CSP: Communicating Sequential Processes

Overview

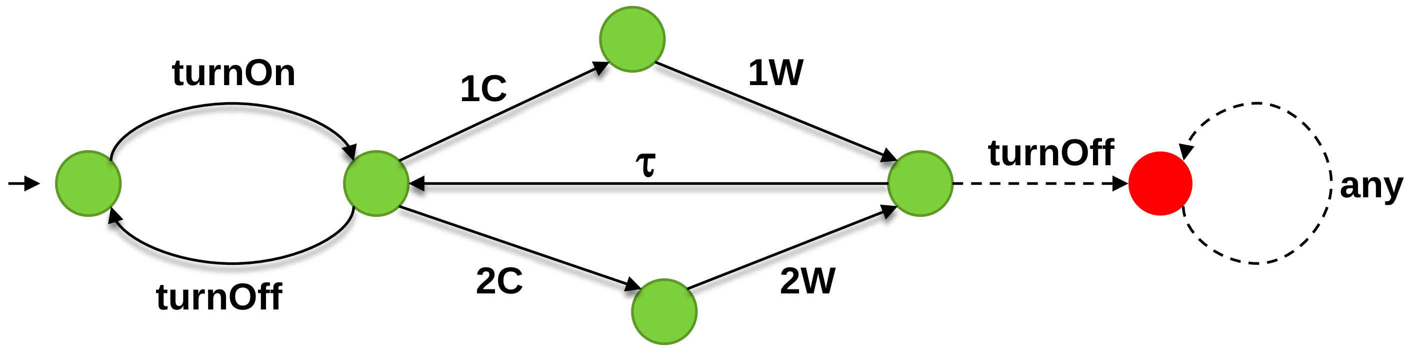
- Computation model and CSP primitives
- Refinement and trace semantics
- Automaton view
- Refinement checking algorithm
- Failures Semantics

CSP

- Communicating Sequential Processes, introduced by Hoare, 1978.
- Abstract and formal event-based language to model concurrent systems.
- Elegant, with refinement based reasoning.

$$\boldsymbol{Senseo} = turnOn \rightarrow \boldsymbol{Active}$$

$$\boldsymbol{Active} = (turnOff \rightarrow \boldsymbol{Senseo}) \quad (1c \rightarrow 1w \rightarrow \boldsymbol{Active}) \quad (2c \rightarrow 2w \rightarrow \boldsymbol{Active})$$



References

- The CSP Archieve: http://vl.fmnet.info/csp/ (down??)
- Quick info at Wikipedia.
- Communicating Sequential Processes, Hoare, Prentice Hall, 1985.

  3rd most cited computer science reference ☺

  Renewed edition by Jim Davies, 2004.

  Available free!
- Model Checking CSP, Roscoe, 1994.

Computation model

- A concurrent *system* is made of a set of interacting *processes*.
- Each process sequentially produces *events*. Each event is atomic. Examples:
    - turnOn, turnOff, Play, Reset
    - lockAcquire, lockRelease
- Some events are internals → not observable from outside.
- There is no notion of variables, nor data. A process is abstractly decribed by the sequences of events that it produces.

Computation model

- Multiple processes can *synchronize* on an event, say a.
  - They will wait each other until all synchronizing processes are ready to execute a.
  - Then they will simultaneously execute a.
  - As in :

$$a \rightarrow STOP \;\; ||_{\{a\}} \;\; x \rightarrow a \rightarrow STOP$$

The 1st process will have to wait until the 2nd has produced x.

Some notation first

- Names :

A,B,C	$\rightarrow$ alphabets (sets of events)
a,b,c	$\rightarrow$ events (actions)
P,Q,R	$\rightarrow$ processes

- Formally for each process we also specify its alphabet, but here we will usually leave this implicit.

- αP denotes the alphabet of P.

CSP constructs

- We'll only consider simplified syntax:

```
Process  ::=  STOP
           |  Event  →  Process
           |  Process  []  Process
           |  Process  |¯|  Process
           |  Process  ||  Process
           |  Process / Alphabet
           |  ProcessName
```

- *Process definition:*

ProcessName "=" *Process*

STOP, sequence, and recursion

- Some simple primitives :
  - STOP // as the name says
  - $a \rightarrow P$ // do a, then behave as P
- Recursion is allowed, e.g. :

  $$\text{Clock} \;=\; \text{tick} \rightarrow \text{Clock}$$

  Recursion must be ‘guarded’ (no left recursion thus).

Internal choice

- We also have *internal / non-deterministic* choice: P |¯| Q, as in :

$$\boxed{R_1 \;=\; (a \rightarrow P) \;\sqcap\; (b \rightarrow Q)}$$

  R_1 behave as either:

  $a \rightarrow P$ or $b \rightarrow Q$

  but the choice is decided internally by R_1 itself. From outside it is as if R_1 makes a non-deterministic choice.

- R_1 may therefore *deadlock* (e.g. the environment only offers a, but R_1 have decided that it wants to do b instead).

External choice

- Denoted by P   Q

  Behave as either P or Q. The choice is decided by the environment.

- Ex:

$$\boxed{R_2 \quad = \quad (a \rightarrow P) \qquad\qquad (b \rightarrow Q)}$$

  R_2 behaves as either:

  $a \rightarrow P$ or $b \rightarrow Q$

  depending on the actions *offered* by the environment (e.g. think a,b as representing actions by a user to push on buttons).

External choice

- However, it can degenerate to non-deterministic choice:

$$R_3 \quad = \quad (a \rightarrow P) \qquad\qquad (a \rightarrow Q)$$

Parallel composition

- Denoted by P || Q

  This denotes the process that behaves as the *interleaving* of P and Q, but *synchronizing* them on $\alpha P \cap \alpha Q$.

  Example:

  $$R \;=\; (a_1 \rightarrow b \rightarrow STOP) \;||\; (a_2 \rightarrow b \rightarrow STOP)$$

  This produces a process that behaves as either of these :

  $$a_1 \rightarrow a_2 \rightarrow b \rightarrow STOP$$

  $$a_2 \rightarrow a_1 \rightarrow b \rightarrow STOP$$

  (Notice the interleaving on a_1,a_2 and synchronization on b).

Hiding (abstraction)

- Denoted by P / A

  Hide (internalize) the events in A; so that they are not visible to the environment.

  Example:

$$R \quad = \quad (a_1 \rightarrow b \rightarrow STOP) \;||\; (a_2 \rightarrow b \rightarrow STOP)$$

$$R \;/\; \{b\} \quad = \quad (a_1 \rightarrow a_2) \qquad (a_2 \rightarrow a_1)$$

- In particular:

$$(P \;||\; Q) \;\;/\;\; (\alpha P \cap \alpha Q)$$

  is the parallel composition of P and Q, and then we internalize their synchronized events.

Specifications and programs have the same status

- That is, a specification is expressed by another CSP process :

$$\boxed{\textit{\textbf{SenseoSpec}} \;=\; (\,1c \rightarrow 1w) \quad (\,2c \rightarrow 2w) \;\rightarrow\; \textit{\textbf{SenseoSpec}}}$$

- More precisely, when events not in {1c,1w,2c,2w} are abstracted away, our Senseo machine should behave as the above SenseoSpec process. This is expressed by *refinement* :

$$\boxed{\textit{\textbf{SenseoSpec}} \;\leq\; \textit{\textbf{Senseo}} \,/\, \{\, \textit{turnOn}, \textit{turnOff}\, \}}$$

Cannot be conveniently expressed in temporal logic. Conversely, CSP has no native temporal logic constructs to express properties.

<u>Refinement relation</u>: P ≤ Q means that Q is at least as good as P. What this exactly entails depends on our intent. In any case, we usually expect a refinement relation to be <u>preorder</u> ☺

Monotonicity

- A relation $\leq$ (over A) is a *preorder* if it is *reflexive* and *transitive* :

  1. $P \leq P$
  2. $P \leq Q$ *and* $Q \leq R$ *implies* $P \leq R$

- A function $F{:}A \rightarrow A$ is *monotonic* roughly if its value increases if we increase its argument.

  More precisely it is monotonic wrt to a relation $\leq$ iff

  $$P \leq Q \quad \Rightarrow \quad F(P) \leq F(Q)$$

- Analogous definition if F has multiple arguments.

Monotonicity & Compositionality

- Suppose we have a preorder $\leq$ over CSP processes, acting as a refinement relation.

$\varphi \leq P \quad \rightarrow \quad$ *express P satisfies the specification* φ

- A monotonic || would give us this result, which you can use to decompose the verification of a system to component level, and avoiding, in theory, state explosion:

$$\frac{\varphi_1 \leq P \quad , \quad \varphi_2 \leq Q \qquad \varphi \leq \varphi_1 \parallel \varphi_2}{\varphi \leq P \parallel Q}$$

So, can we find a notion of refinement such that all CSP constructs are monotonic ??

Many formalisms for concurrent systems do not have this. CSP monotonicity is mainly due to its level of abstraction.

Trace Semantics

- *Idea*: abstractly consider two processes to be equivalent if they generate the same traces.

- Introduce **traces(P)**

  the set of all *finite traces* (sequences of events) that P can produce.

- E.g. **traces**(a $\rightarrow$ b $\rightarrow$ STOP) = { <>, <a> , <a,b> }

- Simple semantics of CSP processes
- But it is oblivious to certain things.
- Still useful to check safety.
- Induce a natural notion of refinement.

Trace Semantics

- We can define “traces” inductively over CSP operators.
- **traces** STOP = $\{ \langle\rangle \}$
- **traces** $(a \rightarrow P) = \{ \langle\rangle \} \cup \{ \langle a\rangle \,\hat{}\, s \mid s \in traces(P) \}$

Trace Semantics

- If s is a trace, $s|_A$ is the trace obtained by throwing away events *not* in A.

  Pronounced “s *restricted* to A”.

  Example : $\langle a,b,b,c\rangle \mid \{a,c\} = \langle a,c\rangle$

- Now we can define:

  traces (P/A) $= \{ s|_{(\alpha P - A)} \mid s \in \text{traces}(P) \}$

Trace Semantics

- If A is an alphabet, A* denote the set of all traces over the events in A. E.g. <a,b,b> $\in$ {a,b}*, and <a,b,b> $\in$ {a,b,c}*; but <a,b,b> $\notin$ {b}*.

- **traces** (P || Q)

  =

  { s | s $\in$ $(\alpha P \cup \alpha Q)^*$,

  $s|_{\alpha P} \in$ traces(P) and $s|_{\alpha Q} \in$ traces(Q)

  }

Example

- Consider :

  $P = a_1 \rightarrow b \rightarrow STOP$ // $\alpha P = \{a_1,b\}$

  $Q = a_2 \rightarrow b \rightarrow STOP$ // $\alpha Q = \{a_2,b\}$

- **traces**$(P||Q) = \{ \langle\rangle , \langle a_1\rangle , \langle a_1,a_2\rangle, \langle a_1,a_2,b\rangle, \ldots \}$

  Notice that e.g. :

  $\langle a_1,a_2,b\rangle \mid_{\alpha P} \in$ **traces**(P)

  $\langle a_1,a_2,b\rangle \mid_{\alpha Q} \in$ **traces**(Q)

Trace Semantics

- **traces**(P   Q)   =   traces(P) ∪ traces(Q)
- **traces**(P |⎺| Q)   =   traces(P) ∪ traces(Q)
- So in this semantics you *can't* distinguish between internal and external choices.

Traces of recursive processes

- Consider

$$P = (a \to a \to P) \;\square\; (b \to P)$$

- How to compute **traces**(P) ? According to defs:

$$\begin{aligned}\textbf{traces}(P) = \{ \langle\rangle, \langle a\rangle \} &\cup \{ \langle a,a\rangle \,\hat{}\, t \mid t \in \textbf{traces}(P) \} \\ &\cup \{ \langle b\rangle \,\hat{}\, t \mid t \in \textbf{traces}(P) \}\end{aligned}$$

- Define traces(P) as the smallest solution of the above equation.

Trace Semantics

- We can now define refinement as trace inclusion. Let P, Q be processes over the *same* alphabet:

$$P \leq Q \quad = \quad traces(P) \supseteq traces(Q)$$

which implies that Q won't produce any 'unsafe trace' unless P itself can produce it.

- Moreover, this relation is obviously a preorder.

- Theorem:

All CSP operators are monotonic wrt this trace-based refinement relation.

Verification

- Because specification is expressed in terms of refinement :

  $$\varphi \leq P$$

  verification in CSP amounts to *refinement checking*.

- In the trace semantics it amounts to checking:

  $$traces(\varphi) \supseteq traces(P)$$

  We can't check this directly since the sets of traces are typically infinite.

- If we view CSP processes as automata, we can do this checking with some form of model checking.

Automata semantic

- Represent CSP process P with an automaton M_P that generates the same set of traces.

- Such an automaton can be systematically constructed from the P’s CSP description.
  - However, the resulting M_P may be non-deterministic.
  - Convert it to a deterministic automaton generating the same traces
    - Comparing deterministic automata are easier as we later check refinement.
    - There is a standard procedure to convert to deterministic automaton.

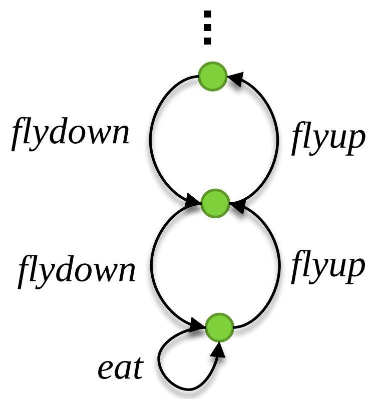
- Things are however more complicated as we later look at failures semantic.

Only finite state processes

- Some CSP processes may have infinite number of states, e.g. $Bird_0$ below:

$$Bird_0 = (flyup \rightarrow Bird_1) \,\square\, (eat \rightarrow Bird_0)$$

$$Bird_{i+1} = (flyup \rightarrow Bird_{i+2}) \,\square\, (flydown \rightarrow Bird_i)$$



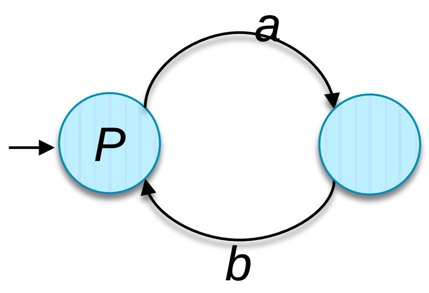
- We will only consider finite state processes.

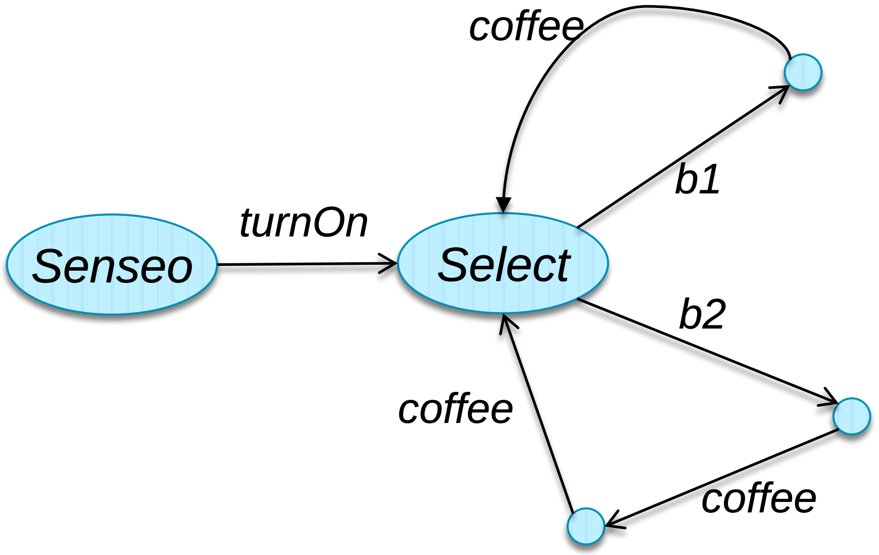
Automaton semantics

$P \;=\; a \rightarrow b \rightarrow P$

$\textbf{\textit{Senseo}} \;=\; \textit{turnOn} \rightarrow \textbf{\textit{Select}}$

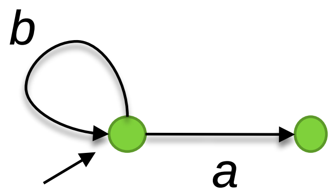
$\textbf{\textit{Select}} \;=\; \textit{b1} \rightarrow \text{coffee} \rightarrow \textbf{\textit{Select}}$
$\square$
$\textit{b2} \rightarrow \text{coffee} \rightarrow \text{coffee} \rightarrow \textbf{\textit{Select}}$



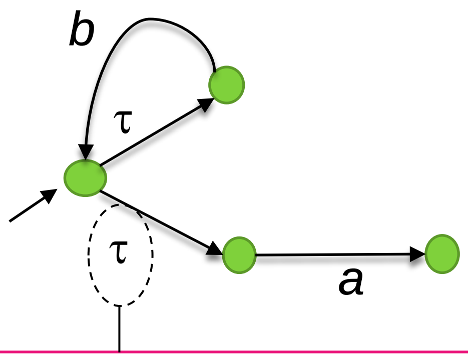


No distinction between ext. and int. choice

$P \;=\; (\, a \rightarrow STOP\,) \qquad (\, b \rightarrow P\;)$



$P \;=\; (a \rightarrow STOP\,) \;\; \sqcap \;\; (b \rightarrow P\,)$



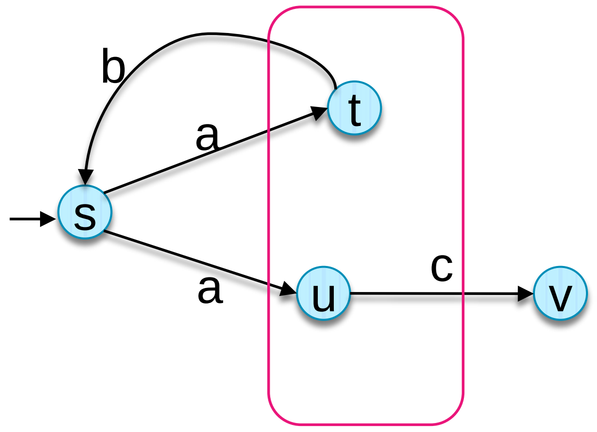
Internal action, representing internal decision in choosing between a and b.

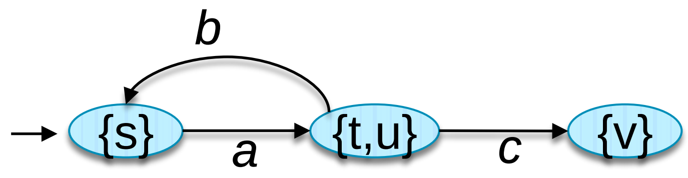
However, since in trace semantics we don't see the difference between □ and ⊓ anyway, so for we define their automata to be the same.

Converting to deterministic automaton

“□” can still lead to an implicit non-determinism. But this should be indistinguishable in the trace semantic, so convert it to a deterministic automaton, essentially by merging end-states with common events. The transformation preserves traces.

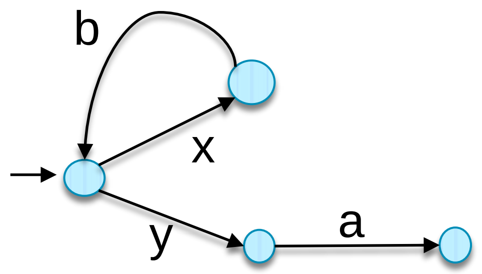
$$P \;=\; (\,a \to c \to STOP\,) \qquad (\,a \to b \to P\,)$$



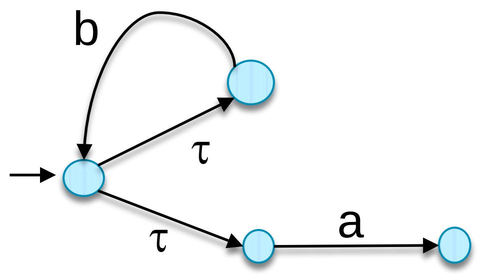


Hiding

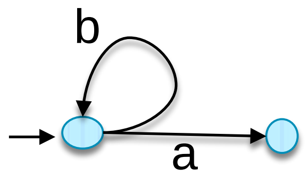
P :



P / {x,y} :

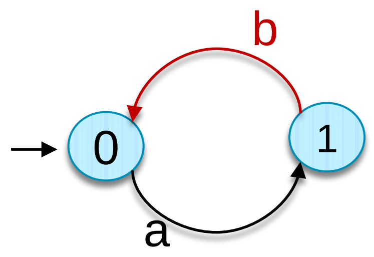


convert it to a deterministic version.

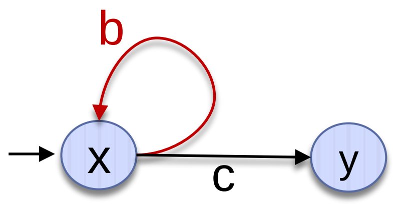


Parallel comp.

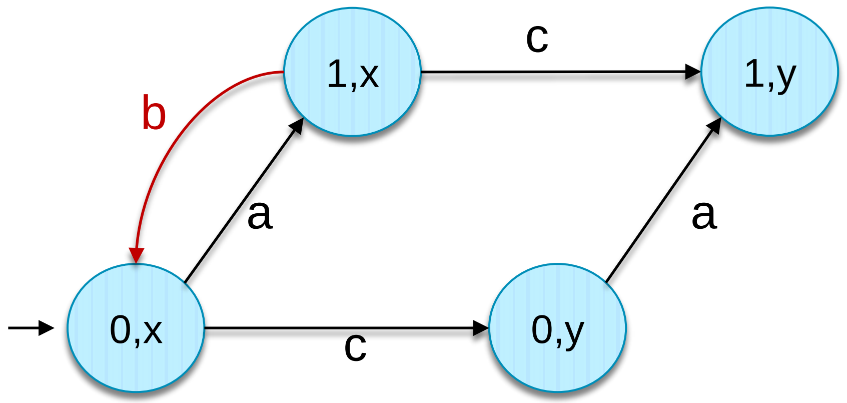
$P \; = \; a \rightarrow b \rightarrow P$



$Q \; = \; (\, b \rightarrow Q \,) \qquad (\, c \rightarrow STOP \,)$



$P \,||\, Q$, *common alphabet is { b } :*



Checking trace refinement

- Formally, we will represent a *deterministic* automaton M by a tuple (S,s_0,A,R), where:

  - S M's set of states
  - s_0 the initial state
  - A the alphabet (set of events) ; every transition in M is labeled by an event.

  - R : S→A→pow(S) encoding the transitions in M.

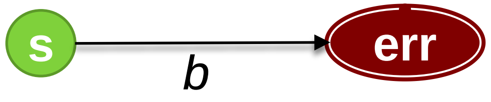
    - *Deterministic*: R s a is either ∅ or a singleton. Else non-deterministic.
    - "R s a = {t}" means that M can go from state s to t by producing event a.
    - "R s a = ∅" means that M can't produce a when it is in state s.

Checking trace refinement

- Let $M_P = (S,s_0,A,R)$ and $M_Q = (S,t_0,B,S)$ be deterministic (!) automata representing respectively processes P and Q; they have the same alphabet. We want to check:

$$\boxed{\textbf{traces}(P) \supseteq \textbf{traces}(Q)}$$

- Imagine first that we modify M_P to K_P by adding an *error state* **err** to M_P. For every state $s \in S$, we add a transition s —b—> err, if b is <u>not</u> a possible next event on the state s:



Checking trace refinement

- Theorem:

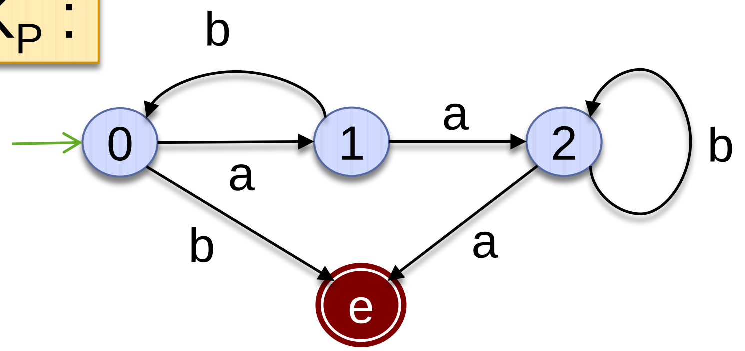
Let $u \in A^*$; $u \notin$ **traces**(P) iff it drives K_P to an error state.

- Now construct $K_P \cap M_Q$. Theorem:

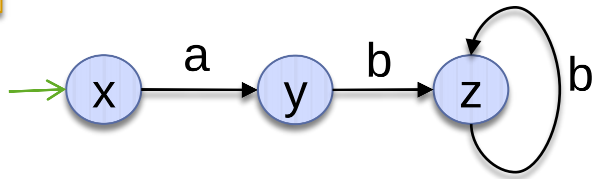
traces$(P) \supseteq$ **traces**(Q) iff $\neg$ **(**$\exists t$:: (err,t) $\in K_P \cap M_Q$)

Example

K_P :

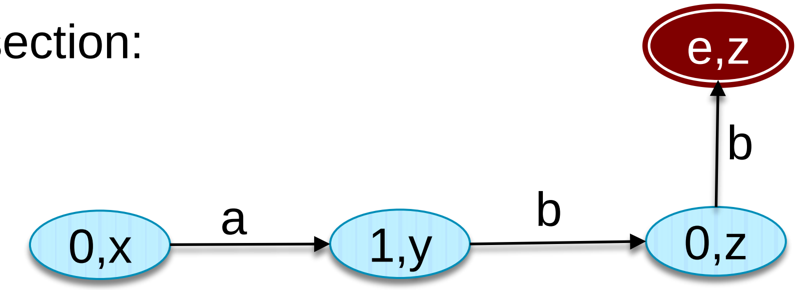


M_Q:



The intersection:

error state is reachable!



However, notice that (s,t) in $K_P \cap M_Q$ can make a step to an error state iff $\neg(\text{initials}_P(s) \supseteq \text{initials}_Q(t))$.

Checking trace refinement

- For $s \in S$, let initials$_P$(s) be the set of P's possible next events when it is in the state s:

$$\textbf{initials}_P(s) \quad = \quad \{\, a \mid R\, s\, a \neq \emptyset \,\}$$

- Note that you can get to the error state in $K_P \cap M_Q$ if and only if there is a state (s,t) in $M_P \cap M_Q$ such that:

$$\neg(\, \textbf{initials}_P(s) \supseteq \textbf{initials}_Q(t)\,)$$

- This gives you an algorithm to check refinement → construct the intersection automaton, and check the above condition on every state in the intersection. → you can also construct it lazily.
-

Refinement Checking Algorithm

```
checked = $\emptyset$ ;

pending = { $(s_0,t_0)$ } ;

**while** pending $\neq \emptyset$ **do** {

        get and remove an (s,t) from pending ;

         **if** initials(s) $\supseteq$ initials(t) **then** {

            pending := pending
                       $\cup$
                       ( { (s',t') | ($\exists$a. s' $\in$ R s a $\wedge$ t' $\in$ R t a ) } / checked ) ;

            checked := {(s,t)} $\cup$ checked }

        **else** error!
 }
```

More refined semantics?

- Unfortunately, in trace-based semantics these are equivalent :

$$P = (a \rightarrow STOP) \, \Box \, (b \rightarrow STOP)$$

$$Q = (a \rightarrow STOP) \sqcap (b \rightarrow STOP)$$

- But Q may deadlock when we put it with e.g. E = a $\rightarrow$ STOP; whereas P won’t.

Refusal

- Suppose $\alpha P = \{a,b\}$, then:

  $P = a \rightarrow STOP$

  will *refuse* to synchronize over b.

- $Q = (a \rightarrow STOP) \;\Box\; (b \rightarrow STOP)$ will refuse neither a nor b.

- $R = (a \rightarrow STOP) \sqcap (b \rightarrow STOP)$

  may refuse to sync over a, or b, not over both (if the env can do either a or b, but leave the choice to P).

Refusal

- An *offer* to P is a set of event choices that the environment (of P) is offerring to P as the first event to synchronize; the choice is up to P.
- So we define a *refusal* of P as an offer that P won't be able to accept (it can't sync over any event in the offer).
- **refusals**(P) = the set of all P's refusals.

$Q = (a \rightarrow STOP) \;\Box\; (b \rightarrow STOP)$

refusals(Q) = $\{ \emptyset \}$

$R = (a \rightarrow STOP) \sqcap (b \rightarrow STOP)$

refusals(R) = $\{ \emptyset, \{a\}, \{b\} \}$

Refusals

- Assuming alphabet A
- **refusals** (STOP) = $\{ X \mid X \subseteq A \}$
- **refusals** $(a \rightarrow P)$ = $\{ X \mid X \subseteq A \land a \notin X \}$

refuse any offer that does not include *a*

Refusals

- **refusals** (P [] Q) = **refusals**(P) $\cap$ **refusals**(Q)

  $P = a \rightarrow \ldots$

  $Q = b \rightarrow \ldots$

  Assuming alphabet {a,b}

- **refusals** (P $\sqcap$ Q) = **refusals**(P) $\cup$ **refusals**(Q)

  In the above example:

  - may refuse $\emptyset$, {a}, {b}
  - won't refuse {a,b}

Refusals of ||

- **refusals**(P || Q) = { X ∪ Y | X∈refusals(P) /\ Y∈refusals(Q) }

$\alpha P = \{a,b,x\}$

P = a → ...

refusals: { b,x } and all its subsets

refusals: { d,x } and all its subsets

$\alpha Q = \{c,d,x\}$

Q = c → ...

refuse common actions or
other Q's non-common actions.

P||Q = (a → c → ...) [] (c → a → ...)

refusals: {b,d, x } and all its subsets

Refusals of ||

- **refusals**(P || Q) = { X ∪ Y | X∈refusals(P) /\ Y∈refusals(Q) }

αP = {a,b,x}

P = x → ...

refusals: {a,b} + subsets

refusals: {c,d} + subsets

αQ = {c,d,x}

Q = x → ...

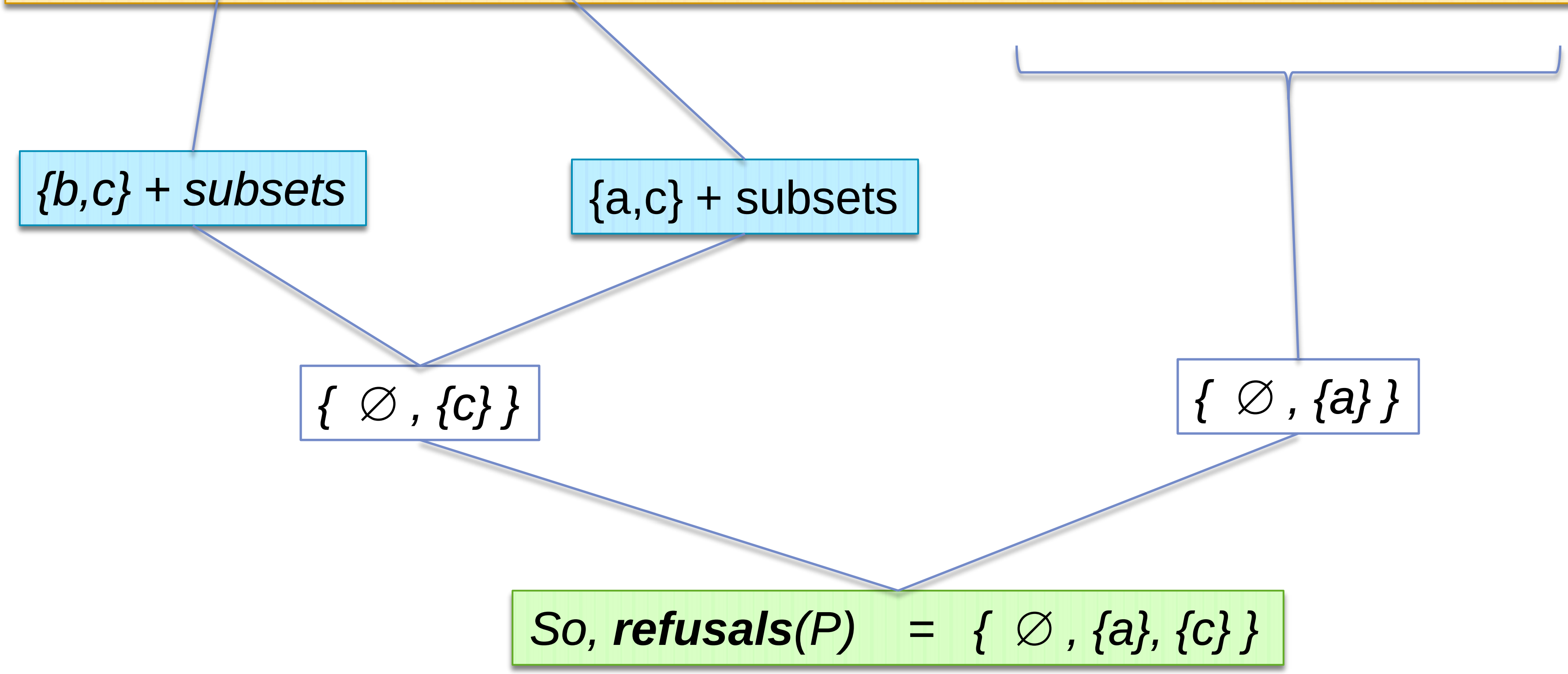
P||Q = x → ...

refusals: {a,b,c,d} + subsets

Example

- What is the refusals of this? Assume *{a,b,c}* as alphabet.

$$P \;=\; ((a \rightarrow STOP)\ [\,]\ (b \rightarrow STOP)) \;\sqcap\; ((b \rightarrow STOP)\ [\,]\ (c \rightarrow STOP))$$



So, ***refusals****(P)* = *{ ∅ , {a}, {c} }*

Refusals after s

- Define:

$\boldsymbol{refusals}(P/s) \quad = \quad \textit{the refusals of P after producing the trace s.}$

- Example, with alphabet $\alpha P = \{a,b\}$:

$P \quad = \quad (a \rightarrow P) \quad \sqcap \quad (b \rightarrow b \rightarrow STOP)$

refusals(P/<>) = refusals(P)

refusals(P/<b>) = $\varnothing$, {a}

refusals(P/<b,b>) = all substes of αP

"Failures"

Note that due to non-determinism, there may be several possible states where P may end up after doing s.

- Define :

$$\textbf{\textit{failures}}(P) \;=\; \{\, (s,X) \;\mid\; s \in traces(P) \;,\; X \in refusals(P/s) \,\}$$

(s,X) is a '*failure*' of P means that P can perform s, afterwhich it may deadlock when offered alternatives in X.

- E.g. $(s,\alpha P) \in$ failures(P/s) means after s P may stop.

- If for all X :

$(s,X) \in \text{failures}(P/s) \quad \Rightarrow \quad a \notin X$

this implies that after s P <u>cannot</u> refuse a (implying progress!) .

Example

- Consider this P with $\alpha P = \{a,b\}$:

$$P = (a \rightarrow STOP) \sqcap (b \rightarrow STOP)$$

- P's failures :
  - $(\varepsilon , \{a\})$, $(\varepsilon , \{b\})$, $(\varepsilon , \varnothing)$
  - (a , {a,b}) ... // and other (a,X) where X is a subset of {a,b}
  - (b , {a,b}) ... // and other (b,X) where X is a subset of {a,b}
- Notice the "closure" like property in X and s.

Failures Refinement

- We can use failures as our semantics, and define refinement as follows. Let P and Q to have the same alphabet.

$$P \leq Q \quad = \quad \textbf{\textit{failures}}(P) \supseteq \textbf{\textit{failures}}(Q)$$

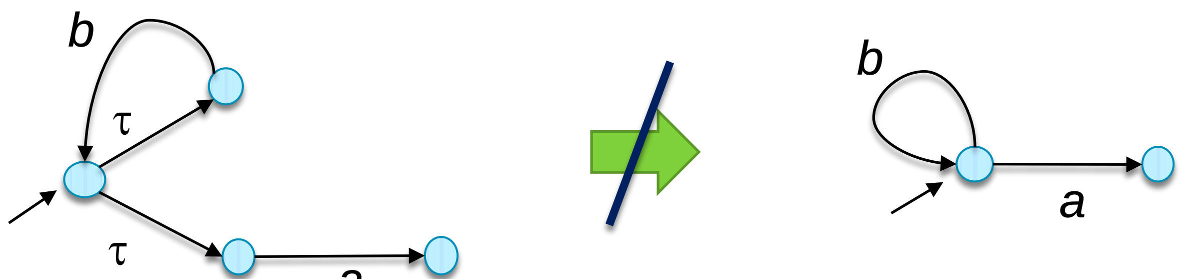
- Also a preorder!
- And it implies trace-refinement, since:

$$\textbf{traces}(P) = \{ s \mid (s,\varnothing) \in \textbf{failures}(P) \}$$

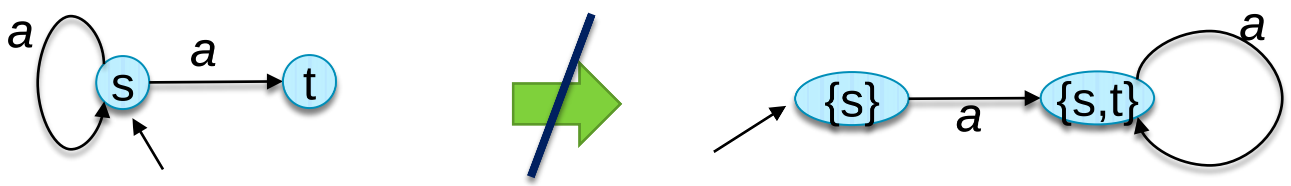
So, it follows that $P \leq Q$ implies traces(P) $\supseteq$ traces(Q).

Back to automata again

- As before we want to use automata to check refinement.
- However now we can't just remove non-determinism, because it does matter in the failures semantic:



Notice that the transformation, although it preserves traces, it does not preserve refusals.



Back to automata

- Still, deterministic automata are attractive because we have seen how we can check trace inclusion.
- Furthermore, in a deterministic automaton, the end-state *u* after producing a trace *s* is *unique*.
- Now remember that a ‘failure’ is a pair of (trace,refusal). Since a trace is identified uniquely by its end-state. this suggests a strategy to label the states with its refusals.
- Then we can adapt our trace-based refinement checking algorithm to also check failures.

Example

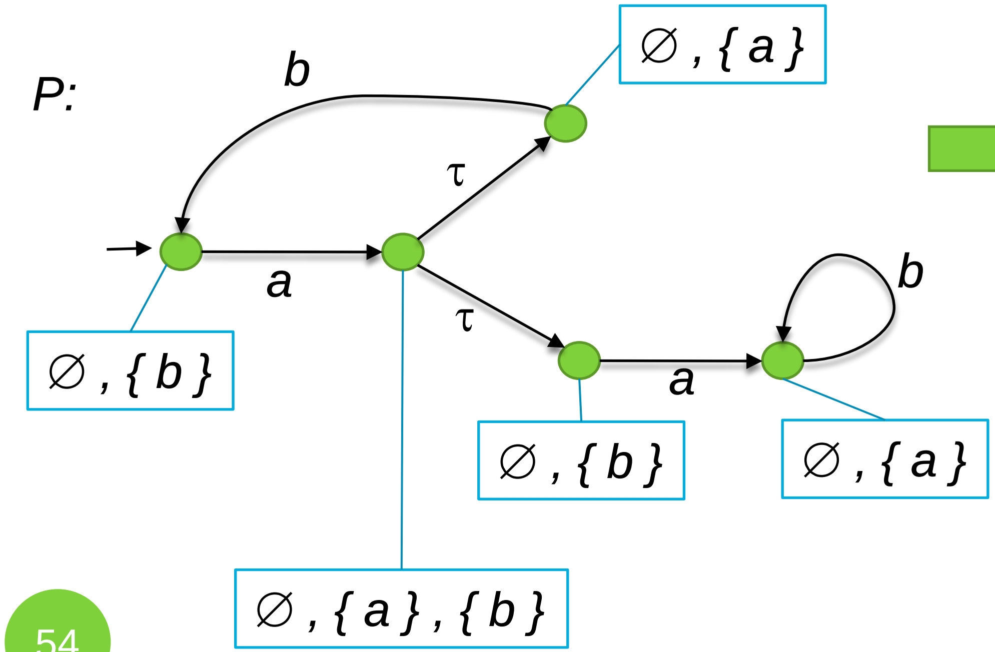
$P = a \rightarrow ((b \rightarrow P) \sqcap (a \rightarrow B))$

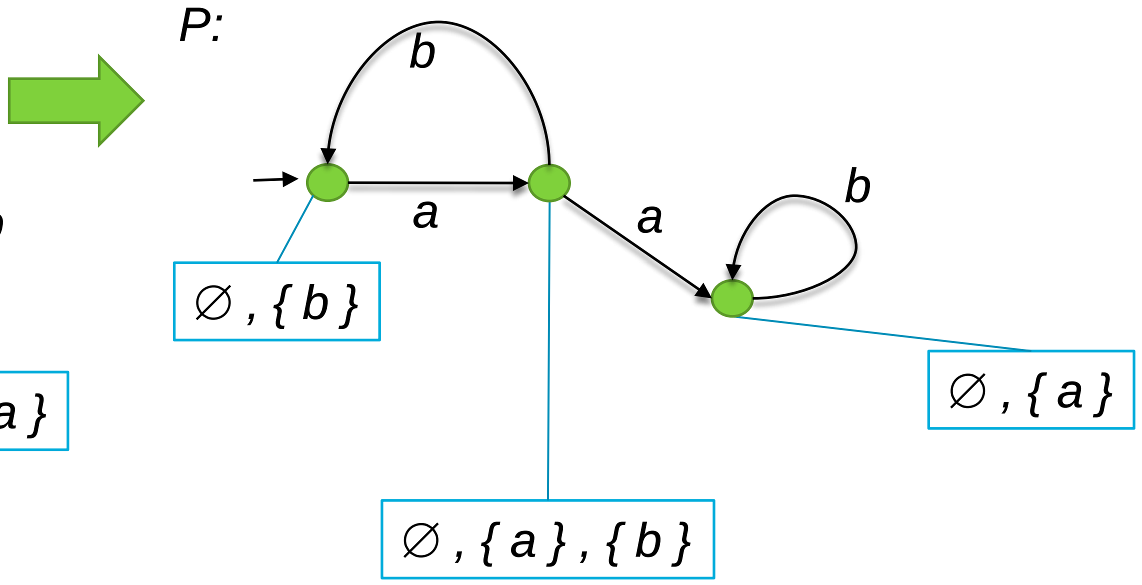
$B = b \rightarrow B$

$Q = a \rightarrow b \rightarrow (Q \sqcap STOP)$

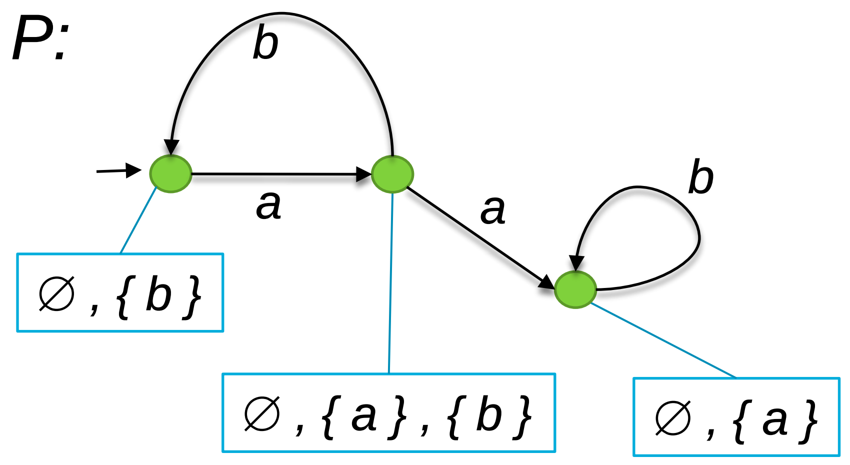
Assuming {a,b} as alphabet.

So, is $P \leq Q$?



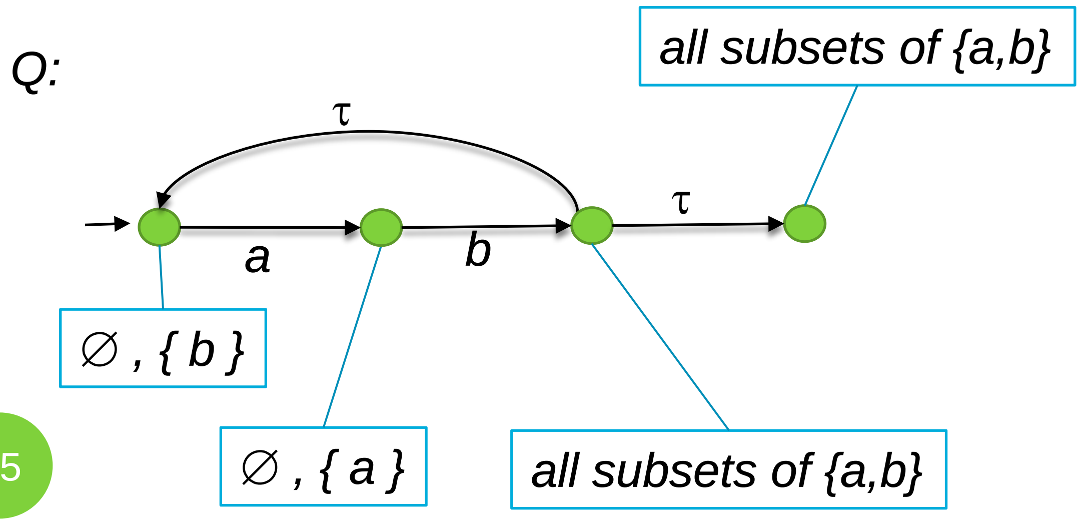


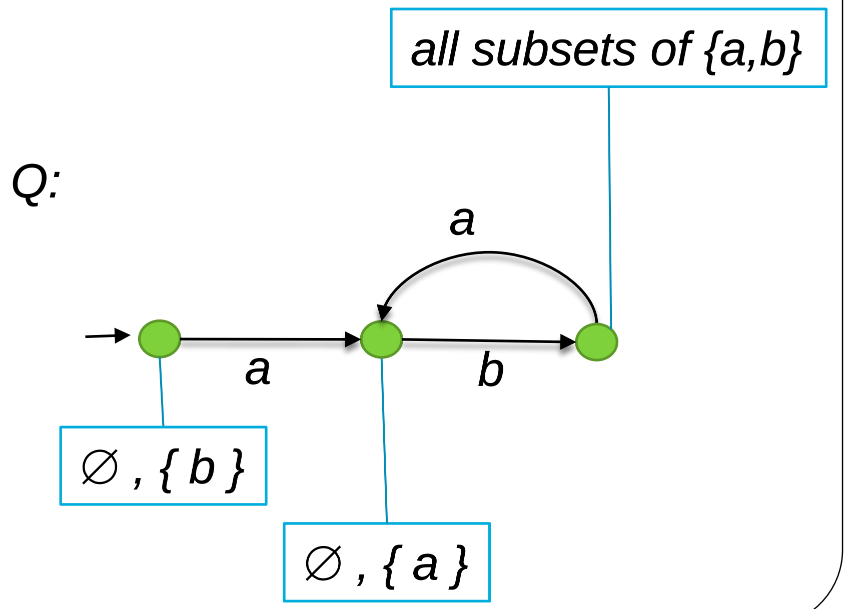
Example



$P \leq Q$?

$Q = a \rightarrow b \rightarrow (Q \sqcap STOP)$

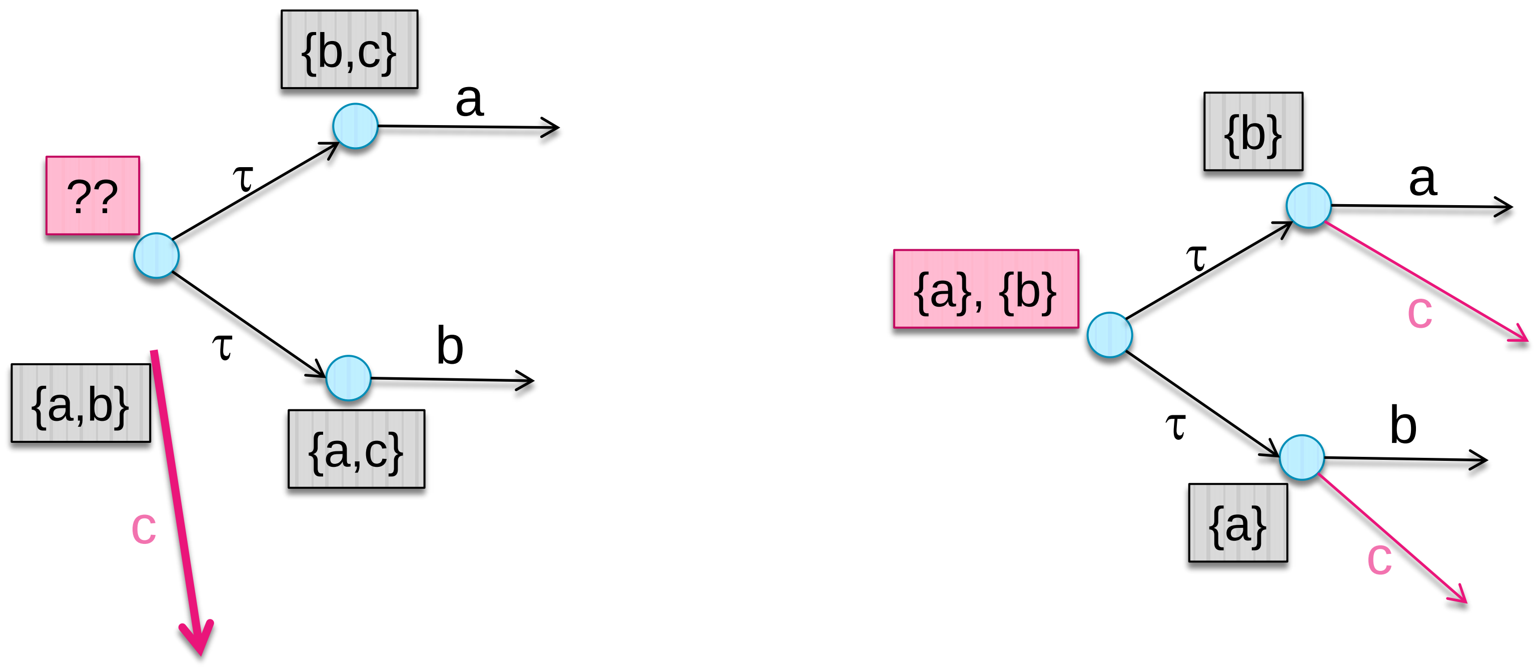




But...

- The procedure doesn't work well with e.g. :

$((a \rightarrow STOP)\ |\overline{\ \ }|\ (b \rightarrow STOP))\quad []\quad (c \rightarrow STOP)$



Normalizing CSP processes

$$\boxed{P \;\square\; (Q \sqcap R) \quad = \quad (P \;\square\; Q) \sqcap (P \;\square\; R)}$$

$$\boxed{P \sqcap (Q \;\square\; R) \quad = \quad (P \sqcap Q) \;\square\; (P \sqcap R)}$$

- Normalize your CSP description so that each process has this form:

$$\begin{aligned} P = {} & (a \rightarrow Q_1) \;[]\; (b \rightarrow Q_2) \;[]\; \ldots && \text{// a,b, ... distinct} \\ & \sqcap \\ & (e \rightarrow R_1) \;[]\; (f \rightarrow R_2) \;[]\; \ldots && \text{// e,f, ... distinct} \\ & \ldots \end{aligned}$$

- When building the automaton representing such a process, each state either:
  - has outgoing arrows which are all tau-steps
  - has outgoing arrows which are all non-tau.

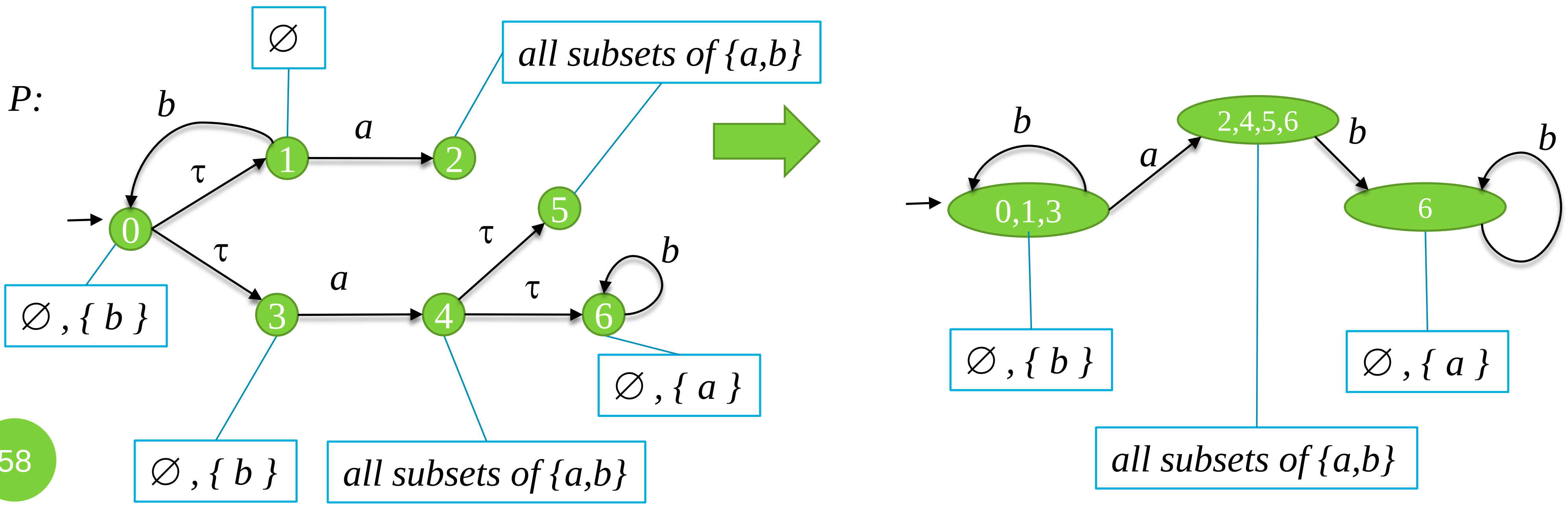
Example

$P = (a \to STOP) [] ((b \to P) \sqcap (a \to B))$

$B = b \to B$

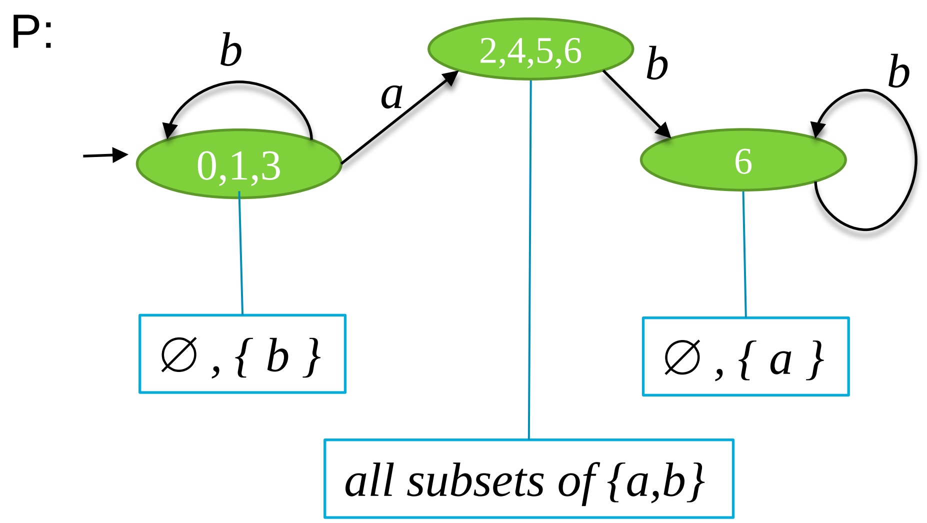
After normalizing:

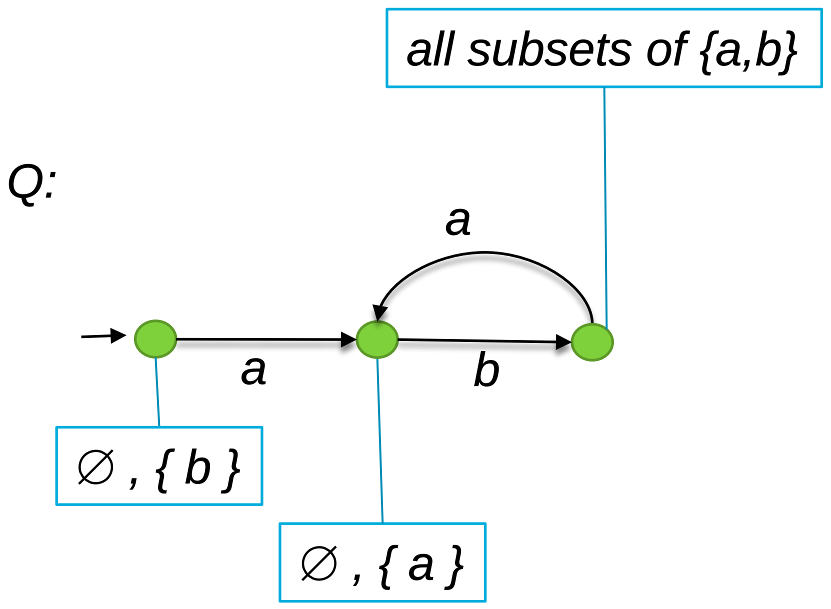
$P = ((a \to STOP) [] (b \to P))$
$\sqcap$
$(a \to (STOP \sqcap B))$



Example

So, is P≤Q, where $Q = a \rightarrow b \rightarrow (Q \sqcap STOP)$?





Some notes

- For the sake of simplicity, the algorithm explained here deviates from the original in Roscoe:
  - It’s not necessary to normalize the ‘implementation’ side.
  - Roscoe still normalize the specification side.
  - We also ignore “divergence”.

- In the worst case, normalization may produce a process whose size is exponential wrt the original.
  - In practice it’s usually not that bad.
  - Specification side is usually much simpler than the implimentation side.