Concurrency and Process Logics

Ken Stevens
Verification

Cannot understand verification
without understanding concurrency
Concurrent 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
\[ x := 1; \]
\[ \text{eval } x \]

Program 2
\[ x := 0; \]
\[ x := x + 1 \]
\[ \text{eval } x \]

Are models equivalent: yes under Von-Neumann model

What about concurrent model?
Von Neumann Architecture

Program 1
\[ x := 1; \]
\[ \text{eval } x \]

Program 2
\[ x := 0; \]
\[ x := x + 1 \]
\[ \text{eval } x \]

Are models equivalent: yes under Von-Neumann model
What about concurrent model: no.
Concurrent architecture

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
   - loses 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
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: $\mathcal{P}$  
Names: $\mathcal{A}$  
CoName: $\overline{\mathcal{A}}$  
Labels: $\mathcal{L}$  
unobservable action: $\tau$  
Actions: $\mathcal{Act}$  
Agent Expressions: $\mathcal{E}$  
Indexing sets: 

$P, Q, \ldots$ range over $\mathcal{P}$  
$a, b, \ldots$ range over $\mathcal{A}$  
$\overline{a}, \overline{b}, \ldots$ range over $\overline{\mathcal{A}}$  
$L = \mathcal{A} \cup \overline{\mathcal{A}}$  
$Act = \mathcal{L} \cup \{\tau\}$  
$\alpha, \beta, \ldots$ range over $\mathcal{Act}$  
$E, F, \ldots$ range over $\mathcal{E}$  
$\{E_i : i \in I\}$
Agent Expressions

<table>
<thead>
<tr>
<th>Nil</th>
<th>does nothing</th>
<th>$\left(\sum_{i \in 0} E_i\right)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>$A = P;$</td>
<td>(infinite behavior via recursion)</td>
</tr>
<tr>
<td>prefixing</td>
<td>$\alpha.E$</td>
<td>$(\alpha \in Act)$</td>
</tr>
</tbody>
</table>

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: $E \setminus \mathcal{L}$ (\mathcal{L} should be \mathcal{A})
- Relabeling: $E[f]$

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

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

\[ a \cdot A \cdot \bar{c} \quad c \cdot B \cdot \bar{b} \]

\[ A \overset{\text{def}}{=} a.A' \quad B \overset{\text{def}}{=} c.B' \]

\[ A' \overset{\text{def}}{=} \bar{c}.A \quad B' \overset{\text{def}}{=} \bar{b}.B \]

Now consider:

\[ a \cdot A \cdot \bar{c} \quad c \cdot B \cdot \bar{b} \]

What is the difference between \( \bar{c} \) and \( c \)?
Composition

\[ a \cdot (A) \cdot \overline{c} \quad c \cdot (B) \cdot \overline{b} \]

\[ A \overset{\text{def}}{=} a.A' \quad B \overset{\text{def}}{=} c.B' \]

\[ A' \overset{\text{def}}{=} \overline{c}.A \quad B' \overset{\text{def}}{=} \overline{b}.B \]

Now consider:

\[ a \cdot (A) \cdot \overline{c} \quad c \cdot (B) \cdot \overline{b} \]

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

Labels and Colabels (inputs and outputs)
Composition

\[ a \cdot \overline{A} \cdot \overline{c} \quad \text{and} \quad c \cdot \overline{B} \cdot \overline{b} \]

\[ A \overset{\text{def}}{=} a \cdot A' \quad \text{and} \quad B \overset{\text{def}}{=} c \cdot B' \]

\[ A' \overset{\text{def}}{=} \overline{c} \cdot A \quad \text{and} \quad B' \overset{\text{def}}{=} \overline{b} \cdot B \]

Now consider:

\[ a \cdot \overline{A} \cdot \overline{c} \quad \text{and} \quad c \cdot \overline{B} \cdot \overline{b} \]

Can \( \overline{c} \) fire \textit{independently} in the lower graph?
Now consider:

Can \( \overline{c} \) fire \emph{independently} in the lower graph?

\textbf{YES!}

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

This does \textbf{not} indicate communication between \( A \) and \( B \)
Composition

\[ a \cdot (A) \cdot \bar{c} \]
\[ c \cdot (B) \cdot \bar{b} \]
\[ A \overset{\text{def}}{=} a \cdot A' \]
\[ B \overset{\text{def}}{=} c \cdot B' \]
\[ A' \overset{\text{def}}{=} \bar{c} \cdot A \]
\[ B' \overset{\text{def}}{=} \bar{b} \cdot B \]

Now consider:

\[ a \cdot (A) \cdot \bar{c} \rightarrow c \cdot (B) \cdot \bar{b} \]

Can \( c \) and \( \bar{c} \) handshake communicate in the lower graph?
Composition

\[
\begin{align*}
    a \cdot (\overline{A} \cdot \overline{c}) & \quad c \cdot (\overline{B} \cdot \overline{b}) \\
    A \overset{\text{def}}{=} a.A' & \quad B \overset{\text{def}}{=} c.B' \\
    A' \overset{\text{def}}{=} \overline{c}.A & \quad B' \overset{\text{def}}{=} \overline{b}.B
\end{align*}
\]

Now consider:

\[
\begin{align*}
    a \cdot (\overline{A} \cdot \overline{c}) & \quad c \cdot (\overline{B} \cdot \overline{b})
\end{align*}
\]

Can \(c\) and \(\overline{c}\) handshake communicate in the lower graph? YES!

Since \(A' \overset{\overline{c}}{\rightarrow} A\), and \(B \overset{c}{\rightarrow} B'\), we infer \(A' \ | \ B \overset{?}{\rightarrow} A \ | \ B'\)
Composition

\[ a \cdot \overline{A} \cdot \overline{c} \quad c \cdot \overline{B} \cdot \overline{b} \]

\[ A \overset{\text{def}}{=} a.A' \quad B \overset{\text{def}}{=} c.B' \]

\[ A' \overset{\text{def}}{=} \overline{c}.A \quad B' \overset{\text{def}}{=} \overline{b}.B \]

Now consider:

\[ a \cdot \overline{A} \overset{\overline{c}}{\longrightarrow} c \cdot \overline{B} \overset{\overline{b}}{\longrightarrow} \]

Since \( A' \overset{\overline{c}}{\longrightarrow} A \), and \( B \overset{c}{\longrightarrow} B' \), we infer \( A' \mid \overline{B} \overset{\overline{?}}{\longrightarrow} A \mid B' \)

When we have handshake communication what is the ‘?’ action?
Composition

\[ a \cdot \ll A \rr \cdot \overline{c} \quad c \cdot \ll B \rr \cdot \overline{b} \]

\[ A \overset{\text{def}}{=} a.A' \quad B \overset{\text{def}}{=} c.B' \]

\[ A' \overset{\text{def}}{=} \overline{c}.A \quad B' \overset{\text{def}}{=} \overline{b}.B \]

Now consider:

\[ a \cdot \ll A \rr \cdot \overline{c} \quad c \cdot \ll B \rr \cdot \overline{b} \]

Since \( A' \overset{\overline{c}}{\rightarrow} A \), and \( B \overset{\overline{c}}{\rightarrow} B' \), we infer \( A' \mid B \overset{?}{\rightarrow} A \mid B' \)

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

\[ a \cdot (A) \cdot \overline{c} \quad c \cdot (B) \cdot \overline{b} \]

\[ A \overset{\text{def}}{=} a.A' \quad B \overset{\text{def}}{=} c.B' \]

\[ A' \overset{\text{def}}{=} \overline{c}.A \quad B' \overset{\text{def}}{=} \overline{b}.B \]

Now consider:

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

\[ a \cdot \overline{A} \cdot \overline{c} \quad c \cdot \overline{B} \cdot \overline{b} \]

\[ A \overset{\text{def}}{=} a.A' \quad B \overset{\text{def}}{=} c.B' \]

\[ A' \overset{\text{def}}{=} \overline{c}.A \quad B' \overset{\text{def}}{=} \overline{b}.B \]

Now consider:

\[ a \cdot \overline{A} \cdot \overline{c} \quad c \cdot \overline{B} \cdot \overline{b} \]

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

\[ a \cdot \overline{A} \cdot \overline{c} \quad \quad c \cdot \overline{B} \cdot \overline{b} \]

\[ A \overset{\text{def}}{=} a \cdot A' \quad \quad B \overset{\text{def}}{=} c \cdot B' \]

\[ A' \overset{\text{def}}{=} \overline{c} \cdot A \quad \quad B' \overset{\text{def}}{=} \overline{b} \cdot B \]

Now consider:

\[ a \cdot \overline{A} \overline{c} \quad c \cdot \overline{B} \overline{b} \]

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

\[ a \cdot (A \cdot \overline{c}) \quad \quad c \cdot (B \cdot \overline{b}) \]
\[ A \overset{\text{def}}{=} a.A' \quad \quad B \overset{\text{def}}{=} c.B' \]
\[ A' \overset{\text{def}}{=} \overline{c}.A \quad \quad B' \overset{\text{def}}{=} \overline{b}.B \]

Now consider:

\[ a \cdot (A \cdot \overline{c}) \quad \quad c \cdot (B \cdot \overline{b}) \]

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 \((A \mid B) \setminus \{c\}\) will restrict the \(c\) and \(\overline{c}\) actions from occurring in the composed process.
Transitional Semantics

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

$S$: state set
$T$: Transition label set
Transition relation: $\xrightarrow{t} \subseteq S \times S \ \forall t \in T$

Also:

$S ::= \mathcal{E}$
$T ::= \text{Act}$

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

Sender $\rightarrow$ (medium) $\rightarrow$ 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 = \text{in}(x) \cdot \text{FIFO}'(x);
FIFO'(x) = \text{out}(x) \cdot \text{FIFO};

Equivalent definition:
FIFO = \text{in}(x) \cdot \text{'out}(x) \cdot \text{FIFO};

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

Exercise 1, page 18 of Milner
Review

\[ E ::= \]

- \( A \) \quad \text{constant}
- \( \alpha.E \) \quad \text{prefixing}
- \( \sum_{i \in I} E_i \) \quad \text{summation}
- \( E_1 | E_2 \) \quad \text{composition}
- \( E[f] \) \quad \text{relabeling}
- \( E\backslash L \) \quad \text{restriction}

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

In composition, a label and colabel interact to form a single indivisible communication action \( \tau \).
Derivation Tree

For the process:

\[
\begin{align*}
& a \quad \text{(A)} \quad \text{(B)} \quad \bar{b} \\
\end{align*}
\]

we can create the derivation tree:

\[
\begin{align*}
& (A \mid B) \\{c\} \\
& \quad \downarrow a \\
& (A' \mid B) \\{c\} \\
& \quad \downarrow \tau \\
& (A \mid B') \\{c\} \\
& \quad \overline{b} \quad a \\
& (A \mid B) \\{c\} \quad (A' \mid B') \\{c\} \\
& \quad \downarrow a \quad \downarrow \overline{b} \\
& (A' \mid B) \\{c\} \quad (A' \mid B) \\{c\} \\
& \quad \vdots \quad \vdots
\end{align*}
\]
Derivation Tree

For the process:

\[ a \cdot (A \rightarrow B) \cdot \bar{b} \]

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, \{ t\rightarrow \in T \})

These come in form of:
RULE name \rightarrow Act

Top of the bar is the hypothesis
Bottom of bar is the conclusion

\[
\alpha \cdot E \rightarrow^\alpha E
\]
Transitional Semantics

\[ \sum_{j} E_j \xrightarrow{\alpha} E'_j \quad (j \in I) \]

\[ \sum_{i \in I} E_i \xrightarrow{\alpha} E'_j \]

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

\[ E_1 \xrightarrow{\alpha} E'_1 \]

\[ \underline{E_1 + E_2 \xrightarrow{\alpha} E'_1} \quad \text{and} \quad E_2 \xrightarrow{\alpha} E'_2 \]

\[ \underline{E_1 + E_2 \xrightarrow{\alpha} E'_2} \]
Example

C-element defined as:

\[ C = a.b.'c.C + b.a.'c.C; \]

rules are

\[ a.b.'c.C \rightarrow b.'c.C \]

\[ a.b.'c.C + b.a.'c.C \rightarrow b.'c.C \]

and

\[ b.a.'c.C \rightarrow a.'c.C \]

\[ a.b.'c.C + b.a.'c.C \rightarrow a.'c.C \]
Transitional Semantics

Parallel Composition

$E_1 \parallel E_2$ has three rules:

Names joined to corresponding conames
Transitional Semantics

Com$_1$

$E_1 \xrightarrow{\alpha} E_1'$

$E_1 \mid E_2 \xrightarrow{\alpha} E_1' \mid E_2$

Com$_2$

$E_2 \xrightarrow{\alpha} E_2'$

$E_1 \mid E_2 \xrightarrow{\alpha} E_1 \mid E_2'$

Com$_3$

$E_1 \xrightarrow{l} E_1'$

$E_2 \xrightarrow{l} E_2'$

$E_1 \mid E_2 \xrightarrow{\tau} E_1' \mid E_2'$

$l \in \mathcal{L}$

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

if $l$ is $c$ then $\bar{l}$ is $\bar{c}$
Transitional Semantics

\( \tau \) is the unobservable internal action

- \( \tau \) is neither a name or coname
- \( \tau \) is not a well-formed action expression

Note:

- In 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 $L \subseteq L$, 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)

<table>
<thead>
<tr>
<th>Conj$_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \xrightarrow{\alpha} E'$</td>
</tr>
<tr>
<td>$E \mid F \xrightarrow{\alpha} E' \mid F$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conj$_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F \xrightarrow{\alpha} F'$</td>
</tr>
<tr>
<td>$E \mid F \xrightarrow{\alpha} E \mid F'$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conj$_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \xrightarrow{\bar{l}} E'$</td>
</tr>
<tr>
<td>$F \xrightarrow{l} F'$</td>
</tr>
<tr>
<td>$E \mid F \xrightarrow{\tau} E' \mid F'$</td>
</tr>
</tbody>
</table>

(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 $f: \text{Act} \rightarrow \text{Act}$

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

\[
\begin{array}{c}
E \xrightarrow{\alpha} E' \\
\hline
\text{Rel} \\
E[f] \xrightarrow{f(\alpha)} E'[f]
\end{array}
\]
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

\[ L \subseteq L \]

\[ \text{Res} \]

\[ E \xrightarrow{\alpha} E' \]

\[ E \setminus L \xrightarrow{\alpha} E' \setminus L \]

- \( E \setminus L \) has the same \( \tau \) transitions as \( E \) since \( \tau \notin L \)
- Syntax of some tools assumes \( L \notin \overline{A} \), but \( L \in A \)
Hiding

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

\[ L \subseteq L \]

**Hid\(_1\)**

\[
E \xrightarrow{\alpha} E' \\
\frac{}{E/L \xrightarrow{\alpha} E'/L} \quad (\alpha, \overline{\alpha} \notin L)
\]

**Hid\(_2\)**

\[
E \xrightarrow{l} E' \\
\frac{}{E/L \xrightarrow{l} E'/L} \quad (l \lor \overline{l} \in L)
\]

Does this result in \( E \) and \( E' \) being equivalent?
Hiding

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

$L \subseteq \mathcal{L}$

Hid$_1$

\[
\begin{align*}
E & \xrightarrow{\alpha} E' \\
E/L & \xrightarrow{\alpha} E'/L
\end{align*}
\]

$(\alpha, \bar{\alpha} \notin L)$

Hid$_2$

\[
\begin{align*}
E & \xrightarrow{l} E' \\
E/L & \xrightarrow{l} E'/L
\end{align*}
\]

$(l \lor \bar{l} \in L)$

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