Process Algebra

Sources:

- R. Cleaveland and O. Sokolsky. Equivalence and preorder checking for finite-state systems. In J.A. Bergstra, A. Ponse and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 391–424. Elsevier, Amsterdam, 2001.
- R. Milner. *A Calculus of Communicating Systems*, volume 92 in *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.
- R. Milner *Communication and Concurrency*. Prentice Hall, New York, 1989.

Process Algebras

... an approach to specifying and verifying concurrent systems

- Emphasis on modeling *open* systems, i.e. ones that can be embedded in other systems
- Theories built around notion of interaction between systems and environments
- Behavioral equivalences, refinement orderings used to relate systems, specifications
- Compositionality of modeling, verification a key feature

Mathematically...

... process algebras contain:

- A specification language containing operators for assembling subsystems into systems;
- A formal operational semantics of the language defining the *atomic* interactions a system may engage in with its environment;
- A notion of *behavioral refinement* for determining when one system "implements" another.

Traditionally, refinement relations are *equivalence relations*, although *preorders* also possible.

How Specification/Verification Works in Process Algebra

1. Formulate system *Spec* describing desired high-level behavior.
2. Devise candidate design *Des*.
3. Show *Des* is correct by establishing that *Des* refines *Spec*

CCS: A Calculus of Communicating Systems

We'll study the process-algebraic approach by looking at a specific process algebra, CCS.

- Devised by Robin Milner (a Turing Award winner!) in the late 1970's/early 1980's.
- Features binary handshaking as basic means of interaction.
- Processes built up from set of *atomic actions* using process constructors.

Actions in CCS

... are either inputs/outputs on ports or internal. Formally:

Let Λ be a(n infinite) set of *labels* (i.e. port names) not containing the reserved symbol τ.

Then an action in CCS is either:

- an input on port $\lambda \in \Lambda$: λ
- an output on port $\lambda \in \Lambda$: $\overline{\lambda}$
- an internal action: τ

Notation for Actions

Λ	set of labels *and* set of input actions
$\overline{\Lambda} = \{\, \overline{\lambda} \mid \lambda \in \Lambda \,\}$	set of output actions
$\Lambda \cup \overline{\Lambda}$	set of *external* actions
$Act = \Lambda \cup \overline{\Lambda} \cup \{\tau\}$	set of all actions

Convention

- $\overline{\overline{a}} = a$ if $a \in \Lambda \cup \overline{\Lambda}$.
- $\overline{\tau}$ is undefined.

What's the Idea with CCS Actions?

CCS systems communicate with their environments (and each other) by synchronizing on ports.

- If one partner can input and the other can output on the same port, then a synchronization may occur and both evolve.
- Inputs and outputs are blocking; only action a system can perform autonomously is τ.
- Thus the external actions a system can perform can be thought of as its interface.

Note No values exchanged in basic CCS; "output" means "emit a signal".

The Syntax of CCS

Let $a \in Act$, $L \subseteq \Lambda$, $f : \Lambda \rightarrow \Lambda$, $C \in \mathcal{C}$ where $\mathcal{C}$ is infinite set of process names.

$$
\begin{array}{rcll}
E & ::= & 0 & \text{"Termination"} \\
 & | & a.E & \text{"Prefixing"} \\
 & | & E_1 + E_2 & \text{"Choice"} \\
 & | & E_1 | E_2 & \text{"Parallel"} \\
 & | & E \backslash L & \text{"Restriction"} \\
 & | & E[f] & \text{"Relabeling"} \\
 & | & C & \text{"Invocation"}
\end{array}
$$

The Syntax of CCS (cont.)

A CCS expression E is *closed* if every process name has been "declared".

Declarations have form: $C \stackrel{\Delta}{=} E$.

$\boxed{\text{Example}}$ A declaration for process name A:

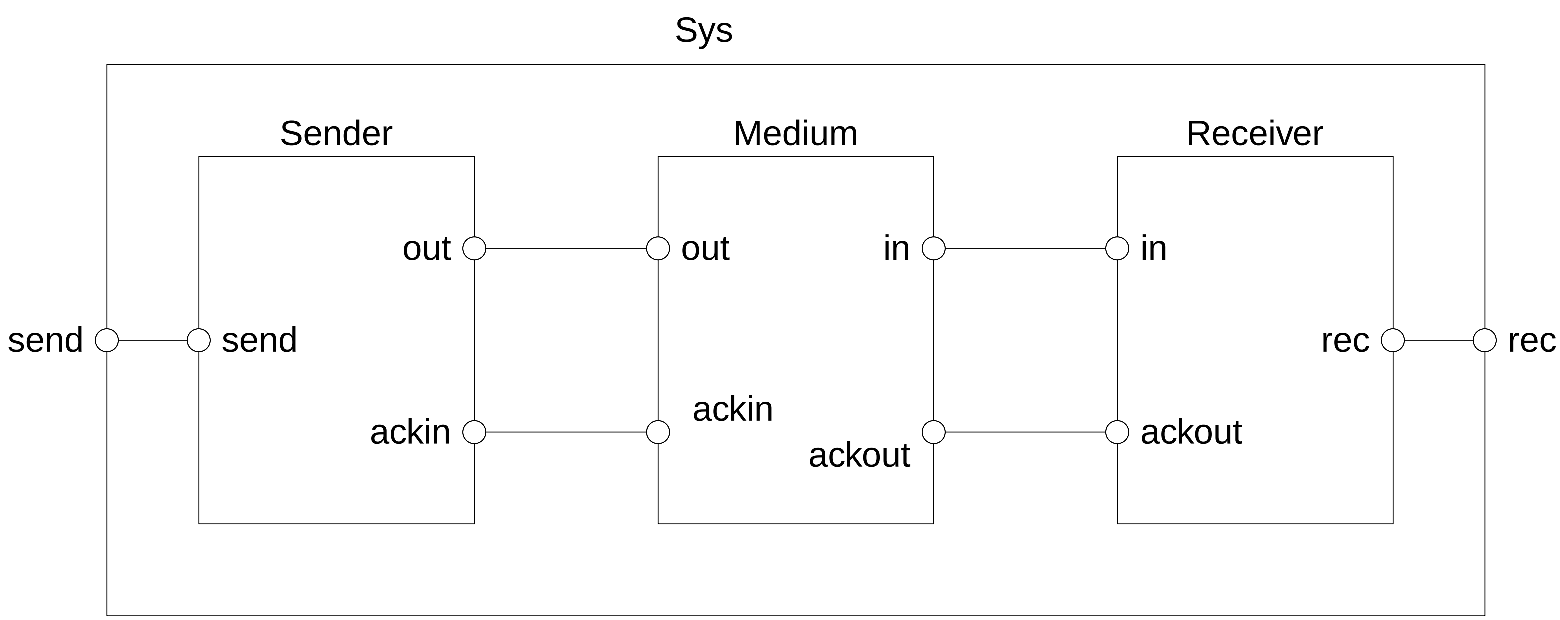
$$A \quad \stackrel{\Delta}{=} \quad a.b.A$$

Once this declaration has been made, expressions such as A, $A|A$ become closed.

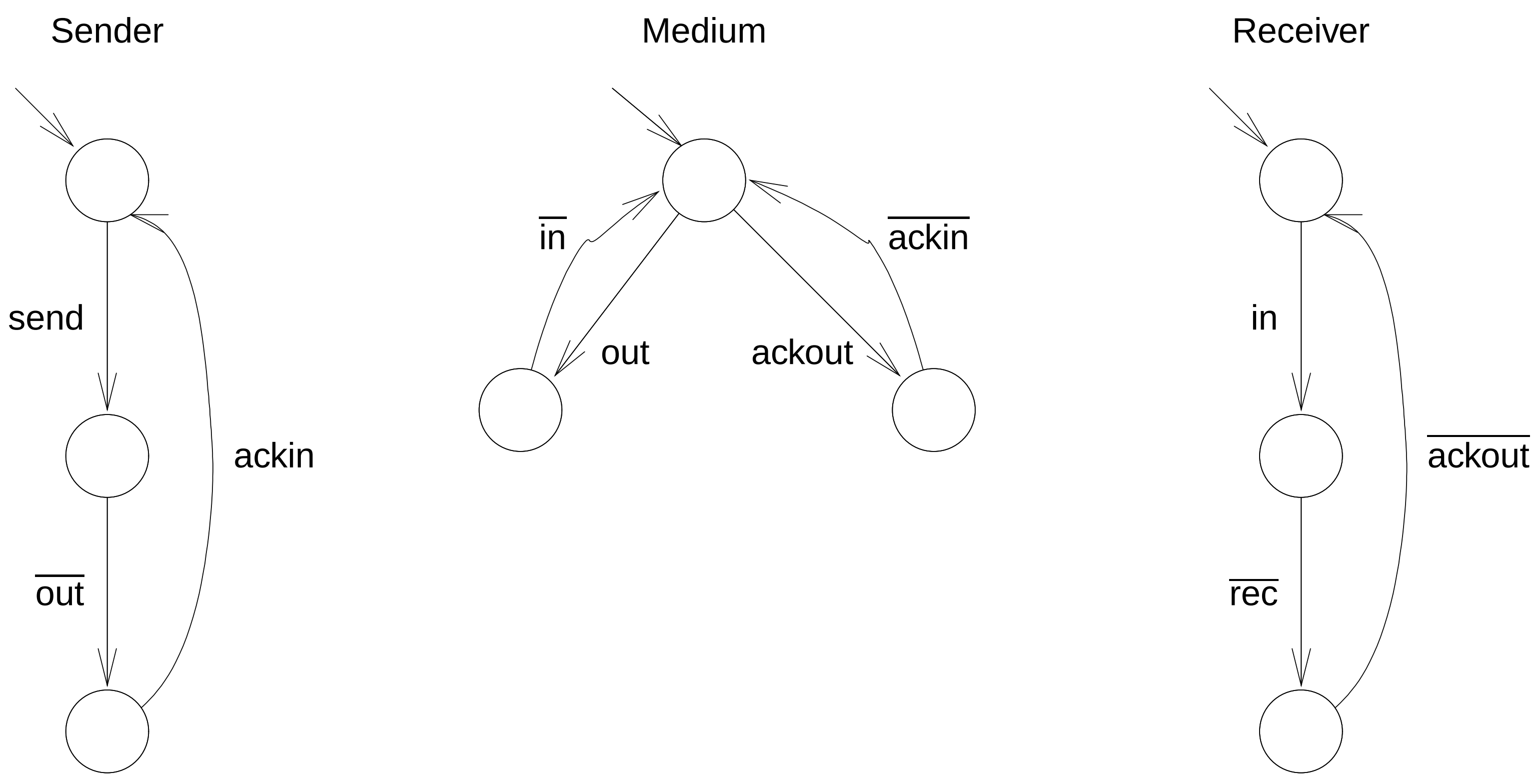
$\mathcal{P} \equiv$ set of CCS *processes* $\equiv$ set of closed CCS expressions.

Huh?

Idea CCS is used to encode textually a system like the following.



Individual Components of Sys



Here's the CCS

$$
\begin{aligned}
\mathsf{Sender} &\stackrel{\triangle}{=} \mathsf{send}.\overline{\mathsf{out}}.\mathsf{ackin}.\mathsf{Sender} \\
\mathsf{Medium} &\stackrel{\triangle}{=} \mathsf{out}.\overline{\mathsf{in}}.\mathsf{Medium} + \mathsf{ackout}.\overline{\mathsf{ackin}}.\mathsf{Medium} \\
\mathsf{Receiver} &\stackrel{\triangle}{=} \mathsf{in}.\overline{\mathsf{rec}}.\overline{\mathsf{ackout}}.\mathsf{Receiver} \\
\mathsf{Sys} &\stackrel{\triangle}{=} (\mathsf{Sender} \mid \mathsf{Medium} \mid \mathsf{Receiver}) \backslash \{\mathsf{in}, \mathsf{out}, \mathsf{ackin}, \mathsf{ackout}\}
\end{aligned}
$$

What Do CCS Descriptions Mean?

So far we've seen the syntax of CCS: $a., +, |, \backslash L, [f], C$

The next step: define the *behavior* of CCS expressions by giving the language an *operational semantics*.

- The semantics will define the execution steps of CCS systems.
- It will also be the basis for behavioral equivalences we will study.

The Operational Semantics of CCS ...

... is intended to capture a notion of “button-pushing”.

- Systems are boxes with buttons labeled by visible actions.
- Two kinds of buttons:
  - Input actions: usual kind of button that user presses.
  - Output actions: button is concealed by a little door.
- In different states, systems enable different buttons.
  - If button is an input, user may press it, and system changes state.
  - If button is an output, user may move little door to one side; then system “pushes out” button and changes state.

CCS Operators and Button-Pressing

Given button-pushing intuition, CCS operators can be viewed as operations for building boxes and composing them.

0**:** Box that responds to nothing.

$a.E$**:** Box that only enables a, behaves like E after a is performed.

$E + F$**:** Box that initially all buttons in both E, F, then behaves E or E based on buttons pressed.

CCS Operators and Button-Pressing II

$E|F$: Composite box responding to all button presses E, F can. In addition, outputs of E have doors swung to one side and “lined up” with inputs of F on same port, and vice versa (so boxes can “press each other’s buttons”)

$E\backslash L$: Box obtained by “taping over” buttons whose ports are in L.

$E[f]$: Box obtained by relabeling buttons according to f.

Capturing Button-Pressing Mathematically

The semantics of CCS is defined mathematically as a *ternary* relation $\longrightarrow \subseteq \mathcal{P} \times Act \times \mathcal{P}$.

- $\langle P, a, Q \rangle \in \longrightarrow$ means "P enables a, then behaves like Q after a performed."
- Notation: we write $P \xrightarrow{a} Q$ in lieu of $\langle P, a, Q \rangle \in \longrightarrow$.

Defining $\longrightarrow$: Structural Operational Semantics

$\longrightarrow$ given inductively as collection of *rules*.

- Each rule has form:

$$\boxed{\frac{\textit{premises}}{\textit{conclusion}}\ (\textit{side condition})}\ (\textit{name})$$

- Premises, conclusion are statements (*judgments*) of form $P \stackrel{a}{\longrightarrow} Q$.
- Intention: $P \stackrel{a}{\longrightarrow} Q$ is true if it can be derived using rules.

Rules may be grouped according to CCS operator; so this rule-based style is often called *Structural Operational Semantics* (SOS).

SOS Rules I

$$\frac{}{a.P \xrightarrow{a} P} \quad (\text{Act})$$

$$\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \quad (\text{Sum}_1)$$

$$\frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'} \quad (\text{Sum}_2)$$

SOS Rules II

$$\boxed{\frac{P \xrightarrow{a} P'}{P|Q \xrightarrow{a} P'|Q}} \quad (\text{Com}_1)$$

$$\boxed{\frac{Q \xrightarrow{a} Q'}{P|Q \xrightarrow{a} P|Q'}} \quad (\text{Com}_2)$$

$$\boxed{\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}} \quad (\text{Com}_3)$$

SOS Rules III

$$\boxed{\frac{P \xrightarrow{a} P'}{P \backslash L \xrightarrow{a} P' \backslash L} \; (a, \overline{a} \notin L)} \quad (\text{Res})$$

$$\boxed{\frac{P \xrightarrow{a} P'}{P[f] \xrightarrow{\hat{f}(a)} P'[f]}} \quad (\text{Rel})$$

$$\boxed{\frac{P \xrightarrow{a} P'}{C \xrightarrow{a} P'} \; (C \in \mathcal{C}, C \stackrel{\Delta}{=} P)} \quad (\text{Con})$$

Notes on Rules

1. Each rule has a name for ease of reference.

2. Act rule has no premises and hence can be viewed as an axiom.

3. Rules for $+$, $\mid$ make precise the "button-pressing" intuitions for these operators.

4. Result of synchronization (Com$_3$) is always τ.

5. In Rel, recall $f : \Lambda \rightarrow \Lambda$. $\hat{f} : Act \rightarrow Act$ is given by:

$$\hat{f}(a) \quad = \quad \begin{cases} a & \text{if } a \in \Lambda \\ \overline{f(b)} & \text{if } a = \overline{b} \text{ and } b \in \Lambda \\ \tau & \text{if } a = \tau \end{cases}$$

SOS and Transitions for CCS Systems

$\boxed{\text{Question}}$ In what sense do the SOS rules "define" $\longrightarrow$?

The answer:

- The SOS rules define an inference system, where statements inferred have form "$P \xrightarrow{a} Q$".
- A transition $P \xrightarrow{a} Q$ can be inferred if one can construct a proof using the rules.
- So the relation $\longrightarrow$ contains exactly those process-action-process triples that can be inferred from the rules.

Example: Infer $((a.P + b.0) \mid \overline{a}.Q)\backslash\{a\} \xrightarrow{b} (0 \mid \overline{a}.Q)\backslash\{a\}$

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{b.0 \xrightarrow{b} 0}\ \text{Act}}{a.P + b.0 \xrightarrow{b} 0}\ \text{Sum}_2}{(a.P + b.0) \mid \overline{a}.Q \xrightarrow{b} 0 \mid \overline{a}.Q}\ \text{Com}_1 \qquad b, \overline{b} \notin \{a\}}{((a.P + b.0) \mid \overline{a}.Q)\backslash\{a\} \xrightarrow{b} (0 \mid \overline{a}.Q)\backslash\{a\}}\ \text{Res}
$$

Example: Infer $((a.P + b.0) \,|\, \overline{a}.Q)\backslash\{a\} \xrightarrow{\tau} (P \,|\, Q)\backslash\{a\}$

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{a.P \xrightarrow{a} P}\text{ Act}}{a.P + b.0 \xrightarrow{a} P}\text{ Sum}_2 \qquad \dfrac{}{\overline{a}.Q \xrightarrow{\overline{a}} Q}\text{ Act}}{(a.P + b.0) \,|\, \overline{a}.Q \xrightarrow{\tau} P \,|\, Q}\text{ Com}_3 \qquad \tau \notin \{a\}}{((a.P + b.0) \,|\, \overline{a}.Q)\backslash\{a\} \xrightarrow{\tau} (P \,|\, Q)\backslash\{a\}}\text{ Res}
$$

Notes

1. Proofs built in *forward-chaining* manner: use inference rules to infer new conclusions from existing ones.

2. Such forward-chaining proofs always “begin” with an application of Act rule.

3. Side condition in Res rule must hold for rule to be applied; so

$$((a.P + b.0)|\overline{a}.Q)\backslash\{a\} \xrightarrow{a} (P|\overline{a}.Q)\backslash\{a\}$$

   cannot be proved!

Static and Dynamic CCS Operators

Based on form of rules, one can categorize CCS operators as either:

- Dynamic: 0, $a.$, $+$
- Static: $|$, $\backslash L$, $[f]$

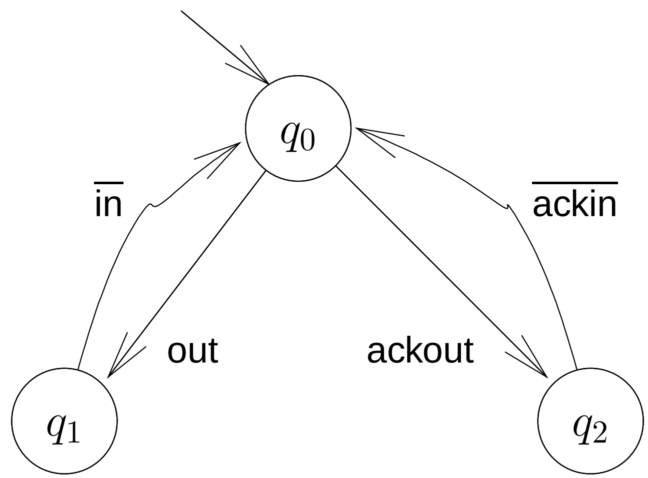
Distinction: whether or not operator appears in target of conclusion of operator's rules!

(Of course, 0 has no rules ... it nevertheless has traditionally called a dynamic operator.)

Building Systems in CCS: Labeled Transition Systems

Labeled transition systems (LTSs): like finite-state automata, but with no accepting states.

E.g.



CCS and LTSs

CCS may be viewed as a (infinite-state) LTS with no initial state.

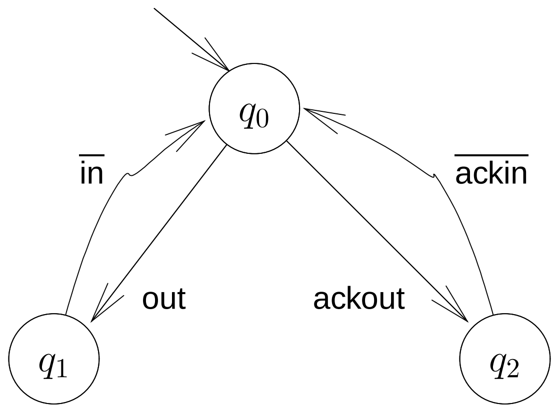
- States are closed terms.
- Transitions given by $\longrightarrow$, i.e. by operational semantics.

Any finite-state LTS can be encoded in CCS.

- Associate a process name S to each LTS state s.
- In declaration of S, sum together terms of form $a.T$ for each transition $s \xrightarrow{a} t$ in LTS.
- Process name for start state is then CCS encoding of LTS.

Example

For LTS:



CCS declarations are:

$$
\begin{aligned}
\mathsf{Q0} &\stackrel{\Delta}{=} \mathsf{out.Q1} + \mathsf{ackout.Q2} \\
\mathsf{Q1} &\stackrel{\Delta}{=} \overline{\mathsf{in}}.\mathsf{Q0} \\
\mathsf{Q2} &\stackrel{\Delta}{=} \overline{\mathsf{ackin}}.\mathsf{Q0}
\end{aligned}
$$

and CCS for over-all system is Q0.

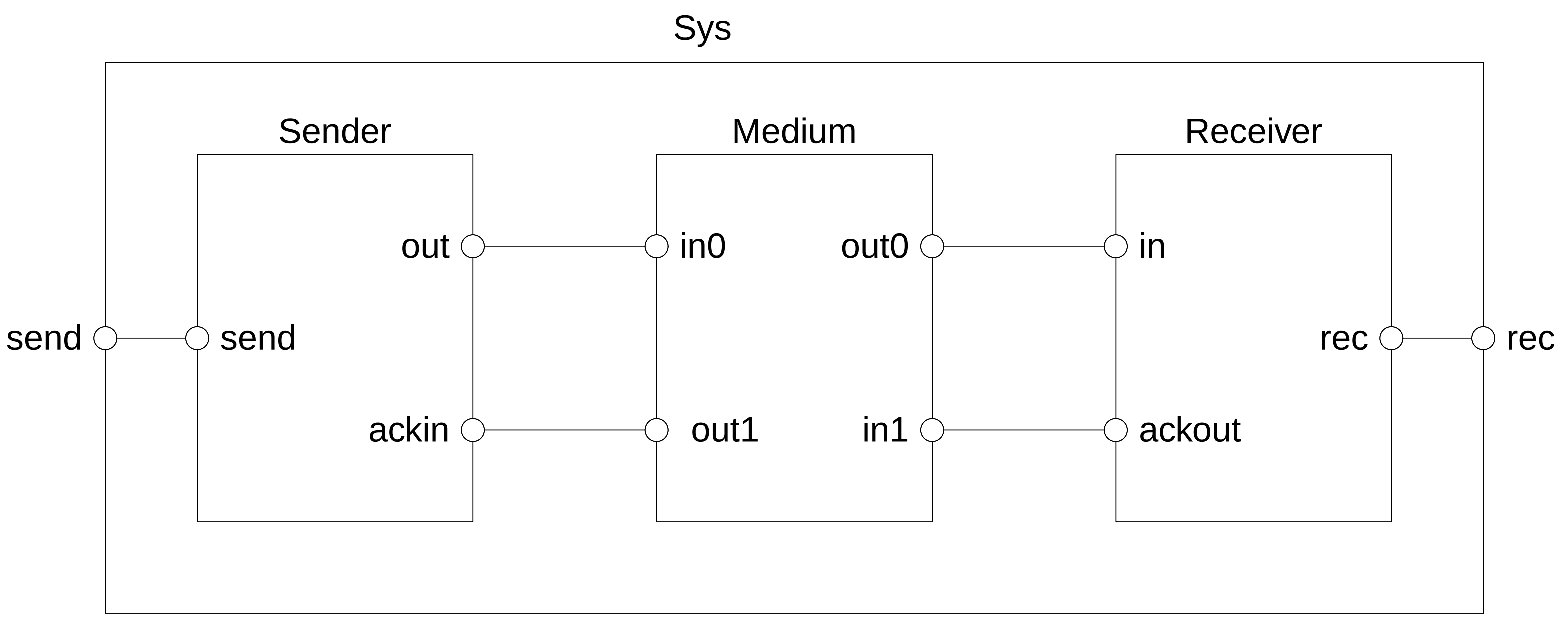
Note

Encoding of LTS's requires only the *dynamic* operators (and declarations)!

So how are static operators used? To encode *architectural information*.

Representing Architectures in CCS

How do we represent information like the following in CCS?



What Architectures Contain

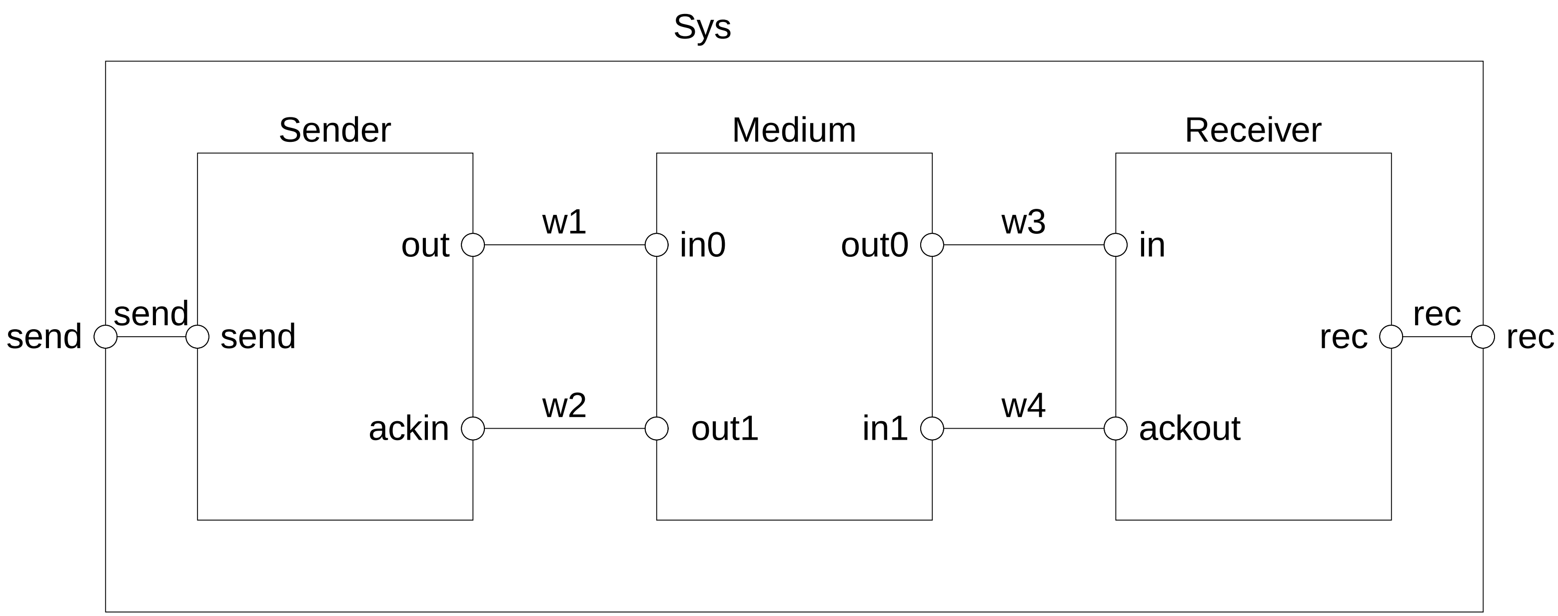
- *Boxes* with *ports*
- *Wires* connecting ports on different boxes
- *Subarchitectures* embedded inside boxes

Basic Ideas Underlying Encoding

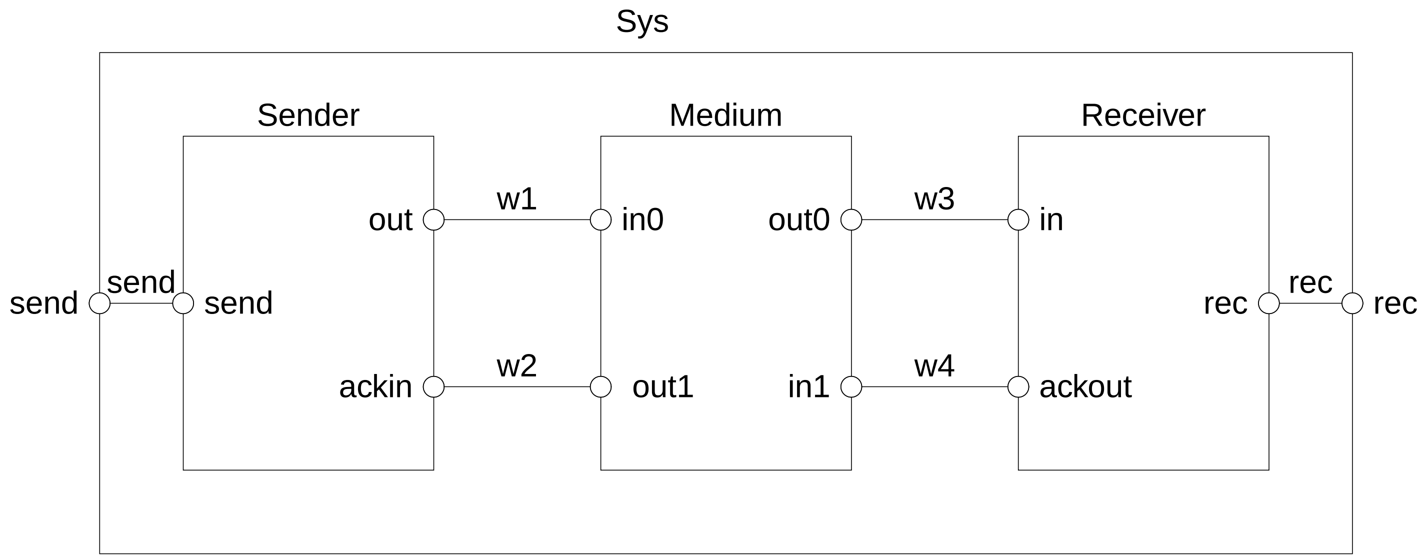
- Associate a name to each box, and a name to each “wire”.
- Boxes in same architecture run in parallel.
- Use renaming to “connect” a port to a wire if wire name is different from port name.
- Use restriction when embedding an architecture inside a box.

Example Revisited

Here is the example with wires named.



Here Is the CCS



$$
\begin{aligned}
\text{Sys} \quad \stackrel{\Delta}{=} \quad & \big(\, \text{Sender}\,[\text{w1/out, w2/ackin}] \\
& \mid \text{Medium}\,[\text{w1/in0, w2/out1, w3/out0, w4/in1}] \\
& \mid \text{Receiver}\,[\text{w3/in, w4/ackout}] \\
& \big) \setminus \{\text{w1, w2, w3, w4}\}
\end{aligned}
$$

Notes

1. Notation for relabeling: $P[a/b, c/d]$ means “substitute a for b, c for d, leave all other labels unchanged.”
2. Relabeling used to do “wiring”.
3. Restriction used to “localize” wires, ports.
4. Only static operators (and process names) needed!
5. This scheme works if wire names are distinct from all ports that they are not connected to.

The CCS Verification Framework

Sys: CCS expressions

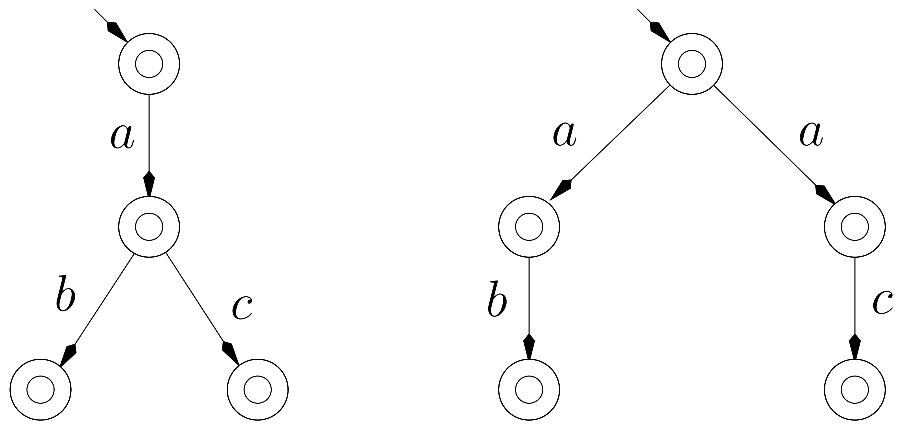
Spec: CCS expressions

sat: *Behavioral equivalence* $\equiv$

Intuition If $I \equiv S$ then implementation I behaves the same as spec S.

When Should We Consider Two CCS Agents Equivalent?

- Consider $P = a.(b.0 + c.0)$ and $Q = a.b.0 + a.c.0$.
- Now consider P and Q's respective LTSs (from CCS's operational semantics), and imagine every state to be "accepting".



- P and Q both accept $\{\epsilon, a, ab, ac\}$ and should therefore be considered equivalent in classical theory of NFA and regular languages.
- But is this what we want in a theory based on "interaction"?

On the (In)Equivalence of P and Q: Another View

- Consider now a “test” or “probe” process $T = \overline{a}.\overline{b}.\overline{w}.0$ ($\overline{w}$ indicates “success”) ...
- ... and consider $(P|T)\backslash L$ and $(Q|T)\backslash L$ where $L = \{a, b, c\}$.
- In the former, the test invariably “succeeds” while in the latter the interaction between Q and T may come to a halt before success can be reported.
- This is because of the nondeterminism in Q. What to do?

Strong Bisimulation

A *bisimulation* is a kind of invariant holding between a pair of dynamic systems, and the technique is to prove two systems equivalent by establishing such an invariant, much as one can prove correctness of a single sequential program by finding an invariant property.

[Milner89]

Definition of a Strong Bisimulation

A binary relation $S \subseteq \mathcal{P} \times \mathcal{P}$ is a *strong bisimulation* if $(P, Q) \in S$ implies, for all a in Act,

1. Whenever $P \xrightarrow{a} P'$ then, for some Q', $Q \xrightarrow{a} Q'$ and $(P', Q') \in S$.
2. Whenever $Q \xrightarrow{a} Q'$ then, for some P', $P \xrightarrow{a} P'$ and $(P', Q') \in S$.

It helps to draw a diagram!

Bisimulation Diagramatically

$$
\begin{tikzcd}
{} \arrow[d, phantom, "\forall"] & P \arrow[r, dash] \arrow[d, "a"'] & S \arrow[r, dash] & Q \arrow[d, dashed, "a"] & {} \arrow[d, phantom, "\exists"] \\
{} & P' \arrow[r, dash] & S \arrow[r, dash] & Q' & {}
\end{tikzcd}
$$

Strong Equivalence

Two agents P and Q are *strongly equivalent* or *strongly bisimilar*, written $P \sim Q$, if $(P, Q) \in S$ for some strong bisimulation S. This may be equivalently expressed as follows:

$$\sim \quad = \quad \bigcup \{S \mid S \text{ is a strong bisimulation}\}$$

This definition immediately suggests a *proof technique* for $\sim$: exhibit a strong bisimulation that relates P and Q.

Examples

1. $a.c.0 + b.0 \sim (a.c.0 + b.0) + a.c.0$
2. $a.(b.0 + c.0) \not\sim a.b.0 + a.c.0$
3. $a.b.0 \not\sim a.b.0 + a.0$

A Larger Example: A Counting Semaphore

$$
\begin{array}{rcll}
Sem_n(0) & \triangleq & get.Sem_n(1) & \\
Sem_n(k) & \triangleq & get.Sem_n(k+1) + put.Sem_n(k-1) & (0 \leq k \leq n) \\
Sem_n(n) & \triangleq & put.Sem_n(n-1) &
\end{array}
$$

$$
\begin{array}{rcl}
Sem & \triangleq & get.Sem' \\
Sem' & \triangleq & put.Sem
\end{array}
$$

$$
\begin{array}{rcl}
S & = & \{\, (Sem_2(0), Sem|Sem), \\
 & & (Sem_2(1), Sem|Sem'), \\
 & & (Sem_2(1), Sem'|Sem), \\
 & & (Sem_2(2), Sem'|Sem') \,\}
\end{array}
$$

Proving $P \sim Q$

$\boxed{\text{Idea}}$ Build strong bisimulation $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ containing $\langle P, Q \rangle$!

Why does this work? Definition of $\sim$:

> $P \sim Q$ iff there exists strong bisimulation $\mathcal{S}$ relating P, Q.

$\boxed{\text{Example}}$ Prove that $a.b.0 \sim a.b.0 + a.b.(0 + 0)$.

Building a Bisimulation for $a.b.0 \sim a.b.0 + a.b.(0 + 0)$

Proving $P \not\sim Q$

Recall: $P \sim Q$ iff some strong bisimulation relates P, Q.

So, to prove $P \not\sim Q$, need to show that no bisimulation relates P, Q. Proofs proceed by contradiction.

- Assume a strong bisimulation exists relating P, Q.
- Show that this leads to a contradiction.

Example Proof: $P \equiv a.(b.0 + c.0) \not\sim Q \equiv a.b.0 + a.c.0$

- Assume $\mathcal{S}$ is a strong bisimulation with $\langle P, Q \rangle \in \mathcal{S}$.
- Since:
  - $P \xrightarrow{a} b.0 + c.0$
  - $\mathcal{S}$ is a bisimulation
  - $Q \xrightarrow{a} b.0$ and $Q \xrightarrow{a} c.0$

  it must be that either $\langle b.0 + c.0, b.0 \rangle \in \mathcal{S}$ or $\langle b.0 + c.0, c.0 \rangle \in \mathcal{S}$
- But $b.0 + c.0 \xrightarrow{c} 0$ while $b.0 \not\xrightarrow{c}$, and $b.0 + c.0 \xrightarrow{b} 0$ while $c.0 \not\xrightarrow{b}$.
- Therefore, no such $\mathcal{S}$ can exist, and $P \not\sim Q$.

Observational Equivalence

Problem with $\sim$: too sensitive to τ (i.e. internal) transitions!

$\boxed{\text{E.g}}$ $a.\tau.b.0 \not\sim a.b.0$

Defining Observational Equivalence: Preliminaries

Need to introduce derived transition relation, $\Longrightarrow$, that “absorbs” internal computation.

- $P \stackrel{\epsilon}{\Longrightarrow} Q$ iff $P \underbrace{\stackrel{\tau}{\longrightarrow} \cdots \stackrel{\tau}{\longrightarrow}}_{\geq 0} Q$.
- $P \stackrel{a}{\Longrightarrow} Q$ iff for some P', Q', $P \stackrel{\epsilon}{\Longrightarrow} P' \stackrel{a}{\longrightarrow} Q' \stackrel{\epsilon}{\Longrightarrow} Q$.

  i.e. $P \stackrel{a}{\Longrightarrow} Q$ if $P \underbrace{\stackrel{\tau}{\longrightarrow} \cdots \stackrel{\tau}{\longrightarrow}}_{\geq 0} \stackrel{a}{\longrightarrow} \underbrace{\stackrel{\tau}{\longrightarrow} \cdots \stackrel{\tau}{\longrightarrow}}_{\geq 0} Q$.
- $\hat{a}$, the *visible content of* a, is ϵ if $a = \tau$ and a otherwise.

$\Longrightarrow$ sometimes called the *weak transition relation*.

Examples

$$\tau.\tau.b.0 + \tau.a.\tau.0 \stackrel{\epsilon}{\Longrightarrow} \tau.\tau.b.0 + \tau.a.\tau.0$$

$$\tau.\tau.b.0 + \tau.a.\tau.0 \stackrel{\hat{\tau}}{\Longrightarrow} b.0$$

$$\tau.\tau.b.0 + \tau.a.\tau.0 \stackrel{\tau}{\Longrightarrow} b.0$$

$$\tau.\tau.b.0 + \tau.a.\tau.0 \stackrel{\tau}{\not\Longrightarrow} \tau.\tau.b.0 + \tau.a.\tau.0$$

$$\tau.\tau.b.0 + \tau.a.\tau.0 \stackrel{a}{\Longrightarrow} \tau.0$$

$$\tau.\tau.b.0 + \tau.a.\tau.0 \stackrel{\hat{a}}{\Longrightarrow} \tau.0$$

$$\tau.\tau.b.0 + \tau.a.\tau.0 \stackrel{a}{\Longrightarrow} 0$$

Defining Observational Equivalence

$\boxed{\text{Definition}}$ A relation $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ is a *(weak) bisimulation* if whenever $\langle P, Q \rangle \in \mathcal{S}$ then:

1. $P \xrightarrow{a} P'$ implies $Q \stackrel{\hat{a}}{\Longrightarrow} Q'$ some Q' such that $\langle P', Q' \rangle \in \mathcal{S}$.

2. $Q \xrightarrow{a} Q'$ implies $P \stackrel{\hat{a}}{\Longrightarrow} P'$ some P' such that $\langle P', Q' \rangle \in \mathcal{S}$.

$\boxed{\text{Definition}}$ $P \approx Q$ iff there exists a bisimulation $\mathcal{S}$ with $\langle P, Q \rangle \in \mathcal{S}$.

Proving/Disproving $\approx$

Definitions of strong/weak bisimulations, $\sim$/$\approx$ are very similar.

Consequence: proof techniques for $\approx, \not\approx$ similar to those for $\sim, \not\sim$.

- To show $P \approx Q$, build a weak bisimulation containing $\langle P, Q \rangle$.
- To show $P \not\approx Q$, use a proof by contradiction.

Example: $a.\tau.b.0 \approx a.b.0$

Example: $a.0 + \tau.b.0 \not\approx a.0 + b.0$

A Larger Example

Consider:

$$
\begin{array}{rcl}
\text{Sender} & \triangleq & \text{send}.\overline{\text{out}}.\text{ackin}.\text{Sender} \\
\text{Medium} & \triangleq & \text{out}.\overline{\text{in}}.\text{Medium} + \text{ackout}.\overline{\text{ackin}}.\text{Medium} \\
\text{Receiver} & \triangleq & \text{in}.\overline{\text{rec}}.\overline{\text{ackout}}.\text{Receiver} \\
\text{Sys} & \triangleq & (\text{Sender} \mid \text{Medium} \mid \text{Receiver}) \setminus \{\text{in, out, ackin, ackout}\} \\
 & & \\
\text{Spec} & \triangleq & \text{send}.\overline{\text{rec}}.\text{Spec}
\end{array}
$$

Then Sys $\approx$ Spec!

A Weak Bisimulation for the Larger Example

Assessing Observational Equivalence

Positives

- Recursive character eliminates problems of $=_L$ (traditional language equivalence).
- Relative insensitivity to τ-transitions remedies deficiency of $\sim$.
- It inherits elegant proof techniques from $\sim$.

Alas, there is a fly in the ointment:

$\approx$ is not a *congruence* for CCS.

Huh?

Intuition An equivalence relation is a *congruence* for a language if you can substitute “equals for equals”.

Why do we care about congruences? They support *compositional reasoning* (reasoning about a system by reasoning about its parts).

$\sim$ Is a Congruence for CCS

$\boxed{\text{Definition}}$ A CCS *context* $C[]$ is a CCS term with a "hole" $[]$ (e.g. $a.[]$, $a.b.0|c.[]$, etc.)
If $C[]$ is a context and p is a term, then $C[p]$ is the term formed by replacing $[]$ by p in $C[]$.

$\boxed{\text{Theorem (Congruence-hood of } \sim \text{ for CCS)}}$ Let $C[]$ be a CCS context. Then for any P, Q, if $P \sim Q$ then $C[P] \sim C[Q]$.

Proof proceeds "operator-wise": show that for any P, Q, if $P \sim Q$ and $a.P \sim a.Q$, $P + R \sim Q + R$, etc.

Congruence-hood and Compositional Reasoning

.

Recall:

$$
\begin{aligned}
Sem_n(0) &\triangleq get.Sem_n(1) \\
Sem_n(k) &\triangleq get.Sem_n(k+1) + put.Sem_n(k-1) \qquad (0 \le k \le n) \\
Sem_n(n) &\triangleq put.Sem_n(n-1)
\end{aligned}
$$

$$
\begin{aligned}
Sem &\triangleq get.Sem' \\
Sem' &\triangleq put.Sem
\end{aligned}
$$

- We showed $Sem_2(0) \sim Sem \,|\, Sem$ by constructing a bisimulation.
- We can use this fact and congruence-hood (“substitutivity”) of $\sim$ to prove $Sem_2(0) \,|\, Sem_2(0) \sim Sem \,|\, Sem \,|\, Sem \,|\, Sem$

$\approx$ Is Not a Congruence for CCS

In setting of CCS, if $\approx$ were a congruence then $p \approx q$ would imply:

1. $a.p \approx a.q$
2. $p + r \approx q + r$ any r
3. $p|r \approx q|r$ any r
4. $p\backslash L \approx q\backslash L$ any L
5. $p[f] \approx q[f]$ any f

All hold except (2).

The Problem with $\approx$ and $+$

... best illustrated by an example.

Fact $\tau.b.0 \approx b.0$

Another Fact $a.0 + \tau.b.0 \not\approx a.0 + b.0$

What To Do?

- Problem with $\approx$ stems from initial internal computation.
- Perhaps we can just hack the definition of $\approx$ to fix this.

$\boxed{\text{Definition}}$ $P \approx^C Q$ if for all $a \in Act$:

1. $P \xrightarrow{a} P'$ implies $Q \overset{a}{\Longrightarrow} Q'$ and $P' \approx Q'$ some Q'.
2. $Q \xrightarrow{a} Q'$ implies $P \overset{a}{\Longrightarrow} P'$ and $P' \approx Q'$ some P'.

Fact about $\approx^C$

1. $\tau.b.0 \not\approx^C b.0$
2. $a.\tau.b.0 \approx^C a.b.0$
3. If $P \approx Q$, $P \stackrel{\tau}{\not\longrightarrow}$ and $Q \stackrel{\tau}{\not\longrightarrow}$ then $P \approx^C Q$.
4. If $P \approx Q$ then either $P \approx^C Q$, $\tau.P \approx^C Q$, or $P \approx^C \tau.Q$

And yet ... $\approx^C$ is still a hack: is there a reason to believe it is the "best hack"?

Justifying $\approx^C$

It turns out that $\approx^C$ is the *largest* congruence contained in $\approx$. That is:

- Whenever $P \approx^C Q$ then $P \approx Q$ (equivalently: $\approx^C \subseteq \approx$).
- For any other congruence $\approx^D \subseteq \approx$, $\approx^D \subseteq \approx^C$.

So $\approx^C$ is the "most permissive" congruence consistent with $\approx$.

Practical Ramifications of $\approx$, $\approx^C$

1. Since problem with $\approx$ stems solely from $+$, some researchers suggest that $+$ is really the issue.

2. On the other hand, in most scenarios compositional reasoning only exploited in context of static operators of CCS; i.e. one does not substitute inside $+$.

3. So people still use $\approx$ in many cases.

Equivalence and Property Preservation

Temporal logic: Focus is on establishing individual properties of systems

Process algebra: Focus is on establishing equivalences between systems

The two points of view turn out to be related: $\sim$ and $\approx$ have *logical characterizations*.

Hennessy-Milner Logic (HML)

... a logic for writing simple *modal* formulas

... proven by Hennessy and Milner to *characterize* $\sim$: two processes are $\sim$ iff they satisfy the same HML formulas.

So if $P \not\sim Q$, there exists a formula satisfied by one and not the other.

Syntax of HML

Let $a \in Act$.

$$
\begin{array}{rcll}
\phi & ::= & \mathrm{tt} & \text{“True”} \\
 & | & \neg\phi & \text{“Not”} \\
 & | & \phi_1 \vee \phi_2 & \text{“Or} \\
 & | & \langle a \rangle \phi & \text{“Possibility”}
\end{array}
$$

Φ: set of all HML formulas.

Semantics of HML ...

... given as a relation $\models \subseteq \mathcal{P} \times \Phi$.

- We write $P \models \phi$ rather than $\langle P, \phi \rangle \in \models$.
- $P \models \phi$: "P makes ϕ true."

Defining $\models$

$P \models \mathsf{tt}$	for any P	
$P \models \neg\phi$	if	$P \not\models \phi$
$P \models \phi_1 \vee \phi_2$	if	$P \models \phi_1$ or $P \models \phi_2$
$P \models \langle a \rangle \phi$	if	$P \xrightarrow{a} P'$ and $P' \models \phi$, for some P'

Derived Operators

“False”	ff	$\equiv$	
“And”	$\phi_1 \wedge \phi_2$	$\equiv$	
“Necessity”	$[a]\phi$	$\equiv$	

Examples

Does...

- $a.0 \models \langle a \rangle$tt?
- $a.(b.0 + c.0) \models \langle a \rangle \langle b \rangle$tt?
- $a.b.0 + a.c.0 \models \langle a \rangle (\langle b \rangle$tt $\wedge \langle c \rangle$tt$)$?
- $a.(b.0 + c.0) \models [a] \langle b \rangle$tt?
- $a.b.0 + a.c.0 \models [a] \langle b \rangle$tt?
- $0 \models [a]$ff?

HML and $\sim$

By itself, HML is not very expressive (any formula can only talk about a finite part of process behavior).

Nevertheless, we have the following.

$\boxed{\text{Definition}}$ $P =_{HML} Q$ iff for all $\phi \in \Phi$, $P \models \phi$ iff $Q \models \phi$.

$\boxed{\text{Result}}$ $P \sim Q$ iff $P =_{HML} Q$.

In other words $P \sim Q$ iff P, Q satisfy exactly the same HML formulas!

(There is a minor technical restriction on P, Q: *image-finiteness*.)

Hmm....

So we can give formulas that "distinguish":

- $a.b.0$ and $a.c.0$
- $a.(b.0 + c.0)$ and $a.b.0$
- $a.(b.0 + c.0)$ and $a.b.0 + a.c.0$

Note This gives us another method for proving $P \not\sim Q$!

Proving the Correspondence Between $\sim$, $=_{HML}$

Proof breaks into two directions.

$\Rightarrow$ Assume $P \sim Q$, show $P =_{HML} Q$. Proof uses case analysis, induction on structure of HML formulas.

$\Rightarrow$ Assume $P =_{HML} Q$, show $P \sim Q$. What do we do?

What About $\approx$?

The results for HML and $\sim$ can be ported to $\approx$ once we notice the following.

Fact $\approx$ is the largest relation such that the following hold for all $a \in Act$.

1. $P \stackrel{\hat{a}}{\Longrightarrow} P'$ implies $Q \stackrel{\hat{a}}{\Longrightarrow} Q'$ some Q' such that $P' \approx Q'$.
2. $Q \stackrel{\hat{a}}{\Longrightarrow} Q'$ implies $P \stackrel{\hat{a}}{\Longrightarrow} P'$ some P' such that $P' \approx Q'$.

How Does this Help?

Can now define "weak HML" (WHML) just like HML except with a weak modality, $\langle\langle a \rangle\rangle$!

$$P \models \langle\langle a \rangle\rangle \phi \text{ if } P \stackrel{\hat{a}}{\Longrightarrow} P' \text{ and } P' \models \phi \text{ some } P'.$$

Derived operator: $[[a]]\phi \equiv \neg\langle\langle a \rangle\rangle\neg\phi$

Can define $=_{WHML}$ analogously with $=_{HML}$.

Then $P \approx Q$ iff $P =_{WHML} Q$!

Axiomatizing $\sim$ / $\approx$

In other verification frameworks, we showed how to prove correctness of systems *vis à vis* specifications.

In CCS we'll show how to give *equational* proofs of equivalences.

Equational Proof Systems ...

... proof systems for establishing equalities!

Recall components of a symbolic logic:

- Syntax
- Semantics
- Proof system (= axioms + inference rules) for establishing *judgments*

In equational proof systems, judgments have form $P = Q$, where P, Q are *terms* in the syntax.

Equational proof systems consist of *logical* axioms and inference rules and *non-logical* axioms.

Such proof allow development of proofs like this.

$$
\begin{array}{rcll}
5 + (3 \cdot 8) + 11 & = & 5 + 24 + 11 & \text{(Mult)} \\
 & = & 29 + 11 & \text{(Add)} \\
 & = & 40 & \text{(Add)}
\end{array}
$$

Logical Axioms and Inference Rules for Equational Proof Systems

$$\boxed{\frac{-}{P = P}}\,(R)$$

$$\boxed{\frac{P = Q}{Q = P}}\,(S)$$

$$\boxed{\frac{P = Q \quad Q = R}{P = R}}\,(T)$$

$$\boxed{\frac{P = Q}{C[P] = C[Q]}}\,(C)$$

Non-logical Axioms in Equational Proof Systems

... depend on *semantics* of judgments.

For CCS, we will study two different semantics.

Strong equivalence: $P = Q$ is true iff $P \sim Q$.

Observational congruence: $P = Q$ is true iff $P \approx^C Q$.

Equational Axiomatization of $\sim$ for Basic CCS

To develop proof system for $\sim$ and CCS, we'll first look at *Basic CCS*:

- No $|, \backslash L, [f]$.
- No process constants.

So only operators are $0, a., +$.

(Non-logical) Axioms for $\sim$ and Basic CCS

$$
\begin{array}{rcll}
P + Q & = & Q + P & \text{(A1)} \\
P + (Q + R) & = & (P + Q) + R & \text{(A2)} \\
P + 0 & = & P & \text{(A3)} \\
P + P & = & P & \text{(A4)}
\end{array}
$$

Sample Proof

$$
\begin{aligned}
a.(b.0 + (c.0 + b.0)) + 0 &= a.(b.0 + (c.0 + b.0)) && \text{(A3)}\\
&= a.(b.0 + (b.0 + c.0)) && \text{(A1)}\\
&= a.((b.0 + b.0) + c.0) && \text{(A2)}\\
&= a.(b.0 + c.0) && \text{(A4)}
\end{aligned}
$$

Soundness and Completeness

$\boxed{\text{Fact}}$ Axioms A1–A4 are *sound* for $\sim$ and Basic CCS. (That is, if one proves $P = Q$ using A1–A4 then $P \sim Q$.)

Why? Can build bisimations; e.g. for any P:
$\{\langle P + P, P\rangle\} \cup \sim$ is a bisimulation.

$\boxed{\text{Fact}}$ Axioms A1–A4 are *complete* for $\sim$ and Basic CCS. (That is, $P \sim Q$ then you can prove $P = Q$ using A1–A4.)

Why? If $P \xrightarrow{a} Q$ then can prove $P = a.Q + R$ for some R.

Axiomatizing $\sim$ for Basic Parallel CCS

The next fragment of CCS: Basic Parallel CCS.

- Extends Basic CCS by including $|$ operator.
- Still no $\backslash L, [f]$ or process constants.

Note Axioms A1–A4 are sound for Basic Parallel CCS (why?); so what we need to do is add axioms for handling $|$.

The Expansion Law ...

... handles occurrences of $|$.

$\boxed{\text{Notation}}$ Let I be a finite set, $\{ P_i \mid i \in I \}$ an I-indexed set of processes. Then $\sum_{i \in I} P_i$ is the process expression given by:

$$\sum_{i \in I} P_i \quad = \quad \begin{cases} 0 & \text{if } I = \emptyset \\ P_{i_1} & \text{if } I = \{i_1\} \\ P_{i_1} + \cdots + P_{i_n} & \text{if } I = \{i_1, \ldots, i_n\} \text{ and } n \geq 2 \end{cases}$$

The Expansion Law (cont.)

(Exp) Let $P \equiv \sum_{i \in I} a_i.P_i$, $Q \equiv \sum_{j \in J} b_j.Q_j$. Then:

$$\begin{aligned} P|Q &= \sum_{i \in I} a_i.(P_i|Q) \\ &\quad + \sum_{j \in J} b_j.(P|Q_j) \\ &\quad + \sum_{\langle i,j \rangle \in \{\, \langle i,j \rangle \in I \times J | a_i = \overline{b_j} \,\}} \tau.(P_i|Q_j) \end{aligned}$$

Example

$$
\begin{array}{lll}
& a.0|(\overline{a}.0 + b.0) & \\
= & a.(0|(\overline{a}.0 + b.0)) + \overline{a}.(a.0|0) + b.(a.0|0) + \tau.(0|0) & \text{(Exp)} \\
= & a.(0 + \overline{a}.(0|0) + b.(0|0) + 0) & \\
& + \overline{a}.(a.0|0) + b.(a.0|0) + \tau.(0|0) & \text{(Exp)} \\
= & a.(\overline{a}.(0|0) + b.(0|0)) + \overline{a}.(a.0|0)) + b.(a.0|0) + \tau.(0|0) & \text{(A3)} \\
= & a.(\overline{a}.0 + b.0) + \overline{a}.(a.0|0)) + b.(a.0|0) + \tau.0 & \text{(Exp)} \\
= & a.(\overline{a}.0 + b.0) + \overline{a}.a.0 + b.a.0 + \tau.0 & \text{(Exp)}
\end{array}
$$

Note $0|0 = 0 + 0 + 0 = 0$

Soundness and Completeness for Basic Parallel CCS

- A1–A4, Exp are sound for $\sim$ and Basic Parallel CCS.

  (Why? Can build strong bisimulations!)

- A1–A4, Exp are *complete* for $\sim$ and Basic Parallel CCS.

  Why?

  – Exp allows $|$ to be "pushed inside" and eventually removed, provided arguments are of correct form.

  – Arguments of "innermost occurrences" of $|$ can be put in correct form using A1–A4!

Axiomatizing $\sim$ for Finite CCS

The next fragment of CCS: Finite CCS

- Extends Basic Parallel CCS with $\backslash L, [f]$.
- No process constants.

A1–A4, Exp are sound; just need axioms for $\backslash L, [f]$.

Axioms for $\backslash L, [f]$

$$
\begin{array}{rcll}
0\backslash L & = & 0 & \text{(Res1)} \\
(a.P)\backslash L & = & \begin{cases} 0 & \text{if } a \in L \text{ or } \overline{a} \in L \\ a.(P\backslash L) & \text{otherwise} \end{cases} & \text{(Res2)} \\
(P+Q)\backslash L & = & (P\backslash L) + (Q\backslash L) & \text{(Res3)}
\end{array}
$$

$$
\begin{array}{rcll}
0[f] & = & 0 & \text{(Rel1)} \\
(a.P)[f] & = & \hat{f}(a).(P[f]) & \text{(Rel2)} \\
(P+Q)[f] & = & (P[f]) + (Q[f]) & \text{(Rel3)}
\end{array}
$$

Soundness and Completeness for Finite CCS

- A1–A4, Exp, Res1–Res3, Rel1–Rel3 are sound for $\sim$ and Finite CCS.

  (Why? Can build strong bisimulations!)

- A1–A4, Exp, Res1–Res3, Rel1–Rel3 are also *complete* for $\sim$ and Finite CCS.
  - Can use Exp to eliminate top-level occurrences of $|$ inside $\backslash L, [f]$.
  - Can then use Res1–Res3, Rel1–Rel3 to “drive” $\backslash L, [f]$ inside $a.$, $+$ and then remove them!

Axiomatizing $\approx^C$ for Finite CCS

Notes

1. All previous axioms are sound for $\approx^C$ (why?).
2. Previous axioms permit any CCS term to be rewritten into one involving only $0, a.$ and $+$ (Basic CCS!).

To handle $\approx^C$, need to add axiom(s) for interplay between τ and the Basic CCS operators.

Is $\tau.P = P$ a good axiom?

Axiomatizing $\approx^C$: The τ Laws

$$
\begin{aligned}
a.\tau.P &= a.P && (\tau 1)\\
P + \tau.P &= \tau.P && (\tau 2)\\
a.(P + \tau.Q) &= a.(P + \tau.Q) + a.Q && (\tau 3)
\end{aligned}
$$

Soundness and Completeness for $\approx^C$, Finite CCS

- A1–A4, Exp, Res1–Res3, Rel1–Rel3, τ1–τ3 are sound for $\approx^C$ and Finite CCS.

  (Why? Can build appropriate weak bisimulations.)

- A1–A4, Exp, Res1–Res3, Rel1–Rel3, τ1–τ3 are also *complete* for $\approx^C$ and Finite CCS.

  (Why? It's magic...)

What About Full CCS?

All we need to do now is handle process constants.

Problem We can't!

- CCS is Turing powerful.
- $\sim, \approx^C$ are not recursively enumerable (r.e.).
- The set of things deducible from any axiomatization has to be r.e.

$\Rightarrow$ No complete axiomatization of $\sim, \approx^C$ can exist.

So What Do We Do?

- Inference rules for restricted classes of CCS can be defined.
- We will study one example: “Unique Fixpoint Induction”
- There are others, e.g. “Regular CCS”
- In practice, these often suffice.

$\sim$ and Unique Fixpoint Induction

$\boxed{\text{Needed}}$ Rules for proving $\sim$ between process constants, other process terms.

$\boxed{\text{Example}}$ Recall:

$$
\begin{array}{rcll}
Sem_n(0) & = & get.Sem_n(1) & \\
Sem_n(k) & = & get.Sem_n(k+1) + put.Sem_n(k-1) & (0 \leq k \leq n) \\
Sem_n(n) & = & put.Sem_n(n-1) &
\end{array}
$$

$$
\begin{array}{rcl}
Sem & = & get.Sem' \\
Sem' & = & put.Sem
\end{array}
$$

- We know $Sem_2(0) \sim Sem \,|\, Sem$ (why?)
- How can we prove $Sem_2(0) = Sem \,|\, Sem$?

Two Rules for $\sim$ and Process Constants

$$\frac{C \stackrel{\Delta}{=} P}{C = P} \text{ (Unr)}$$

$$\frac{X = P \text{ is an equation with a unique solution up to } \sim \quad Q = P[Q/X] \quad R = P[R/X]}{Q = R} \text{ (UFI)}$$

UFI?

... stands for *Unique Fixpoint Induction*

- $X = P$ is an equation, with X a variable and P a process term involving X.

  E.g. $X = a.X + b.X$

- A *solution to* $X = P$ is a process term Q such that $Q \sim P[Q/X]$ ($P[Q/X]$ is P with instances of variable X replaced by Q).

- If $X = P$ has a unique solution up to $\sim$ then any two solutions must be $\sim$!

Question What equations have unique solutions?

Equations and Solutions

Which have unique solutions?

1. $X = a.0$
2. $X = X$
3. $X = a.0 + X$
4. $X = a.X + b.X$
5. $X = a.0|X$
6. $X = a.0|b.X$

Guardedness

Definition In equation $X = P$, X is *guarded* in P if every occurrence of X in P falls inside the scope of a prefixing operator.

Theorem (Milner) If X is guarded in P then $X = P$ has a unique solution up to $\sim$.

Example

Consider:

$$\begin{array}{rcl} A & \stackrel{\Delta}{=} & a.A \\ B & \stackrel{\Delta}{=} & a.a.B \end{array}$$

We can prove $A = B$ using axioms for $\sim$ as follows.

1. X is guarded in $a.a.X$, so $X = a.a.X$ has a unique solution with respect to $\sim$

2. $$\begin{array}{rcll} A & = & a.A & \text{(Unr)} \\ & = & a.a.A & \text{(Unr)} \\ B & = & a.a.B & \text{(Unr)} \end{array}$$

   So A, B are both solutions to $X = a.a.X$

3. Therefore, by UFI, $A = B$

UFI and Systems of Equations

UFI can be generalized to *systems* of equations.

Definition

1. A system of n equations has form:

$$\begin{array}{rcl} X_0 & = & P_0 \\ & \vdots & \\ X_{n-1} & = & P_{n-1} \end{array}$$

   where $\vec{X} = \langle X_0, \ldots, X_{n-1} \rangle$ are the unknowns and $\vec{P} = \langle P_0, \ldots, P_{n-1} \rangle$ are CCS terms built up from $\vec{X}$.

2. A solution to a system of n equations $\vec{X} = \vec{P}$ is a vector of CCS terms $\vec{Q} = \langle Q_0, \ldots, Q_{n-1} \rangle$ such that for every equation $X_i = P_i$, $Q_i \sim P[Q_0/X_0, \ldots, Q_{n-1}/X_{n-1}]$.

Notions of uniqueness of solutions, guardedness can be extended to systems of equations.

Example: Prove $Sem_2(0) = Sem \,|\, Sem$

1. Consider the equation system E:

$$\begin{array}{rcl} X_0 & = & get.X_1 \\ X_1 & = & get.X_2 + put.X_0 \\ X_2 & = & put.X_1 \end{array}$$

2. Prove that $\langle Sem_2(0), Sem_2(1), Sem_2(2)\rangle$ is a solution to E

3. Prove that $\langle Sem \,|\, Sem, Sem' \,|\, Sem, Sem' \,|\, Sem'\rangle$ is a solution to E

UFI and $\approx^C$

Unique Fixpoint Indunction for $\approx^C$:

$$\boxed{\dfrac{X = P \text{ is an equation with a unique solution up to } \approx^C \quad Q = P[Q/X] \quad R = P[R/X]}{Q = R}} \text{ (UFI)}$$

$\boxed{\text{Question}}$ What equations have unique solutions?

Guardedness Is Not Enough

What are the solutions to $X = \tau.X$?

"Strong" Guardedness Is Not Enough

What are solutions to $X = (a.X|\overline{a}.X)\backslash\{a\}$?

Strong Sequential Guardedness

Definition Variable X is *strongly sequential* in P if every occurrence of X appears within at least one prefixing operator whose action is visible (i.e. not τ) and is not inside any parallel composition operator.

Examples

1. X is not strongly sequential in $\tau.X$.
2. X is strongly sequential in $a.X$ if $a \neq \tau$.
3. X is not strongly sequential in $a.X \,|\, \overline{a}.X$.
4. X is not strongly sequential in $a.X \,|\, b.X$.
5. X is strongly sequential in $a.X + b.X$ if $a, b \neq \tau$.

Theorem (Milner89) Let X is strongly sequential in P. Then $X = P$ has a unique solution up to $\approx^C$.

Less restrictive conditions also possible:

E. Brinksma. On the uniqueness of fixpoints modulo observation congruence. In R. Cleaveland, editor, *CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 47–61, Stony Brook, NY, August 1992. Springer-Verlag, Heidelberg.

Computing $\sim$, $\approx$

For finite-state systems, both $\sim$, $\approx$ may be computed automatically.

$\boxed{\text{How?}}$

Basic approach is as follows.

1. Build LTS containing of all states reachable from P, Q; call set of states in LTS $\mathcal{S}_{P,Q}$.
2. Compute set of *equivalence classes* for $\approx / \sim$ over $\mathcal{S}_{P,Q}$.
3. See if P, Q belong to same equivalence class.

A Review of Equivalence Classes

Given a set S and an equivalence relation $\mathcal{R} \subseteq S \times S$, one can use $\mathcal{R}$ to partition S into *equivalence classes*.

Definition Given S, equivalence relation $\mathcal{R}$, $S' \subseteq S$ is an equivalence class with respect to $\mathcal{R}$ if the following hold.

- For all $s, s' \in S'$, $s \,\mathcal{R}\, s'$.
- For all $s \in S$, if $s \,\mathcal{R}\, s'$ some $s' \in S'$ then $s \in S'$.

That is, S' represents a maximal “clump” of equivalent elements in S.

Notation If $s \in S$ then $[s]_{\mathcal{R}} \stackrel{\Delta}{=} \{\, s' \in S \mid s \,\mathcal{R}\, s' \,\}$ is the equivalence class of s.

Notes about Equivalence Classes

Let S be a set, $\mathcal{R} \subseteq S \times S$ be an equivalence relation.

1. For any two equivalence classes S_1, S_2, either $S_1 = S_2$ or $S_1 \cap S_2 = \emptyset$.
2. Every element $s \in S$ belongs to exactly one equivalence class, namely, $[s]_{\mathcal{R}}$.
3. $s_1 \, \mathcal{R} \, s_2$ iff s_1, s_2 belong to the same equivalence class.

So How Does This Help Us Compute $\sim / \approx$?

- $\mathcal{S}_{P,Q} \subseteq \mathcal{P}$, and since $\sim / \approx$ are equivalences over $\mathcal{P}$, they are equivalences over $\mathcal{S}_{P,Q}$ also.
- If P, Q in same equivalence class of $\sim / \approx$ over $\mathcal{S}_{P,Q}$, then they are equivalent; otherwise, they are not.
- So ... if we can compute equivalence classes of $\sim / \approx$ over $\mathcal{S}_{P,Q}$, we can determine whether or not P, Q are strongly/observationally equivalent!

Thus, if we can compute the relevant equivalence classes, we can compute $\sim / \approx$. To see how we do this we'll focus first on $\sim$.

An Iterative Characterization of $\sim$

$\boxed{\text{Note}}$ Definition of $\sim$ can be given for arbitrary LTSs (i.e. triples $\langle \mathcal{S}, Act, \longrightarrow \rangle$), not just CCS.

Assume LTS $\langle \mathcal{S}, Act, \longrightarrow \rangle$ satisfies: $\mathcal{S}, Act$ are finite.

Then $\sim \subseteq \mathcal{S} \times \mathcal{S}$ is the same as $\bigcap_{i=0}^{\infty} \sim_i$, where:

- $P \sim_0 Q$ holds all P, Q.
- $P \sim_{i+1} Q$ holds if for all $a \in Act$:
  1. $P \xrightarrow{a} P'$ implies $Q \xrightarrow{a} Q'$ some Q' with $P' \sim_i Q'$.
  2. $Q \xrightarrow{a} Q'$ implies $P \xrightarrow{a} P'$ some P' with $P' \sim_i Q'$.

Facts About the $\sim_i$

(Remember that $\mathcal{S}$ is finite.)

1. Each $\sim_i$ is an equivalence relation.
2. For all i, $\sim_{i+1} \subseteq \sim_i$. (In other words, if $P \sim_{i+1} Q$ then it follows that $P \sim_i Q$ holds also.)
3. For all $i \geq |\mathcal{S}|$, $\sim_i = \sim_{i+1}$.
4. If $\sim_{i+1} = \sim_i$, then $\sim_i$ is a strong bisimulation.
5. $\sim \subseteq \sim_i$ for all i.

Believe It Or Not ...

... we have most of the information necessary to compute $\sim$!

- Use *equivalence classes* (i.e. lists of subsets of $\mathcal{S}$) to represent $\sim_i$.
- Can then use a while loop like the following:

```
R := [𝒮]; // Set R to ~0
newR := refine(R); // Set newR to ~1
while (R ≠ newR) do
  R := newR;
  newR := refine(R);
```

When loop terminates, R contains equivalence classes of $\sim$.

What Does `refine` Do?

$\boxed{\text{Idea}}$ If R $= \sim_i$ then `refine(R)` should return $\sim_{i+1}$.

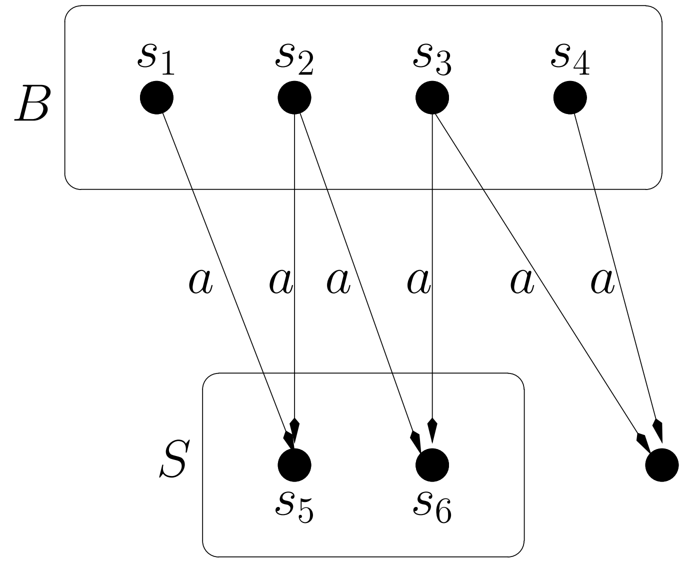
$\boxed{\text{How?}}$ By using notion of *partition refinement*.

Terminology:

- *Partition* is another name for “list of equivalence classes”.
- *Block* is another name for “equivalence class”.

Partition Refinement

Suppose B, S are blocks in R with $a \in Act$ an action such that:



Then $s_4 \sim_i s_1$, but $s_4 \not\sim_{i+1} s_1$!

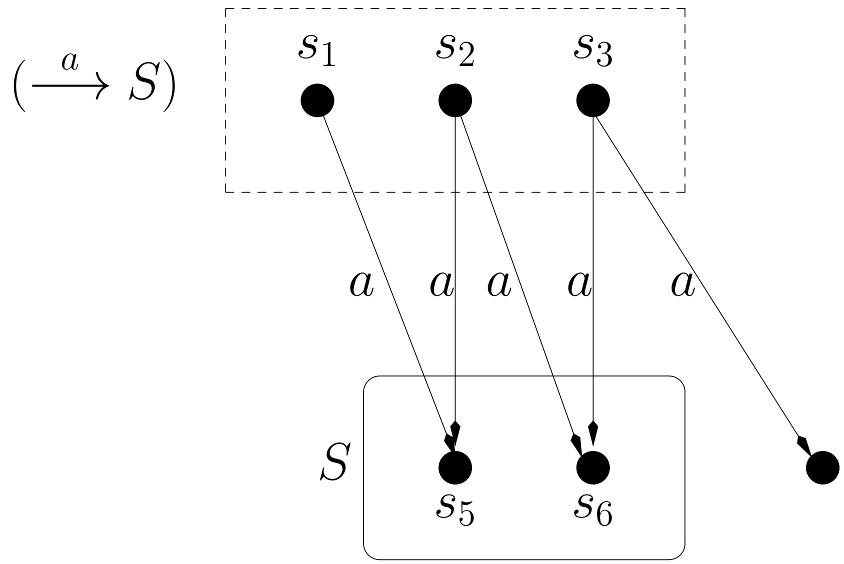
$\Rightarrow$ In $\sim_{i+1}$, B should be *split* into (at least) two pieces:

states with an a-transition into S, and states without.

(S called a *splitter*.)

Defining `split` Operation

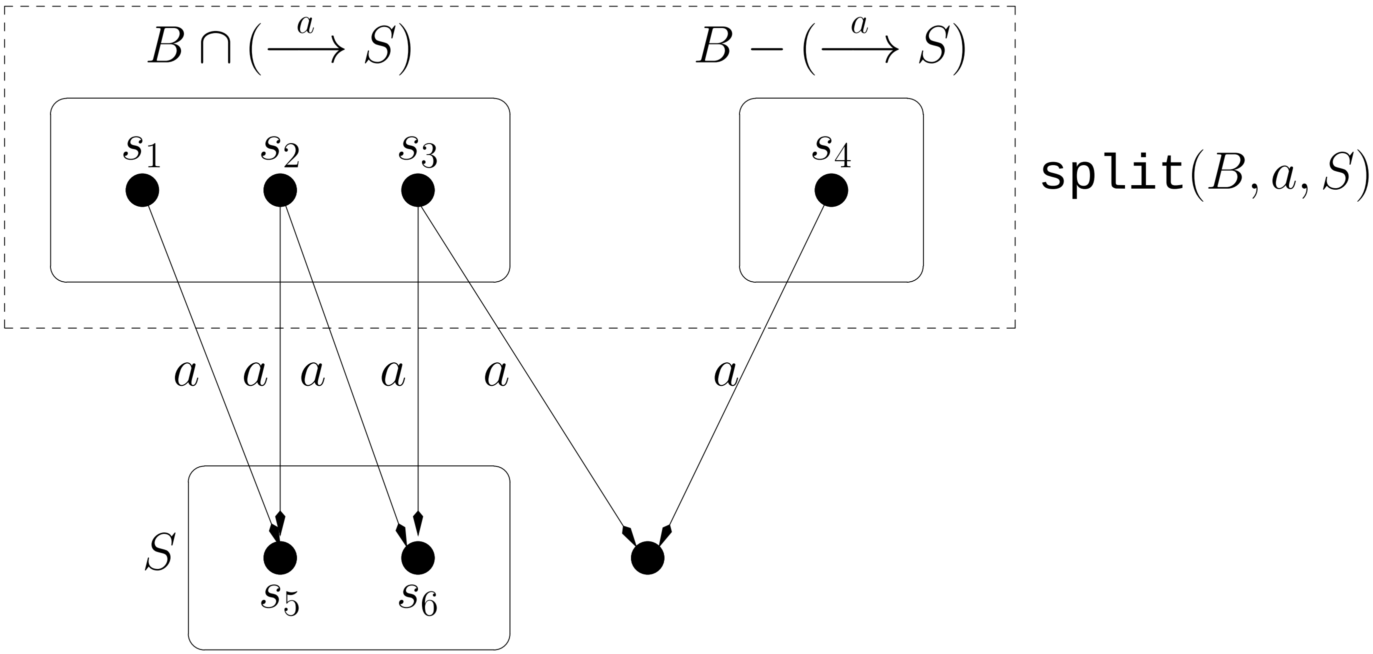
Let $B, S \subseteq \mathcal{S}$, $a \in Act$; define $(\xrightarrow{a} S) = \{\, s \mid \exists s' \in S. s \xrightarrow{a} s' \,\}$



$$\texttt{split}(B, a, S) = \begin{cases} [B \cap (\xrightarrow{a} S), B - (\xrightarrow{a} S)] & \text{if both nonempty} \\ [B] & \text{otherwise} \end{cases}$$

Example

Consider previous example: $\mathtt{split}(B, a, S)$ given as follows.



Splitting Partitions

`split` can be lifted to partitions (lists of blocks).

Let Π be a partition.

`split`(Π, a, S) =
    `res := [ ];`
    `foreach` B `in` Π `do`
        `res := res @ split`(B, a, S)`;`
    `return res;`

(That is, apply `split`$(-, a, S)$ to every block in Π and "collect" the results into a single partition. @ is "append" (i.e. list merge).)

Splitting Over All Actions

Similarly, we can define `all-split` that splits a partition with respect to a splitter and *all* actions.

```
all-split (Π, S) =
  R := Π;
  foreach a ∈ Act do
    R := split(R, a, S);
  return R;
```

Does order of actions matter? No....

Splitting a Partition With Respect to Another

We can now lift the notion of “splitting a partition” to a list of “splitters”: just split with respect to all splitters!

```
part-split(Π1, Π2) =
  R := Π1;
  foreach S ∈ Π2 do
    R := all-split(R, S);
  return R;
```

Then `refine(R)` is just `part-split (R,R)`!

Complexity Analysis

- $\texttt{all-split}(\Pi, S)$ can be implemented in $O(\Sigma_{a \in Act}|(\stackrel{a}{\longrightarrow} S)|)$. (How?)
- So $\texttt{refine}(\mathtt{R})$ takes $O(|\longrightarrow|)$. (Why?)
- Loop can iterate at most $|\mathcal{S}|$ times. (Why?)
- So complexity is $O(|\mathcal{S}| \cdot |\longrightarrow|)$!

Optimizations

- If S' is a yet-to-be-processed splitter in R that is itself split by another splitter S, then there is no need to split with respect to S'; just use the “children” of S'.

  (Note: this does not affect complexity, but it simplifies implementation. Just maintain a list of splitters to be processed!)

- By doing some extra work, $O(\text{log}(|\mathcal{S}|) \cdot | \longrightarrow |)$ possible.

Computing $P \sim Q$

1. Compute $\mathcal{S}_{P,Q}$ = CCS expressions reachable from P, Q.
2. Compute equivalence classes of $\mathcal{S}_{P,Q}$ with respect to $\sim$.
3. Determine whether P, Q belong to same equivalence class.

Computing $P \approx Q$

... combine *LTS transformation* with approach for computing $\sim$!

- $\langle \mathcal{S}_{P,Q}, Act, \longrightarrow \rangle$ forms an LTS.
- So does $\langle \mathcal{S}_{P,Q}, \widehat{Act}, \Longrightarrow \rangle$.
- We can transform $\langle \mathcal{S}_{P,Q}, Act, \longrightarrow \rangle$ into $\langle \mathcal{S}_{P,Q}, \widehat{Act}, \Longrightarrow \rangle$.

(Here $\widehat{Act} = \{ \widehat{a} \mid a \in Act \}$.)

Computing $P \approx Q$ (cont.)

So we can compute $P \approx Q$ as follows.

1. Compute $\mathcal{S}_{P,Q}$ = CCS expressions reachable from P, Q.
2. Build $\langle \mathcal{S}_{P,Q}, \widehat{Act}, \Longrightarrow \rangle$ from $\langle \mathcal{S}_{P,Q}, Act, \longrightarrow \rangle$.
3. Compute equivalence classes of $\langle \mathcal{S}_{P,Q}, \widehat{Act}, \Longrightarrow \rangle$ with respect to $\sim$.
4. Determine whether P, Q belong to same equivalence class.

Why Does This Work?

... because $\approx$ is the largest relation such that whenever $P \approx Q$ then the following hold for all $a \in Act$.

1. $P \stackrel{\hat{a}}{\Longrightarrow} P'$ implies $Q \stackrel{\hat{a}}{\Longrightarrow} Q'$ some Q' such that $P' \approx Q'$.

2. $Q \stackrel{\hat{a}}{\Longrightarrow} Q'$ implies $P \stackrel{\hat{a}}{\Longrightarrow} P'$ some P' such that $P' \approx Q'$.

There Are Other Process Algebras...

1. CSP: like CCS, but multiway rendezvous is basic notion of synchronization.
2. ACP: like CCS except that notion of synchronization is parameterized.
3. LOTOS: CCS/CSP-like ISO standard.
4. SCCS: synchronous systems.

All, however, share emphasis on: operational semantics, equational reasoning.