \alpha.P \mid P+P \mid P|P \mid P\backslash a \mid P[f] \mid K$$

- "0" // inaction
- "$\alpha.P$" // action prefix
- "$P+P$" // non-deterministic choice
- "$P|P$" // parallel composition
- "$P\backslash a$" // restriction (private name)
- "$P[f]$" // action relabelling
- "K" // process variable

... where $X \stackrel{def}{=} P,\ Y \stackrel{def}{=} Q,\ \ldots$

Note: *restrictions on f*

$$f:Act \rightarrow Act$$

$$\forall a:\ f(\overline{a}) = \overline{f(a)} \quad \wedge\ f(\tau) = \tau$$

Alternative Syntax

- Alternative Syntax (that we may use):
  - *Abbreviate inaction termination*:
    - P for $P.0$ // obvious from context
  - *Parameterized sum*:
    - $\Sigma_{i \in I} P_i$ for $P_0 + P_1 + \ldots + P_n$
  - *Inactive process (as empty sum)*:
    - $\Sigma_{i \in \emptyset} P_i$ for 0
  - *Restriction (by set)*:
    - $P \setminus L$ for $P \setminus a_1 \setminus \ldots \setminus a_n$ $L=\{a_1,\ldots,a_n\}$

Algebraic Operator *Precedence*

- 1. Restriction and relabelling "`P\L`"
  "`P[f]`"
- 2. Action prefixing "`α.P`"
- 3. Parallel composition "`P|Q`"
- 4. Summation "`P+Q`"

- **Q:** How is "`R+a.P|b.Q\L`" then to be read **?**
  - **A:** "`R+((a.P)|(b.(Q\L)))`" **!**

Semantics (SOS) for CCS

April 2009

SOS for CCS

- *Structural Operational Semantics*:

A relation $\longrightarrow$ between processes indicating behaviour.

P $\xrightarrow{\alpha}$ **Q** denotes process **P** performing an action **α** and becoming process **Q**

SOS for CCS (2)

- *Structural Operational Semantics*:

$$[\text{DEF}]\ \frac{P \xrightarrow{\alpha} P'}{K \xrightarrow{\alpha} P'}\ K \stackrel{\text{def}}{=} P \qquad [\text{ACT}]\ \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad [\text{SUM}]\ \frac{P_j \xrightarrow{\alpha} P'_j}{\sum_{i\in I} P_i \xrightarrow{\alpha} P'_j}\ j \in I$$

$$[\text{COM}_1]\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad [\text{COM}_2]\ \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \qquad [\text{COM}_3]\ \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

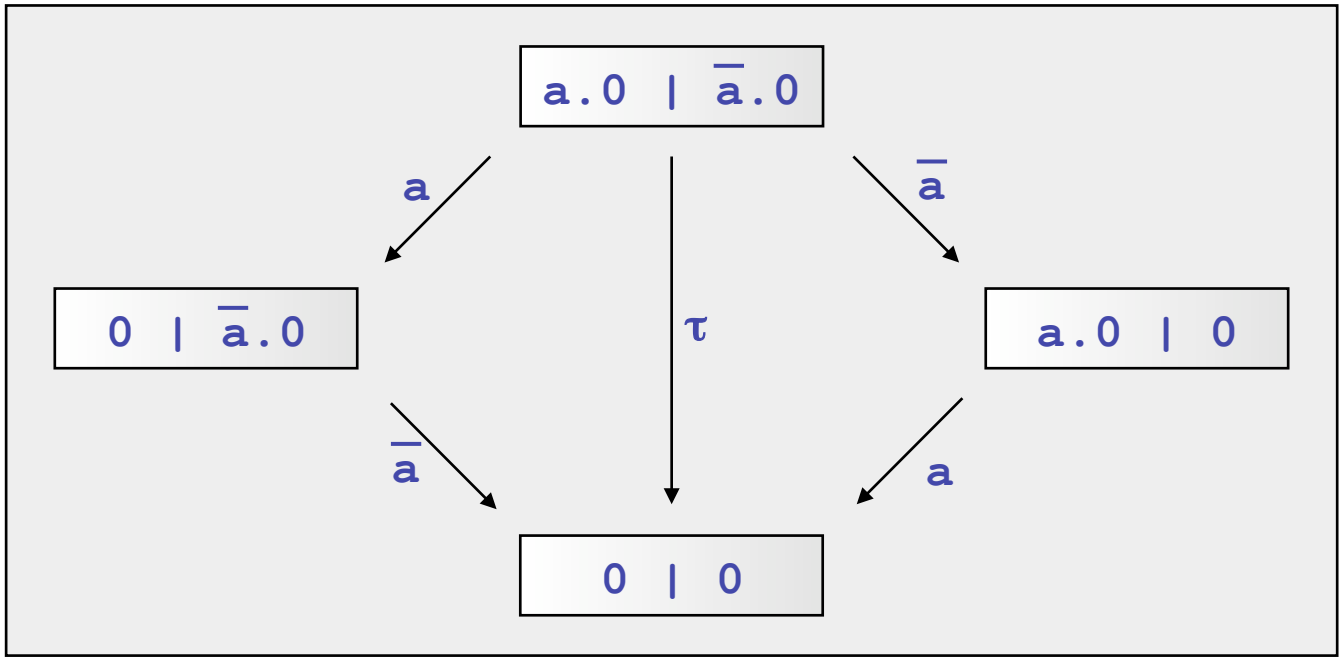
$$[\text{REN}]\ \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \qquad [\text{RES}]\ \frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L}\ \alpha, \bar{\alpha} \notin L$$

Q: why τ (tau) in communication "P | Q" (instead of propagating a or $\bar{a}$) **?**

τ ~ "the unobservable hand-shake"

Transition Diagram

- *Transition Diagram*:
  - A visualization of a *Labelled Transition System:*
    - Configurations annotated with processes (e.g. $a.0 \mid 0$)
    - Transitions annotated with actions (e.g. $\xrightarrow{a}$)



Example Derivation

- Assume:

$$A \overset{\text{def}}{=} a.A$$

- Consider:

$(b.0 \mid (A \mid \bar{a}.0))[c/a] \xrightarrow{c} ?$

$$[\text{DEF}]\frac{P \xrightarrow{\alpha} P'}{K \xrightarrow{\alpha} P'}\ K \overset{\text{def}}{=} P \qquad [\text{ACT}]\frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad [\text{SUM}]\frac{P_j \xrightarrow{\alpha} P'_j}{\sum_{i\in I} P_i \xrightarrow{\alpha} P'_j}\ j \in I$$

$$[\text{COM}_1]\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad [\text{COM}_2]\frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \qquad [\text{COM}_3]\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$[\text{REN}]\frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \qquad [\text{RES}]\frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L}\ \alpha, \bar{\alpha} \notin L$$

$$\dfrac{\dfrac{\dfrac{\dfrac{[\text{ACT}]\ \overline{a.A \xrightarrow{a} A}}{[\text{DEF}]\ \overline{A \xrightarrow{a} A}\ A \overset{\text{def}}{=} a.A}}{[\text{COM}_1]\ \overline{(A \mid \bar{a}.0) \xrightarrow{a} (A \mid \bar{a}.0)}}}{[\text{COM}_2]\ \overline{(b.0 \mid (A \mid \bar{a}.0)) \xrightarrow{a} (b.0 \mid (A \mid \bar{a}.0))}}}{[\text{REN}]\ \overline{(b.0 \mid (A \mid \bar{a}.0))[c/a] \xrightarrow{c} (b.0 \mid (A \mid \bar{a}.0))[c/a]}}$$

Derivation Sequence and Traces

A **derivation sequence** (or **execution**) is a sequence of derivations:

$$P \xrightarrow{a1} P_2 \text{ and } P_2 \xrightarrow{a2} P_3 \quad \ldots \quad P_n \xrightarrow{an} Q$$

or (simpler writing)

$$P \xrightarrow{a1} P_2 \xrightarrow{a2} P_3 \xrightarrow{a3} \ldots \xrightarrow{a(n-1)} P_n \xrightarrow{an} Q$$

a1 a2 a3 … an is a **trace** of process **P**

Derivation Sequence and Traces (2)

The process:

```
a  | ‘a.b
```

has the following traces:

```
a
```

```
‘a b
```

```
‘a a b
```

```
‘a b a
```

```
a ‘a b
```

```
τ
```

```
τ b
```

Derivation Sequence and Traces (3)

The process:

$A =_{def} a \mid \overline{a}.A$

has the following traces:

a

$a\ \overline{a}\ \overline{a}\ \overline{a}\ \overline{a}\ a\ a\ a$

$\tau\ \overline{a}\ \overline{a}\ \overline{a}\ a\ a\ a$

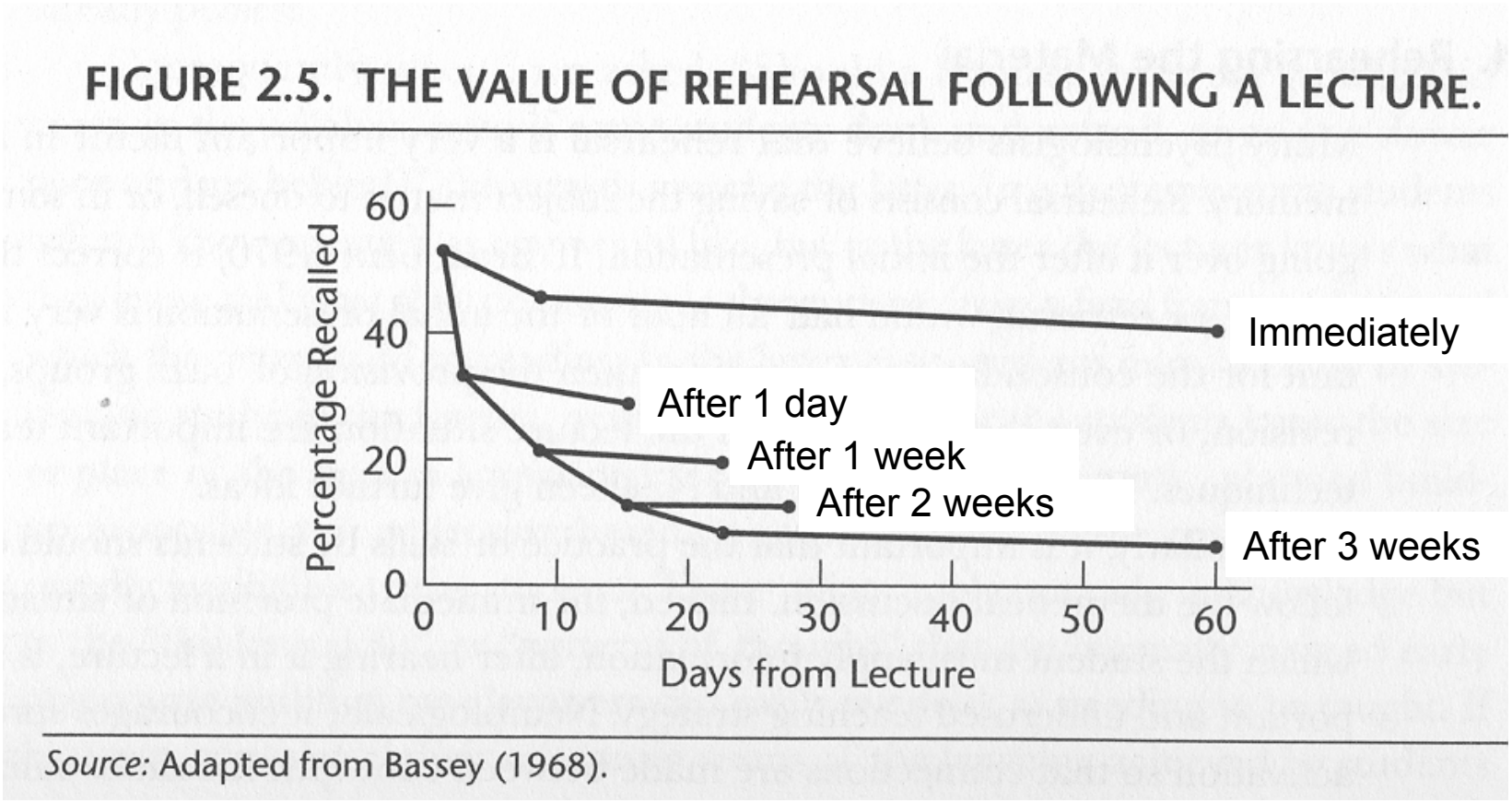
...

What we have been through today

- Introduction to Part 2
- Concurrency vs. Sequentiality
- ***CCS***: *"Calculus of Communicating Systems"*
  - *By-example* *(one construction at a time)*
- Syntax of ***CCS***
  - 7 linguistic constructions(!)
- Semantics of ***CCS***
  - 1 page(!)

"Three minutes paper"

- Please spend three minutes writing down the most important things that you have learned today (***now***).



Exercise 1

Draw the transition graphs for the following:

(a) $\alpha.0$

(b) $\mathtt{Cl}_1 \stackrel{\text{def}}{=} \mathtt{tick.tock.Cl}_1$

(c) $\mathtt{Cl}_2 \stackrel{\text{def}}{=} \mathtt{tick.tick.Cl}_2$

(d) $\mathtt{Cl}_3 \stackrel{\text{def}}{=} \mathtt{tick.Cl}$ where $\mathtt{Cl} \stackrel{\text{def}}{=} \mathtt{tick.Cl}$

Exercise 2

Draw the transition graphs for the following:

$$\mathtt{Ven} \stackrel{\text{def}}{=} \mathtt{2p.Ven_b} + \mathtt{1p.Ven_l}$$

$$\mathtt{Ven_b} \stackrel{\text{def}}{=} \mathtt{big.collect_b.Ven}$$

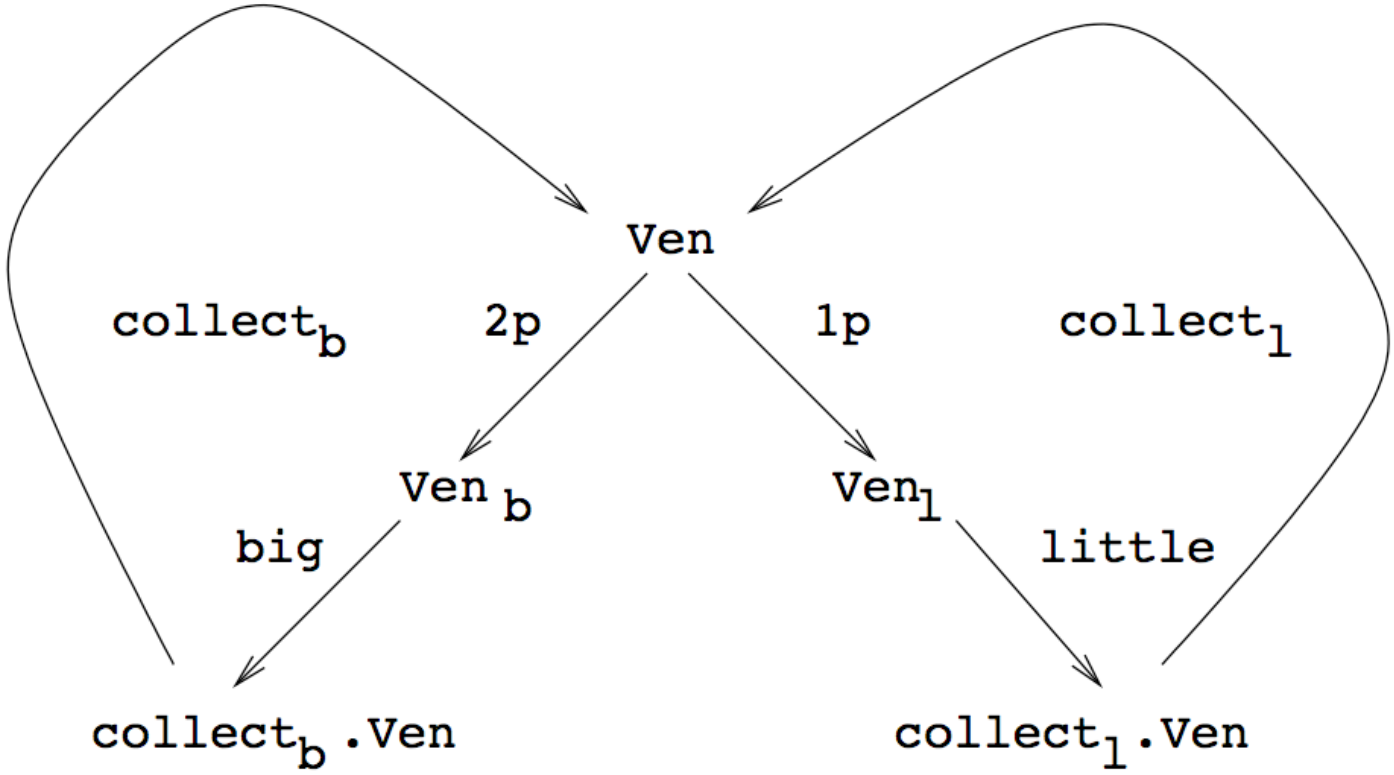
$$\mathtt{Ven_l} \stackrel{\text{def}}{=} \mathtt{little.collect_l.Ven}$$

Exercise 2 (2)

$$\mathtt{Ven} \stackrel{\text{def}}{=} \mathtt{2p.Ven_b} + \mathtt{1p.Ven_l}$$

$$\mathtt{Ven_b} \stackrel{\text{def}}{=} \mathtt{big.collect_b.Ven}$$

$$\mathtt{Ven_l} \stackrel{\text{def}}{=} \mathtt{little.collect_l.Ven}$$



Exercise 3

$$\mathtt{Ct}_0 \stackrel{\text{def}}{=} \mathtt{up}.\mathtt{Ct}_1$$

$$\mathtt{Ct}_{i+1} \stackrel{\text{def}}{=} \mathtt{up}.\mathtt{Ct}_{i+2} + \mathtt{down}.\mathtt{Ct}_i$$

Write a derivation for Ct_3

Exercise 4

Give all derivations and transition diagram for the following CCS processes:

A $=_{def}$ tau . 0

B $=_{def}$ a . 0 + 'a . 0

C $=_{def}$ a . 0 | 'a . 0

D $=_{def}$ C \ a

Exercise 5

a) Give all possible executions (transition diagram) for the GossipingGirls.
The process GossipingGirls models three girls, Alice, Betty, and Carla who communicate with each other along private (cell-phone) channels named **alice**, **betty**, and **carla**:

1) **Alice** $=_{def}$ **alice . 'betty . Alice**

2) **Betty** $=_{def}$ **betty . 'carla . Betty**

3) **Carla** $=_{def}$ **carla . 'alice . Carla**

i.e., (1) wait for someone to call my phone (and tell me something); then call (and tell) betty; (2) wait for someone to call my phone (and tell me something); then call (and tell) carla; and (3) wait for someone to call my phone (and tell me something); then call (and tell) alice.

GossipingGirls $=_{def}$ **(Alice | Betty | Carla) \ {alice,betty,carla}**

Note. Of course, these "communication channels" are private ;)

Exercise 5 (2)

b) What can happen if now we have (a new process), David, call any one of them?:

$$\textbf{David} =_{\textbf{def}} \textbf{'alice . 0 + 'betty . 0 + 'carla . 0}$$

i.e., call **alice**, **betty**, or **carla** (and tell her something)

$$\textbf{NowWhatHappens} =_{\textbf{def}} \textbf{(Alice | Betty | Carla | David) \ \{alice,betty,carla\}}$$

Exercises 6

1) Define a more rational vending machine than **Ven** that allows to choose **big** if two 1p coins are entered, and **little** to be chosen twice after a 2p coin is deposited (look at **big** and **little** as buttons). Give the transition graph.

2) Define a process **Change** that describes a change-making machine with one input port and one output port, that is capable initially of accepting either a 10p or a 5p, and can then dispense any sequence of 1p, 2p, 5p and 10p coins, the sum of whose values is equal to that of the coin accepted, before returning to its initial state.