# Concurrency and Process Logics

Ken Stevens

# Verification

Cannot understand verification
without understanding concurrency

# Concurrency Challenge

Concurrency is major challenge

- practically: plethora of concurrent implementations
  - architectures
    - Mayfly, transputer, reduction machines, microprocessors, ...
  - Programming
    - Algorithms, actors, domains, threads, ...
- Theoretical: model and prove properties
  - architectures
    - PRAM model, ...
  - Programming
    - CSP, Occam, CCS, ...

# Concurrency

Interesting dichotomy exists:

- more parallel architectures than languages
  - inherent parallelism in hardware
  - inherent difficulty in parallel programming
    - perhaps Verilog is most used parallel programming language?

Humans don't think in parallel

- missed off-ramp on freeway

# Concurrency Models

Plethora of variations can create confusion

Theory is useful

- provides clarity and understanding
- classifies/groups similarities/variations
- applicable to real “practice” (but sterile)

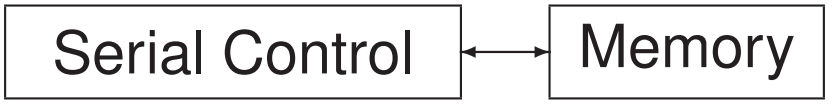
# Process Logics

- apply theory to concurrency
- focus on interaction of theory and practice
- still a large breach in both

There are other theories

We will use simplest process logic: CCS

# Von Neumann Architecture




Program is a mathematical function over memory state

- given a start state, can calculate solution

Assumes subservient memory, under sole control of single serial thread.

**Equality** – for now nebulous –
semantic equality results in same results.

# Von Neumann Architecture

Program 1

$x := 1;$

eval $x$

Program 2

$x := 0;$

$x := x + 1$

eval $x$

Are models equivalent?

# Von Neumann Architecture

Program 1

$x := 1;$

eval $x$

Program 2

$x := 0;$

$x := x + 1$

eval $x$

Are models equivalent: yes under Von-Neumann model

What about concurrent model?

# Von Neumann Architecture

Program 1

$x := 1;$

eval $x$

Program 2

$x := 0;$

$x := x + 1$

eval $x$

Are models equivalent: yes under Von-Neumann model

What about concurrent model: no.

# Concurrent architecture

| Program 1 | Program 2 |
| --- | --- |
| $x := 1;$ | $x := 0;$ |
| eval $x$ | $x := x + 1$ |
| | eval $x$ |

Assume daemon: $x := 1;$, or concurrent operation of programs. $x$ can be 1 or 2.

Look at results of interactions of concurrent systems

- who and when memory location $x$ is accessed

# Concurrent architectures

key: memory no longer a “slave”;

memory now 1st class agent with behavior that needs modeling.

“He who serves two masters serves none”

Think asynchronously: each model is own master.

# Calculus of Communicating Systems

Milner's CCS:

- semantic theory in which interaction and communication is central idea.
- achieves smooth mathematical treatment

Hoar's CSP:
same primitive notion for different purpose

- the ability to derive other primitives
  - semaphores and monitors (themselves taken as primitives)

  from one general primitive

Both converged to same point of the pivotal nature of interaction and communication between behaviors.

# Process Logics

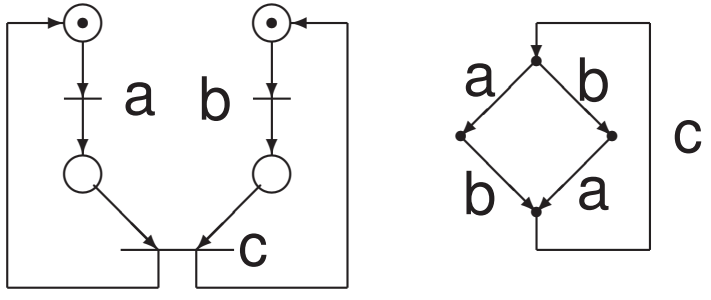
Evidence that this notation is a pivotal one:

- serial interacting agents
- communication primitive

# Other concurrency theories

- Petri nets
  - (I learned this from one of Carl Petri's contemporaries)
  - independence (parallel) of actions
  - place transition nets / signal transition graphs

True Concurrency vs interleaving concurrency




- true concurrency
- first good theory of concurrency
  - still used today for most asynchronous modeling

# Petri Nets

Benefits:

1. causality directly represented
   - virtue of places
   - preconditions

Problems:

1. Structural issue not addressed
   - results in serious drawbacks
     - not compositional operator defined
     - not able to hide actions
   - attempted to model with “Snippets” and other “warts”
2. no general theory of equality or canonical representation

# Process Logics

1. Interleaving semantics
   - looses causality
     - becomes an “observational” or “existential” theory
     - (turns weakness into a strength)
     - CCS not even directly observable (due to $\tau$)
2. Direct Structural aspect
   - independent parallel “process” or “agent” set
     - each process has localized behavior
   - clear definition of “interaction” or “synchronization”
   - ability to hide non-critical detail and hierarchy

*results in tractable theory which focuses on local behavior and interaction.*

*no theory exists that directly supports both causality and structure*

# Process Logics

Processes *interact*

- core is an **algebra**

  *various systems or branches of mathematics concerned with the properties and relationships of abstract entities manipulated in symbolic form under operations often analogous to those of arithmetic*.

  - everything *cannot* be done with algebra
  - still have need for **higher order logic** or *theorem proving*

- "Calculus" of communicating systems

  *a method of computation or calculation in a special notation (as of logic or symbolic logic)*

  - implies use of logic
  - temporal logic applies here

# SECTION 2

# CCS

Communication and concurrency are complex notations formalized in process logics.

| | |
|---|---|
| concurrency: | *Independence or locality of behavior* |
| communication: | *unifies independent models into a single system* |

Underlying notion of identity:

- agents: identifiable behaviors that persist in time

# CCS

Systems consist of discrete actions.

Actions come in two independent classes:

1. dependency
   - actions can be **independent**
   - or constrained by **communication** or **synchronization**
2. observability
   - actions can be externally observed
   - actions are unobservable, but their results can be observed
     - unobservable action: $\tau$

# CCS

Agents represent the current **state**

State to state transitions occur via **actions**

Hence it is convenient to write these as:

$$P \xrightarrow{\alpha} Q$$

# Notation

| | |
|---|---|
| Agents: $\mathcal{P}$ | $P, Q, \ldots$ range over $\mathcal{P}$ |
| Names: $\mathcal{A}$ | $a, b, \ldots$ range over $\mathcal{A}$ |
| CoName: $\overline{\mathcal{A}}$ | $\overline{a}, \overline{b}, \ldots$ range over $\overline{\mathcal{A}}$ |
| Labels: $\mathcal{L}$ | $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$ |
| unobservable action: $\tau$ | |
| Actions: *Act* | $Act = \mathcal{L} \cup \{\tau\}$ |
| Actions: *Act* | $\alpha, \beta, \ldots$ range over *Act* |
| Agent Expressions: $\mathcal{E}$ | $E, F, \ldots$ range over $\mathcal{E}$ |
| Indexing sets: | $\{E_i : i \in I\}$ |

# Agent Expressions

| | | |
|---|---|---|
| Nil | does nothing | $(\sum_{i \in 0} E_i)$ |
| Constant | $A = P$; | (infinite behavior via recursion) |
| prefixing | $\alpha.E$ | $(\alpha \in Act)$ |

Now we have the power to define simple agents:

Match = strike.burn.Nil;

CooCoo = tick.tock.CooCoo;

# Choice

Summation $\sum_{i \in I} E_i$

Summation implements non-deterministic choice.

- interleaving parallelism

We now have ability to express more powerful agents:

C-element can now be derived from state diagram earlier:

C-element = a.b.'c.C-Element + b.a.'c.C-Element

(note: 'a is textual representation of $\overline{a}$)

# Synchronization

These are the synchronizing primitives:

| | | |
|---|---|---|
| Composition | $E_1 \mid E_2$ | |
| Restriction | $E \backslash \mathcal{L}$ | ($\mathcal{L}$ should be $\mathcal{A}$) |
| Relabeling | $E[f]$ | |

$f$ is a relabeling function mapping $\mathcal{L}$ to $\mathcal{L}$ such that $f(\overline{\alpha}) = \overline{f(\alpha)}$ and $f(\tau) = \tau$

signal complementation $\overline{\overline{a}} = a$

# Composition


$$A \stackrel{\text{def}}{=} a.A' \qquad B \stackrel{\text{def}}{=} c.B'$$

$$A' \stackrel{\text{def}}{=} \overline{c}.A \qquad B' \stackrel{\text{def}}{=} \overline{b}.B$$

Now consider:

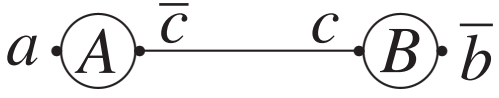

What is the difference between $\overline{c}$ and $c$?

# Composition


$$A \stackrel{\text{def}}{=} a.A' \qquad B \stackrel{\text{def}}{=} c.B'$$

$$A' \stackrel{\text{def}}{=} \overline{c}.A \qquad B' \stackrel{\text{def}}{=} \overline{b}.B$$

Now consider:




What is the difference between $c$ and $\overline{c}$?

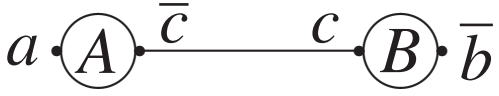
Labels and Colabels (inputs and outputs)

# Composition


$$A \stackrel{\text{def}}{=} a.A' \qquad B \stackrel{\text{def}}{=} c.B'$$

$$A' \stackrel{\text{def}}{=} \overline{c}.A \qquad B' \stackrel{\text{def}}{=} \overline{b}.B$$

Now consider:



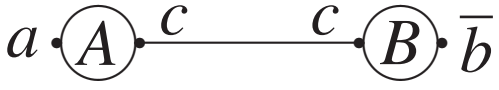

Can $\overline{c}$ fire *independently* in the lower graph?

# Composition


$$A \stackrel{\text{def}}{=} a.A' \qquad B \stackrel{\text{def}}{=} c.B'$$

$$A' \stackrel{\text{def}}{=} \overline{c}.A \qquad B' \stackrel{\text{def}}{=} \overline{b}.B$$

Now consider:




Can $\overline{c}$ fire *independently* in the lower graph?

YES!

Since $A' \xrightarrow{\overline{c}} A$, we infer $A' \mid B \xrightarrow{\overline{c}} A \mid B$

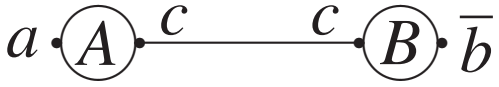
This does **not** indicate communication between $A$ and $B$

# Composition


$$A \stackrel{\text{def}}{=} a.A' \qquad B \stackrel{\text{def}}{=} c.B'$$

$$A' \stackrel{\text{def}}{=} \overline{c}.A \qquad B' \stackrel{\text{def}}{=} \overline{b}.B$$

Now consider:



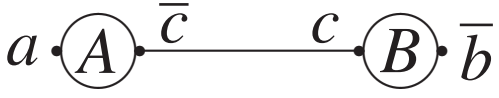

Can $c$ and $\overline{c}$ **handshake communicate** in the lower graph?

# Composition


$$A \stackrel{\text{def}}{=} a.A' \qquad B \stackrel{\text{def}}{=} c.B'$$

$$A' \stackrel{\text{def}}{=} \overline{c}.A \qquad B' \stackrel{\text{def}}{=} \overline{b}.B$$

Now consider:




Can $c$ and $\overline{c}$ **handshake communicate** in the lower graph?

YES!

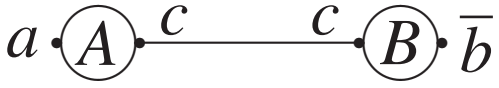
Since $A' \xrightarrow{\overline{c}} A$, and $B \xrightarrow{c} B'$, we infer $A' \mid B \xrightarrow{?} A \mid B'$

# Composition


$$A \stackrel{\text{def}}{=} a.A' \qquad B \stackrel{\text{def}}{=} c.B'$$

$$A' \stackrel{\text{def}}{=} \overline{c}.A \qquad B' \stackrel{\text{def}}{=} \overline{b}.B$$

Now consider:




Since $A' \xrightarrow{\overline{c}} A$, and $B \xrightarrow{c} B'$, we infer $A' \mid B \xrightarrow{?} A \mid B'$

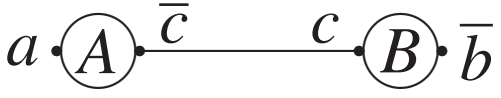
When we have handshake communication what is the '?' action?

# Composition


$$A \stackrel{\text{def}}{=} a.A' \qquad B \stackrel{\text{def}}{=} c.B'$$

$$A' \stackrel{\text{def}}{=} \overline{c}.A \qquad B' \stackrel{\text{def}}{=} \overline{b}.B$$

Now consider:




Since $A' \xrightarrow{\overline{c}} A$, and $B \xrightarrow{c} B'$, we infer $A' \mid B \xrightarrow{?} A \mid B'$

When we have handshake communication what is the '?' action?

$\tau$ – the invisible internal action.

*The $\tau$ action has semantic properties in CCS that are significantly different than other process logics, labeled transition systems (LTS), or concurrency models.*

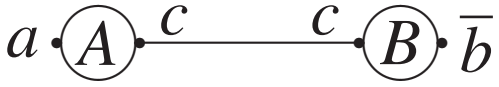
# Composition



$$A \stackrel{\text{def}}{=} a.A' \qquad B \stackrel{\text{def}}{=} c.B'$$

$$A' \stackrel{\text{def}}{=} \overline{c}.A \qquad B' \stackrel{\text{def}}{=} \overline{b}.B$$

Now consider:



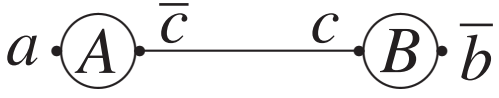

Can $\overline{c}$ communicate with other agents in the system that can do the $c$ action?

# Composition


$$A \stackrel{\text{def}}{=} a.A' \qquad B \stackrel{\text{def}}{=} c.B'$$

$$A' \stackrel{\text{def}}{=} \overline{c}.A \qquad B' \stackrel{\text{def}}{=} \overline{b}.B$$

Now consider:




Can $\overline{c}$ communicate with other agents in the system that can do the $c$ action?

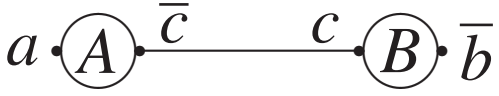
YES!! This is a primary means of specifying semaphores, nondeterminism, etc. in a system!

# Composition


$$A \stackrel{\text{def}}{=} a.A' \qquad B \stackrel{\text{def}}{=} c.B'$$

$$A' \stackrel{\text{def}}{=} \overline{c}.A \qquad B' \stackrel{\text{def}}{=} \overline{b}.B$$

Now consider:



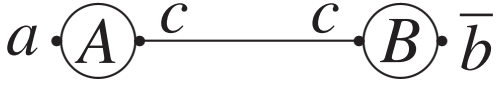

Can we **force** the communication to occur? How do we prevent the **independent** $\overline{c}$ and $c$ actions from occurring?

# Composition


$$A \stackrel{\text{def}}{=} a.A' \qquad B \stackrel{\text{def}}{=} c.B'$$

$$A' \stackrel{\text{def}}{=} \overline{c}.A \qquad B' \stackrel{\text{def}}{=} \overline{b}.B$$

Now consider:




Can we **force** the communication to occur? How do we prevent the **independent** $\overline{c}$ and $c$ actions from occurring?

YES! – this is the purpose of the **restriction** operator! The operator $(A \mid B)\backslash\{c\}$ will restrict the $c$ and $\overline{c}$ actions from occurring in the composed process.

# Transitional Semantics

A labeled transition system LTS $= (S, T, \{\xrightarrow{t} : t \in T\})$

S: state set

T: Transition label set

Transition relation: $\xrightarrow{t} \subseteq S \times S \; \forall t \in T$

Also:

S ::= $\mathcal{E}$

T ::= *Act*

Semantics for agent expressions $\mathcal{E}$ consist in the definition of each transition relation over $\mathcal{E}$.

# Simple Examples

Sender $\longrightarrow$ (medium) $\longrightarrow$ Receiver

- sender can send data at any time
- when ether not empty, receiver can accept at any time

Medium: ether
non-order-preserving
bounded buffer
buffer
shared memory
...

Must be able to represent them all.

Just like Von-Neumann example and parallel computer – *the medium* just like the memory *can have its own arbitrary behavior*, and thus can be represented as an independent agent!

# Bounded FIFO

```
FIFO      = in(x).FIFO'(x);
FIFO'(x)  = 'out(x).FIFO;
```

Equivalent definition:

```
FIFO = in(x).'out(x).FIFO;
```

What of $F \circ F \circ F$? (composition?)

Exercise 1, page 18 of Milner

# SECTION 3

# Review

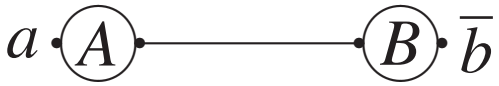
$$
\begin{array}{lll}
\mathcal{E} ::= & A & \text{constant} \\
\mid & \alpha.E & \text{prefixing} \\
\mid & \sum_{i \in I} E_i & \text{summation} \\
\mid & E_1 \mid E_2 & \text{composition} \\
\mid & E[f] & \text{relabeling} \\
\mid & E \backslash \mathcal{L} & \text{restriction}
\end{array}
$$

$$\text{LTS} = (S, T, \{\xrightarrow{t} : t \in T\})$$

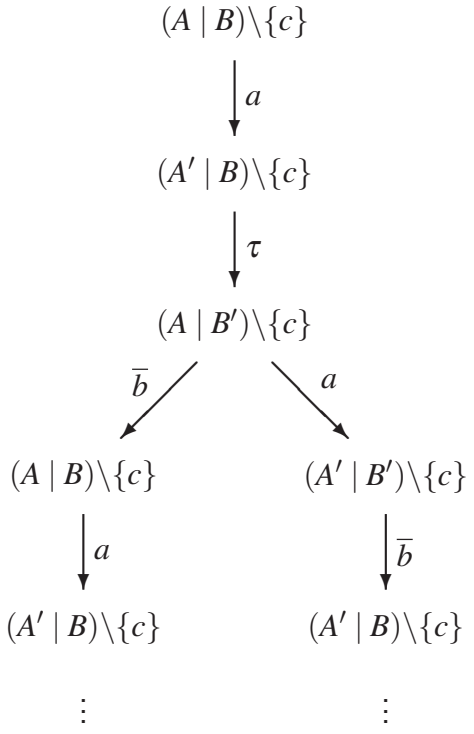
In composition, a label and colabel interact to form a single indivisible communication action $\tau$.

# Derivation Tree

For the process:

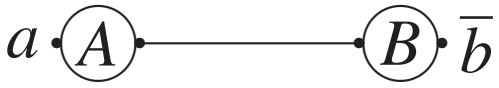



we can create the derivation tree:

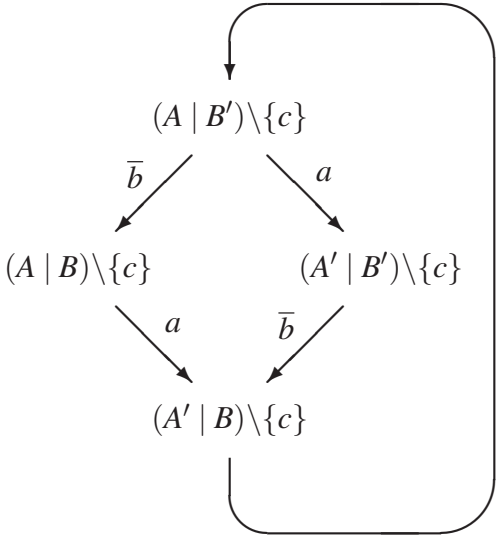


# Derivation Tree

For the process:




The cyclic nature allows us to represent this behavior as a *transition graph*:



# Internal Actions

The internal action $\tau$

1. has no complement
2. cannot communicate with other actions
3. is not directly observable

We want to define systems as equivalent if they can perform the same pattern of *external* actions.

This results in the ability to abstract:

Should $P \xrightarrow{\tau} P_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} P_n$

be equal to $P \xrightarrow{\tau} P_n$??

# Internal Actions

The internal action $\tau$

1. has no complement
2. cannot communicate with other actions
3. is not directly observable

We want to define systems as equivalent if they can perform the same pattern of *external* actions.

This results in the ability to abstract:

Should $P \xrightarrow{\tau} P_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} P_n$

be equal to $P \xrightarrow{\tau} P_n$??

YES!! We will define the transitional semantics and rules that show this to be the case.

# Transitional Semantics

Transitional semantics of
LTS $= (S, T, \{\xrightarrow{t} : t \in T\})$

These come in form of:

RULE $\overset{\text{name}}{\longrightarrow}$ Act

Top of the bar is the **hypothesis**

Bottom of bar is the **conclusion**

Act $\dfrac{}{\alpha.E \xrightarrow{\alpha} E}$ Prefixing

# Transitional Semantics

$$\text{Sum}_j \quad \boxed{\frac{E_j \xrightarrow{\alpha} E'_j}{\sum_{i\in I} E_i \xrightarrow{\alpha} E'_j} \quad (j \in I)} \quad \text{Summation}$$

- case $I = 0$, result is $\boxed{0}$ or $\boxed{\text{NIL}}$
- case $I = \{1,2\}$
  $\sum_{i\in\{1,2\}} E_i$ is written $E_1 + E_2$ and has transition rules

$$\frac{E_1 \xrightarrow{\alpha} E'_1}{E_1 + E_2 \xrightarrow{\alpha} E'_1} \qquad \text{and} \qquad \frac{E_2 \xrightarrow{\alpha} E'_2}{E_1 + E_2 \xrightarrow{\alpha} E'_2}$$

# Example

C-element defined as:

```
C = a.b.'c.C + b.a.'c.C;
```

rules are

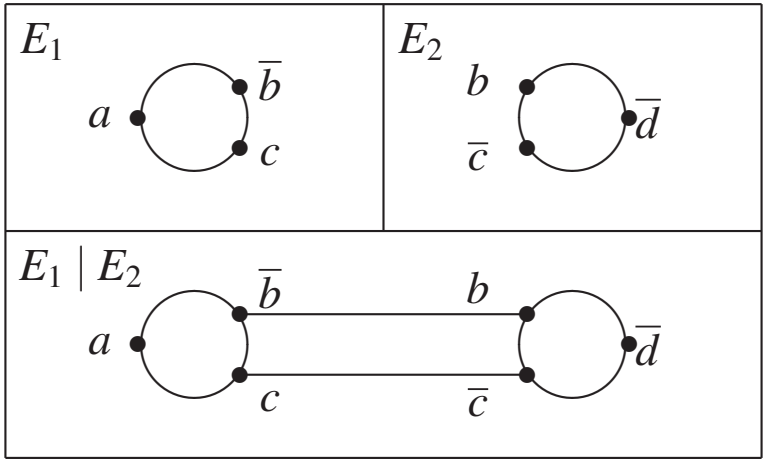
$$\frac{\texttt{a.b.'c.C} \xrightarrow{\texttt{a}} \texttt{b.'c.C}}{\texttt{a.b.'c.C + b.a.'c.C} \xrightarrow{\texttt{a}} \texttt{b.'c.C}}$$

and

$$\frac{\texttt{b.a.'c.C} \xrightarrow{\texttt{b}} \texttt{a.'c.C}}{\texttt{a.b.'c.C + b.a.'c.C} \xrightarrow{\texttt{b}} \texttt{a.'c.C}}$$

# Transitional Semantics

Parallel Composition




Names joined to corresponding conames

$E_1 | E_2$ has three rules:

# Transitional Semantics

$$\text{Com}_1 \quad \frac{E_1 \xrightarrow{\alpha} E_1'}{E_1 \mid E_2 \xrightarrow{\alpha} E_1' \mid E_2}$$

$$\text{Com}_2 \quad \frac{E_2 \xrightarrow{\alpha} E_2'}{E_1 \mid E_2 \xrightarrow{\alpha} E_1 \mid E_2'}$$

$$\text{Com}_3 \quad \frac{E_1 \xrightarrow{\bar{l}} E_1' \qquad E_2 \xrightarrow{l} E_2'}{E_1 \mid E_2 \xrightarrow{\tau} E_1' \mid E_2'}$$

$l \in \mathcal{L}$

if $l$ is $\overline{b}$ then $\bar{l}$ is $\overline{\overline{b}} = b$

if $l$ is $c$ then $\bar{l}$ is $\overline{c}$

# Transitional Semantics

$\tau$ is the unobservable internal action

- $\tau$ is neither a name or coname
- $\overline{\tau}$ is not a well-formed action expression

Note:

- In $\text{Com}_3$ we get $\tau$ whether $l$ or $\bar{l}$ give rise to the communication.
- We can not tell from $\tau$ exactly what goes on inside $E_1 \mid E_2$
- Likewise, we cannot differentiate between the $b$ and $c$ tau transitions
  - *unless* it creates a difference in externally observable behavior
- only single internal transition label necessary
- **most (all?) other languages use a *hiding* semantics!**

# Transition Semantics

SORTS:

**Definition:** for any $L \subseteq \mathcal{L}$, if the actions of $P$ and all its derivatives lie in $L \cup \{\tau\}$ then we say $P$ *has sort* $L$, or $L$ *is a sort of* $P$, and write $P{:}L$.

**Proposition:** For every $E$ and $L$, $L$ is a sort of $E$ if and only if, whenever $E \xrightarrow{\alpha} E'$, then

1. $\alpha \in L \cup \{\tau\}$
2. $L$ is a sort of $E'$

# Transitional Semantics

For $E$ with sort $K$ and $F$ with sort $L$

(This is used for hardware modeling and broadcast communication)

$$\text{Conj}_1 \quad \frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F} \quad (\alpha \notin L)$$

$$\text{Conj}_2 \quad \frac{F \xrightarrow{\alpha} F'}{E \mid F \xrightarrow{\alpha} E \mid F'} \quad (\alpha \notin K)$$

$$\text{Conj}_3 \quad \frac{E \xrightarrow{\bar{l}} E' \qquad F \xrightarrow{l} F'}{E \mid F \xrightarrow{\tau} E' \mid F'} \quad (l \in K \cap L)$$

(We can allow $\tau$ to be $\bar{l}$ for circuit semantics, and can allow $l$ to communicate with $l$ as shared inputs.)

# Relabeling

$f$ is relabeling function, so $f$: *Act* $\longrightarrow$ *Act*

- $f(\tau) = \tau$
- $f(l) \in \mathcal{L}$ when $l \in \mathcal{L}$
- $f(\bar{l}) = \overline{f(l)} \; \forall \, l \in \mathcal{L}$

Rel

$$\frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$$

# Relabeling

Notation:

when $f(l_i) = l'_i . (i = 1, \ldots, n)$

$f(l) = l$ if $l \neq l_1, \ldots, l_n$

written $E[f]$ as $E[l'_1/l_1, \ldots, l'_n/l_n]$

# Relabeling

Problems exist without restriction:

```
FIFO = in.'out.FIFO
```

♣:

```
FIFO | FIFO
```

♠:

```
FIFO[mid/out] | FIFO[mid/in]
```

Draw transition graph for ♣

Draw derivation tree for ♠

Note problem of independent actions!

# Restriction

$L \subseteq \mathcal{L}$

Res $\dfrac{E \xrightarrow{\alpha} E'}{E\backslash L \xrightarrow{\alpha} E'\backslash L} \quad (\alpha, \overline{\alpha} \notin L)$

- $E\backslash L$ has the same $\tau$ transitions as $E$ since $\tau \notin \mathcal{L}$
- Syntax of some tools assumes $L \notin \overline{\mathcal{A}}$, but $L \in \mathcal{A}$

# Hiding

Hiding is the semantics for internal signals for *most* process logics and LTS

$L \subseteq \mathcal{L}$

$$\text{Hid}_1 \quad \frac{E \xrightarrow{\alpha} E'}{E/L \xrightarrow{\alpha} E'/L} \quad (\alpha, \overline{\alpha} \notin L)$$

$$\text{Hid}_2 \quad \frac{E \xrightarrow{l} E'}{E/L \rightarrow E'/L} \quad (l \vee \bar{l} \in L)$$

Does this result in $E$ and $E'$ being equivalent?

# Hiding

Hiding is the semantics for internal signals for *most* process logics and LTS

$L \subseteq \mathcal{L}$

$$\text{Hid}_1 \quad \frac{E \xrightarrow{\alpha} E'}{E/L \xrightarrow{\alpha} E'/L} \quad (\alpha, \overline{\alpha} \notin L)$$

$$\text{Hid}_2 \quad \frac{E \xrightarrow{l} E'}{E/L \rightarrow E'/L} \quad (l \vee \bar{l} \in L)$$

Does this result in $E$ and $E'$ being equivalent?

YES

# Example

Now try this FIFO:

```
FIFO = in.'out.FIFO

(FIFO[mid/out] | FIFO[mid/in]) \{ mid }
```

Make transition Graph

Make Derivation Tree

# Examples

1. inverter
2. buffer
3. toggle
4. fork
5. merge
6. semaphore
7. mutex element