Computer aided verification

Lecture 4:

Algorithmic aspects of LTL model-checking, partial-order reductions

Sławomir Lasota
University of Warsaw

Algorithm

$M \models \phi$?

(i) $M \mapsto \mathcal{A}_M$

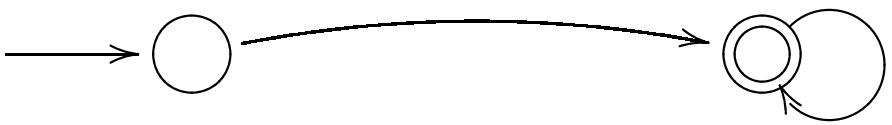
(ii) $\neg\phi \mapsto \mathcal{A}_{\neg\phi}$ (never claim) (not $\phi \mapsto \mathcal{A}_\phi \mapsto \bar{\mathcal{A}}_\phi$)

(iii) $L_\omega(\mathcal{A}_M) \cap L_\omega(\mathcal{A}_{\neg\phi}) = \emptyset$? (not $L_\omega(\mathcal{A}_M) \subseteq L_\omega(\mathcal{A}_\phi)$)

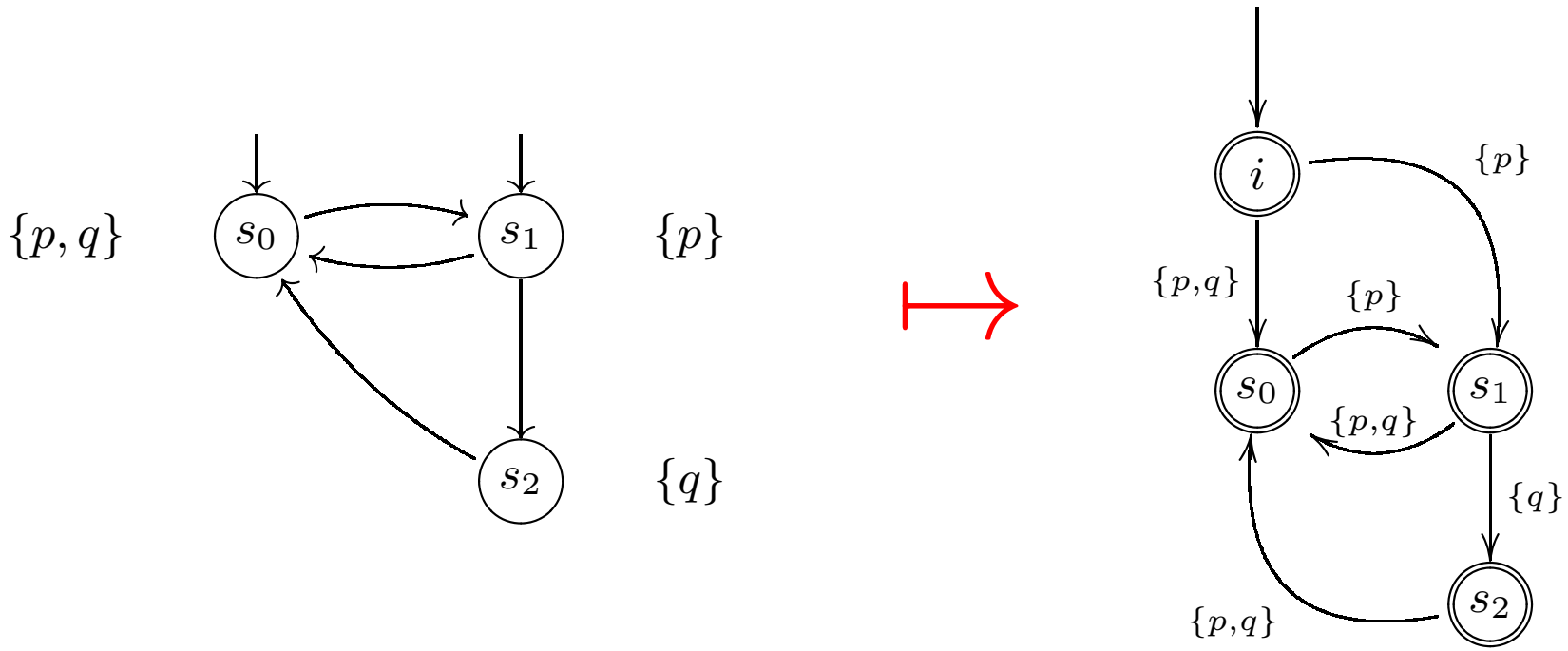
$L_\omega(\mathcal{A}_M \times \mathcal{A}_{\neg\phi}) = \emptyset$?

yes $\rightarrow$ $M \models \phi$

no $\rightarrow$ $\neg(M \models \phi)$, counterexample = a path in M



(i) $M \mapsto \mathcal{A}_M$



Algorithm

$M \models \phi$?

(i) $M \mapsto \mathcal{A}_M$

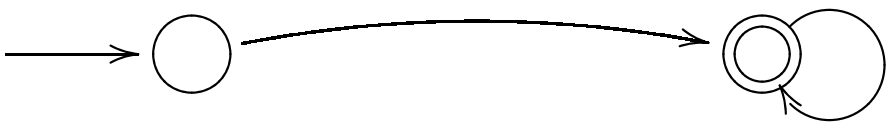
(ii) $\neg\phi \mapsto \mathcal{A}_{\neg\phi}$ (never claim) (not $\phi \mapsto \mathcal{A}_\phi \mapsto \bar{\mathcal{A}}_\phi$)

(iii) $L_\omega(\mathcal{A}_M) \cap L_\omega(\mathcal{A}_{\neg\phi}) = \emptyset$? (not $L_\omega(\mathcal{A}_M) \subseteq L_\omega(\mathcal{A}_\phi)$)

$L_\omega(\mathcal{A}_M \times \mathcal{A}_{\neg\phi}) = \emptyset$?

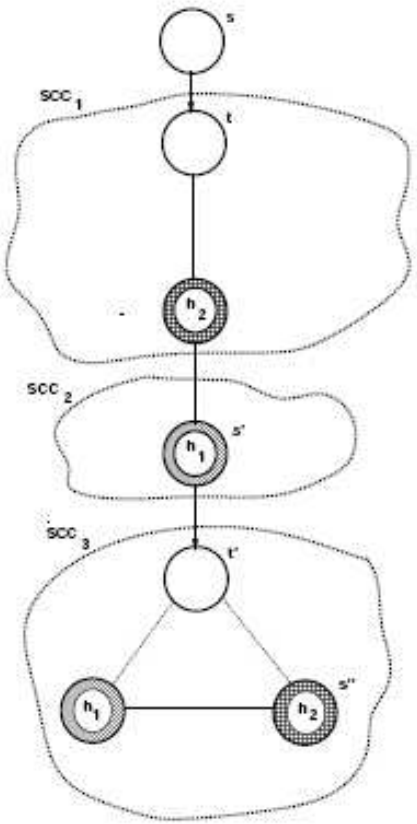
yes $\rightarrow$ $M \models \phi$

no $\rightarrow$ $\neg(M \models \phi)$, counterexample = a path in M



(iii) $L_\omega(\mathcal{A}) \neq \emptyset$?

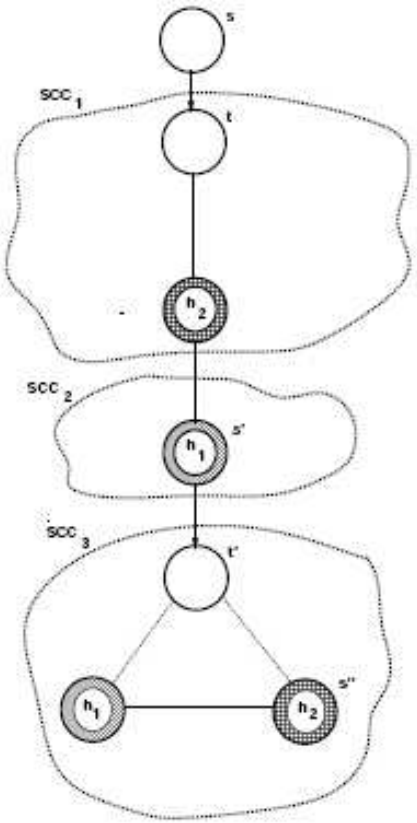
Traversal of a graph:



[Clarke, Grumberg, Peled 2000]

(iii) $L_\omega(\mathcal{A}) \neq \emptyset$?

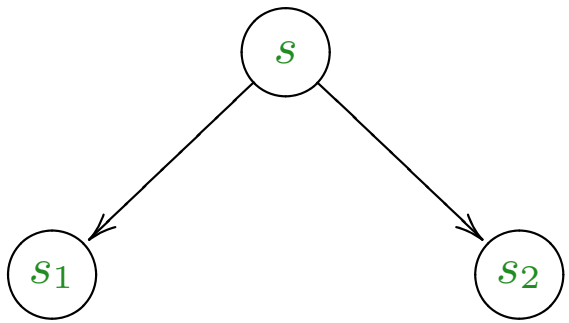
Traversal of a graph:



[Clarke, Grumberg, Peled 2000]

On the fly verification

for each successsor s_i of s do . . .

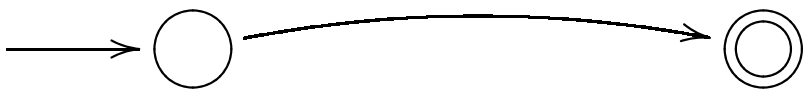


Reachability: F bad state

For reachability it is enough to use DFS or BFS

```
proc dfs(s)
    if error(s) then report error fi
    add s to Statespace
    for each successor t of s do
       if t not in Statespace then dfs(t) fi
    od
end
```

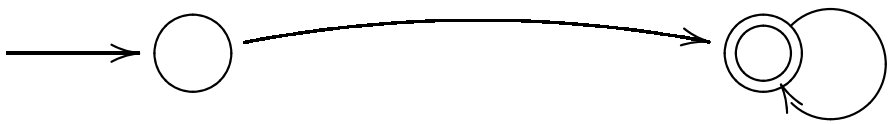
[Holzmann,Peled,Yannakakis 1996]



Nested DFS

```
proc dfs(s)
    if error(s) then report error fi
    add {s,0} to Statespace
    for each successor t of s do
      if {t,0} not in Statespace then dfs(t) fi
    od
    if accepting(s) then seed:=s; ndfs(s) fi
end
proc ndfs(s) /* the nested search */
    add {s,1} to Statespace
    for each successor t of s do
      if {t,1} not in Statespace then ndfs(t) fi
      else if t==seed then report cycle fi
    od
end
```

[Holzmann,Peled,Yannakakis 1996]



Proof of correctness

Assume an acceping state `p` with a cycle not detected by `ndfs(p)`. Let `p` – the first such state.

Proof of correctness

Assume an acceping state `p` with a cycle not detected by `ndfs(p)`. Let `p` – the first such state.

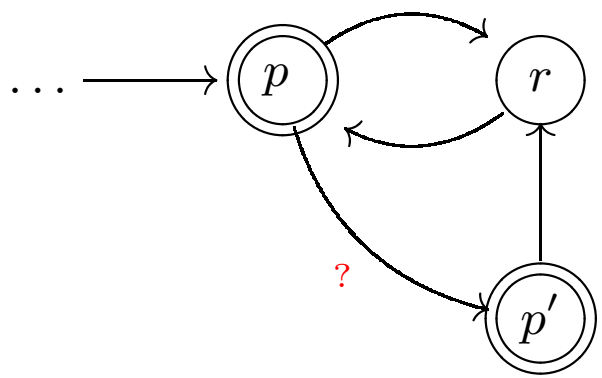
Let `r` – the first state inspected by `ndfs(p)` that is on a `p`-cycle and for which `{r,1}` in `Statespace`.

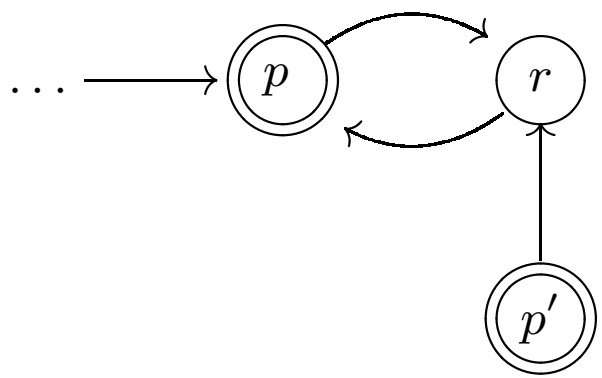
Proof of correctness

Assume an acceping state `p` with a cycle not detected by `ndfs(p)`. Let `p` – the first such state.

Let `r` – the first state inspected by `ndfs(p)` that is on a `p`-cycle and for which `{r,1}` in `Statespace`.

Let `p'` – the accepting state such that `r` visited by `ndfs(p')`.





Partial-order reductions?

(1) On the fly verification: **for each successsor s_i of s do ...**

Partial-order reductions?

(1) On the fly verification: **for each successsor s_i of s do ...**

(2) Partial-order reductions: **for each selected successsor s_i of s do ...**

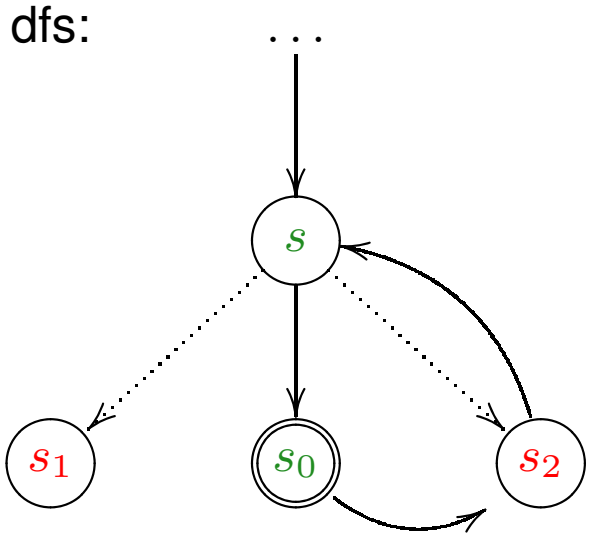
selected depends on the DFS stack !

Partial-order reductions?

(1) On the fly verification: **for each successsor s_i of s do ...**

(2) Partial-order reductions: **for each selected successsor s_i of s do ...**

selected depends on the DFS stack !



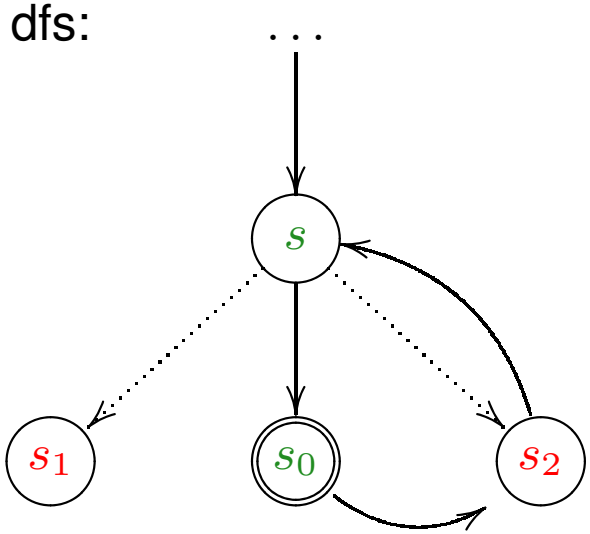
Partial-order reductions?

(1) On the fly verification: **for each successsor s_i of s do ...**

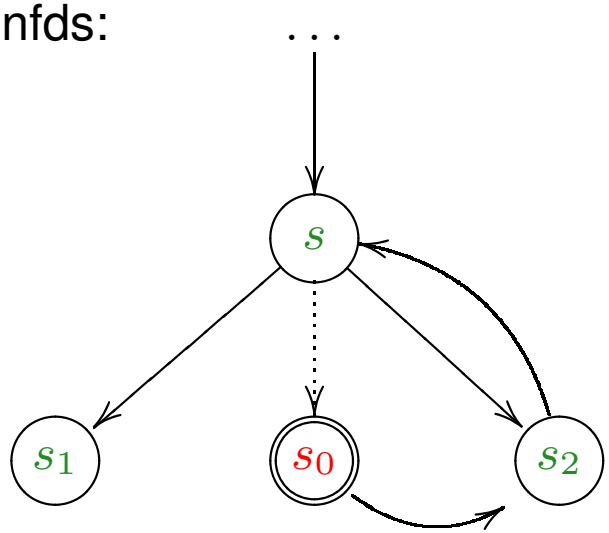
(2) Partial-order reductions: **for each selected successsor s_i of s do ...**

selected depends on the DFS stack !

dfs:



nfds:



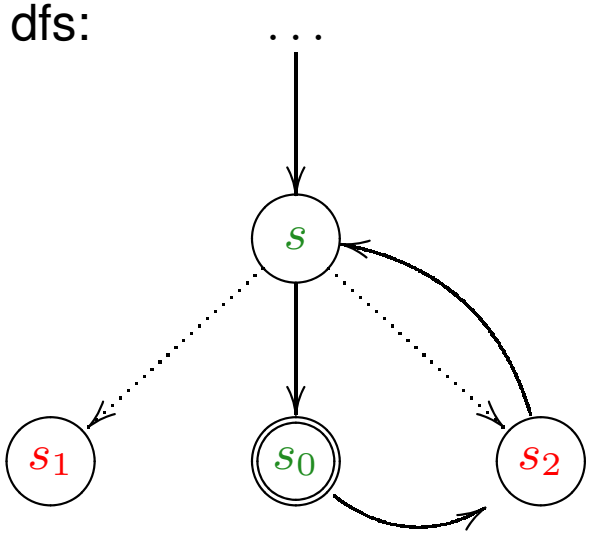
Partial-order reductions?

(1) On the fly verification: **for each successsor s_i of s do ...**

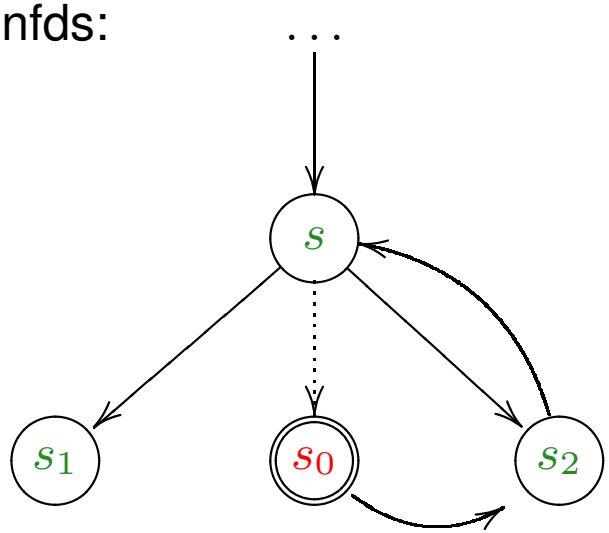
(2) Partial-order reductions: **for each selected successsor s_i of s do ...**

selected depends on the DFS stack !

dfs:



nfds:



Solution: Report a cycle when a stack is hit in ndfs.

Nested DFS compatible with p.-o.red.

```
proc dfs(s)
    if error(s) then report error fi
    add {s,0} to Statespace
    add s to Stack
    for each (selected) successor t of s do
      if {t,0} not in Statespace then dfs(t) fi
    od
    if accepting(s) then ndfs(s) fi
    delete s from Stack
end
proc ndfs(s) /* the nested search */
    add {s,1} to Statespace
    for each (selected) successor t of s do
      if {t,1} not in Statespace then ndfs(t) fi
      else if t in Stack then report cycle fi
    od
end
```

[Holzmann,Peled,Yannakakis 1996]

Nested DFS compatible with p.-o.red.

```
proc dfs(s)
    if error(s) then report error fi
    add {s,0} to Statespace
    add s to Stack
    for each (selected) successor t of s do
      if {t,0} not in Statespace then dfs(t) fi
    od
    if accepting(s) then ndfs(s) fi
    delete s from Stack
end
proc ndfs(s) /* the nested search */
    add {s,1} to Statespace
    for each (selected) successor t of s do
      if {t,1} not in Statespace then ndfs(t) fi
      else if t in Stack then report cycle fi
    od
end
```

[Holzmann,Peled,Yannakakis 1996]

Question: Is nested DFS correct now?

np-cycles: FG ¬ progress

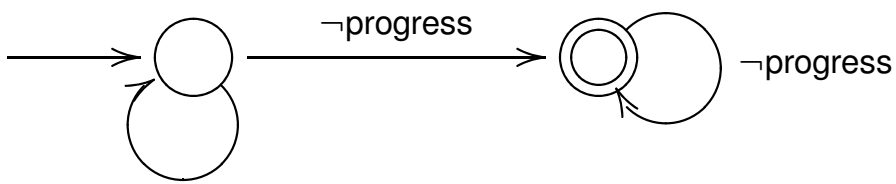
```
proc dfs(s)
    if error(s) then report error fi
    add {s,0} to Statespace
    for each successor t of s do
      if {t,0} not in Statespace then dfs(t) fi
    od
    ndfs(s) /* different */
end
proc ndfs(s) /* the nested search */
    if s is Progress State then return fi /* new */
    add {s,1} to Statespace
    add s to Stack /* new */
    for each successor t of s do
      if {t,1} not in Statespace then ndfs(t) fi
      else if t is in Stack then report cycle fi /* different */
    od
    delete s from Stack /* new */
end
```

[Holzmann,Peled,Yannakakis 1996]

np-cycles: never claim

```
never { /* non-progress: ◇□¬progress */
    do
    :: skip
    :: !progress -> break
    od;
accept: do
    :: !progress
    od
}
```

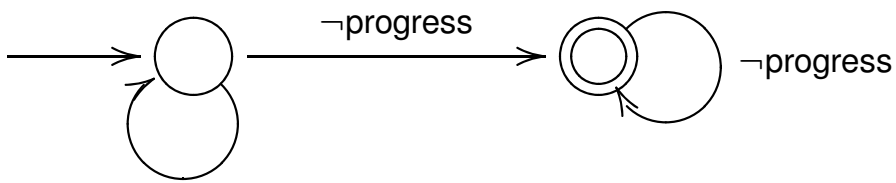
[Holzmann,Peled,Yannakakis 1996]



np-cycles: never claim

```
never { /* non-progress: ◇□¬progress */
    do
    :: skip
    :: !progress -> break
    od;
accept: do
    :: !progress
    od
}
```

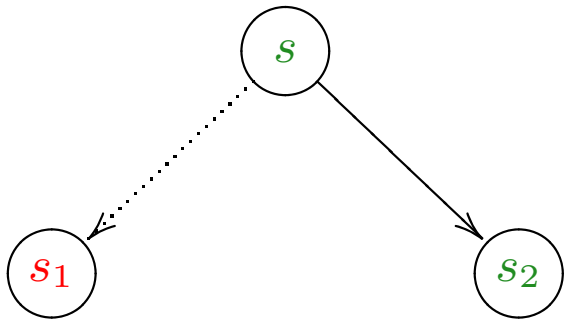
[Holzmann,Peled,Yannakakis 1996]



Question: What overhead is introduced, when the never claim is used?

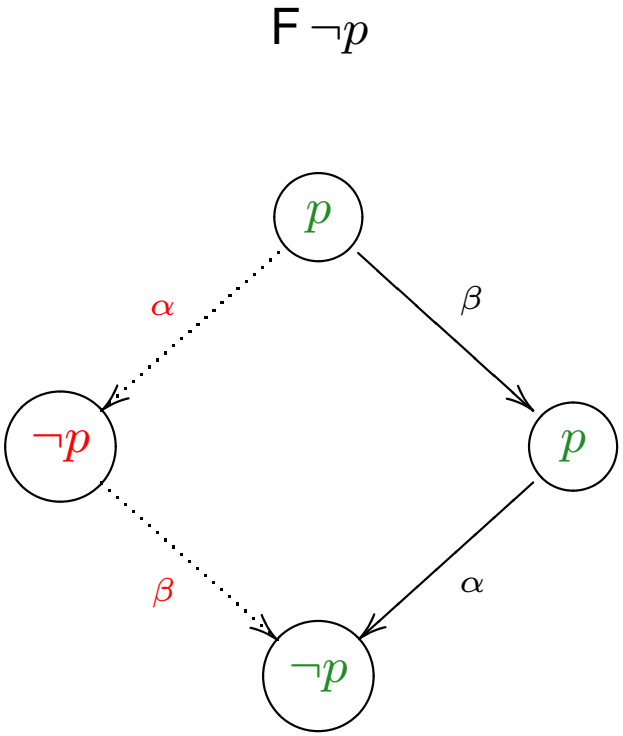
Partial-order reductions

for each selected successsor s_i of s do . . .

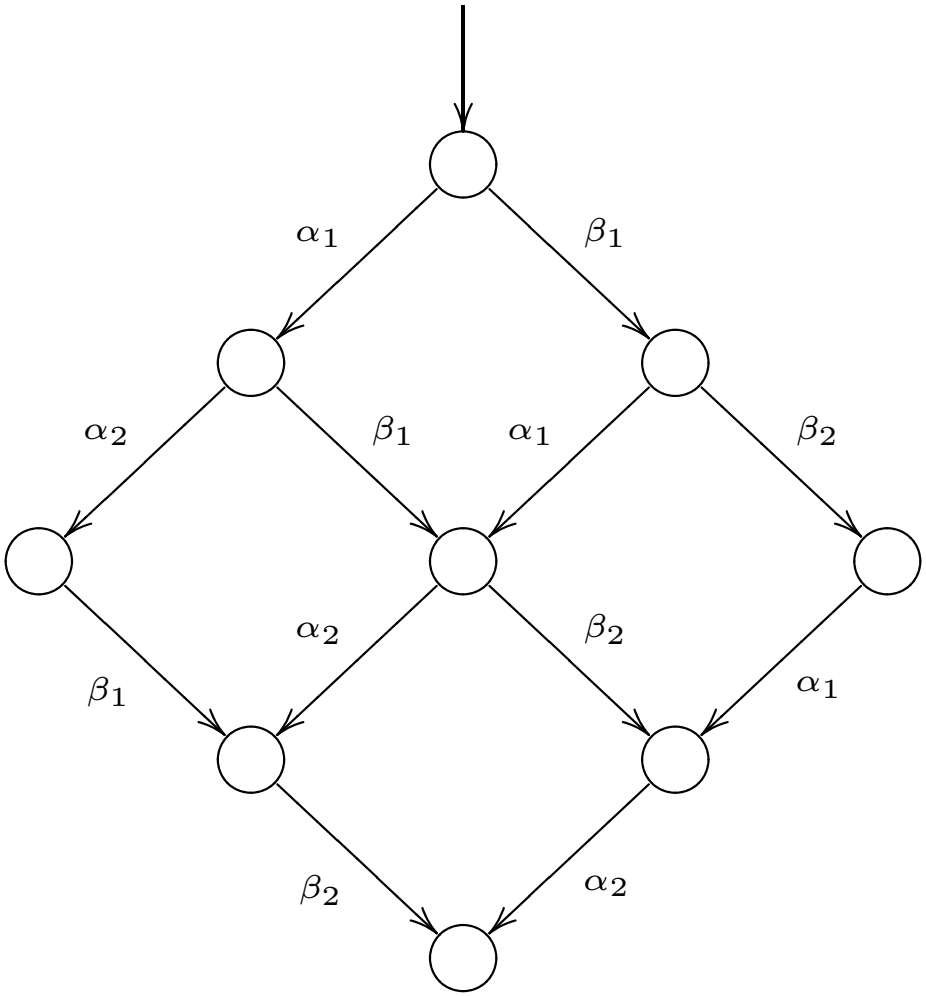


Motivation

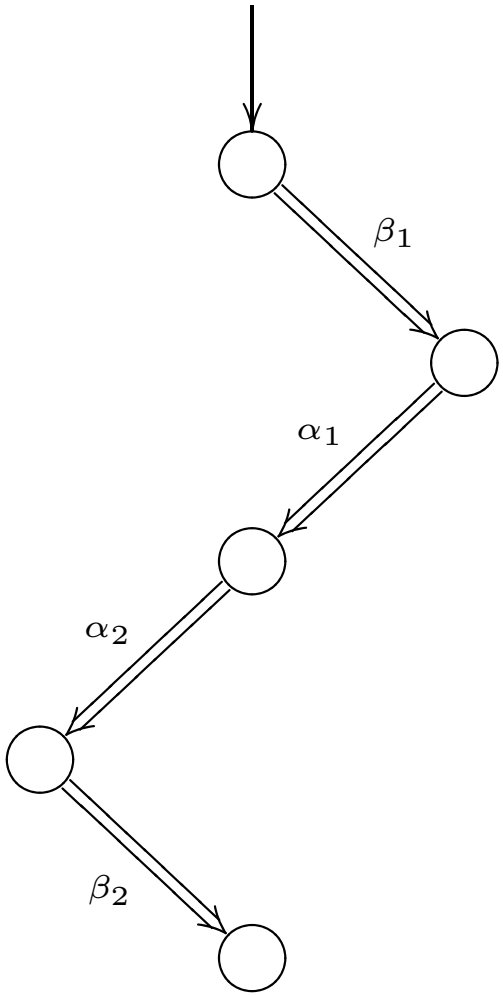
Idea: exploit indpendence (concurrency) of transitions



Motivation


α_1
β_1
α_2
β_1
α_1
β_2
α_2
β_2
β_1
α_1
β_2
α_2

Motivation



Model

Def.: $M = \langle S, S_{\text{init}}, T, L \rangle$ T – operations (transitions)

for $\alpha \in T$: $\text{en}_\alpha \subseteq S, \quad \alpha : \text{en}_\alpha \to S$ (determinism)

path: $\Pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots$ $s_0 = s_{\text{init}}$

$\alpha_i(s_i) = s_{i+1}$

$\text{en}_s := \{\alpha \mid s \in \text{en}_\alpha\}$ $(\alpha \in \text{en}_s \iff s \in \text{en}_\alpha)$

Idea: $\text{ample}_s \subseteq \text{en}_s$ instead of en_s in nested DFS ?

Cost-effectivity

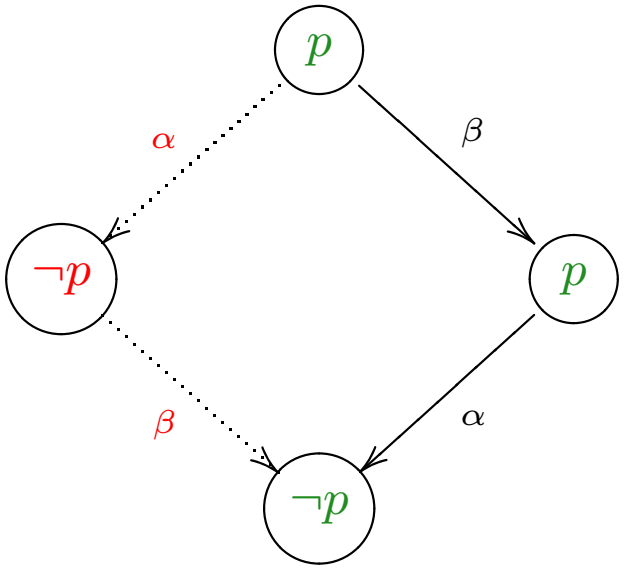
Idea: $\mathrm{ample}_s \subseteq \mathrm{en}_s$ instead of en_s in nested DFS ?

This makes sense, when:

- the result of verification is the same (correctness)
- significantly less states visited
- time overhead reasonable (effectivity)

Correctness?

When may we ignore α ?



Problem 1: Property may depend on state $\neg p$.

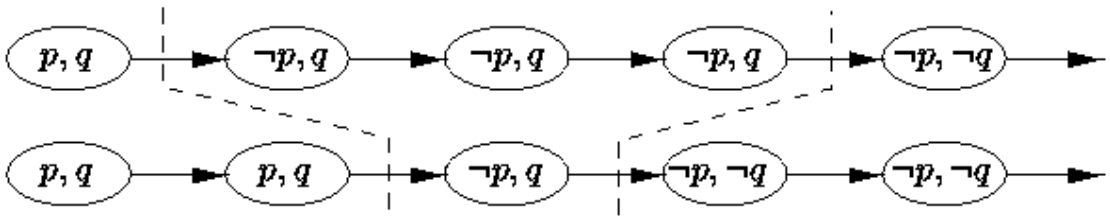
Problem 2: $\neg p$–successors unreachable otherwise.

Stuttering

Def.: $\Pi = s_0 \to s_1 \to s_2 \to \ldots$ and $\Pi' = s'_0 \to s'_1 \to s'_2 \to \ldots$ are stuttering equivalent, $\Pi \equiv \Pi'$, if sequences

$$L(s_0), L(s_1), L(s_2), \ldots \qquad L(s'_0), L(s'_1), L(s'_2), \ldots$$

become identical after grouping is done:



Def.: $M \equiv M'$ if and only if

– $\forall \Pi$ in M $\exists \Pi'$ in M' $\Pi \equiv \Pi'$

– $\forall \Pi'$ in M' $\exists \Pi$ in M $\Pi \equiv \Pi'$

LTL$_{-X}$ = LTL without X

Thm: If $\phi \in$ LTL$_{-X}$ and $\Pi \equiv \Pi'$, then $\Pi \vDash \phi \iff \Pi' \vDash \phi$

Thm: If $\phi \in$ LTL$_{-X}$ and $M \equiv M'$, then $M \vDash \phi \iff M' \vDash \phi$

Thm: LTL$_{-X}$ $=$ FO$_{\equiv}(\leq)$

Correctness

$$M \xmapsto{\text{partial-order reduction}} M'$$

$$M \equiv M'$$

Sufficient condition for correctness

(C0) $\text{ample}_s = \emptyset \iff \text{en}_s = \emptyset$

(C1) ...

(C2) ...

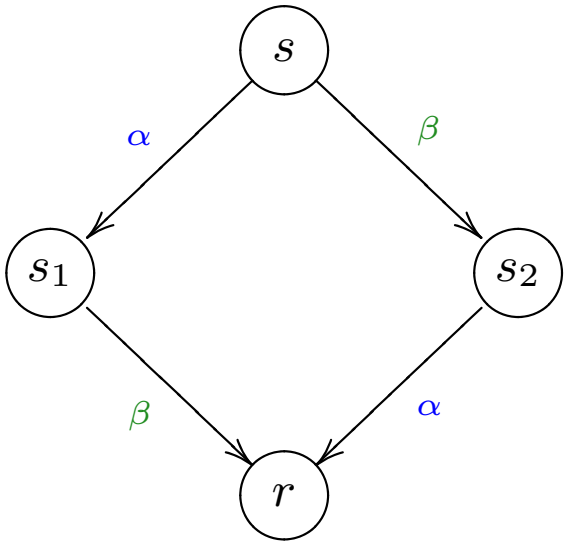
(C3) ...

Invisibility

Def.: α is invisible if $L(s) = L(\alpha(s)), \forall\ s \in \mathrm{en}_\alpha$.

Example: If α invisible, then

$$ss_1r \equiv ss_2r$$



Sufficient condition for correctness

(C0) $\text{ample}_s = \emptyset \iff \text{en}_s = \emptyset$

(C1) if $\text{ample}_s \neq \text{en}_s$ then every $\alpha \in \text{ample}_s$ is invisible

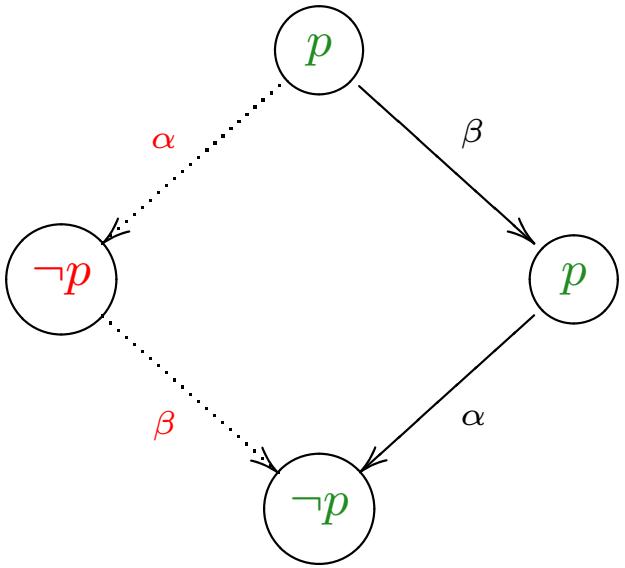
(C2) ...

(C3) ...

Idea: Instead of doing sth now, do it in future!

Correctness?

Problem 1: Property may depend on state $\neg p$.



Solved due to **(C1)** !

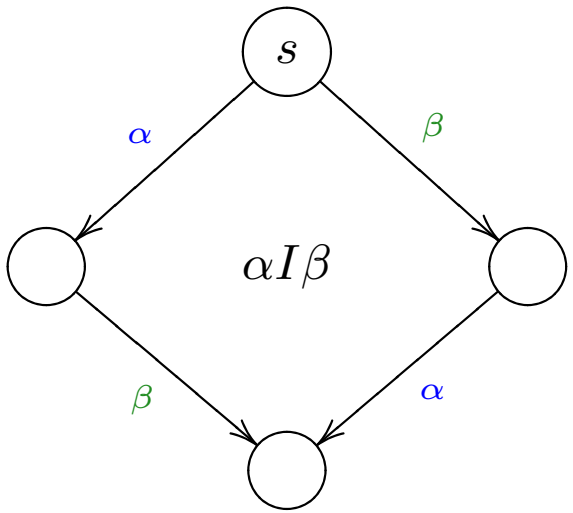
(C1) if $\mathrm{ample}_s \neq \mathrm{en}_s$, then every $\alpha \in \mathrm{ample}_s$ is invisible

Independence

Def.: Relation of independence $I \subseteq T \times T$:

- irreflexive and antisymmetric
- if $\alpha I \beta$, $\alpha \in \mathrm{en}_s$, $\beta \in \mathrm{en}_s$, then $(s \in \mathrm{en}_\alpha \cap \mathrm{en}_\beta)$
  - $\beta(s) \in \mathrm{en}_\alpha$, $\alpha(s) \in \mathrm{en}_\beta$
  - $\beta(\alpha(s)) = \alpha(\beta(s))$

$D = T \times T \setminus I$ (dependency)



Independence

Example: Independent may be:

– 2 instructions of different processes operating on local variables

Independence

Example: Independent may be:

- 2 instructions of different processes operating on local variables
- 2 instructions of different processes that increment the same global variable

Independence

Example: Independent may be:

- 2 instructions of different processes operating on local variables
- 2 instructions of different processes that increment the same global variable
- 2 instructions of different processes writing to/reading from different buffers

Independence

Example: Independent may be:

- 2 instructions of different processes operating on local variables
- 2 instructions of different processes that increment the same global variable
- 2 instructions of different processes writing to/reading from different buffers
- 2 instructions of different processes:

  one writing to a buffer

  the other one reading from the same buffer

Independence

Example: Independent may be:

- 2 instructions of different processes operating on local variables
- 2 instructions of different processes that increment the same global variable
- 2 instructions of different processes writing to/reading from different buffers
- 2 instructions of different processes:

  one writing to a buffer

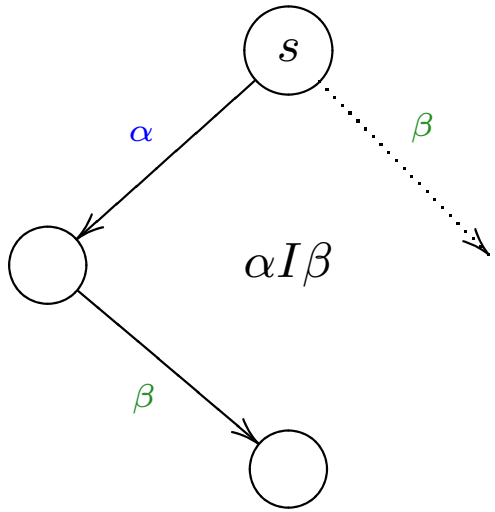
  the other one reading from the same buffer

Question: Can 2 instructions of the same process be independent ?

Independence

Question: Let $\alpha I \beta$. Is it possible that

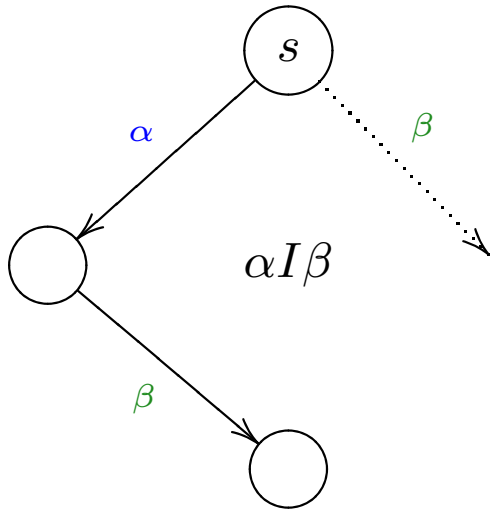
$$s \in \mathrm{en}_\alpha \setminus \mathrm{en}_\beta \qquad \alpha(s) \in \mathrm{en}_\beta \ ?$$



Independence

Question: Let $\alpha I \beta$. Is it possible that

$$s \in \mathrm{en}_\alpha \setminus \mathrm{en}_\beta \qquad \alpha(s) \in \mathrm{en}_\beta \; ?$$



Yes! E.g. asynchronous reading and writing from/to the same buffer by two different processes.

Sufficient condition for correctness

(C0) $\text{ample}_s = \emptyset \iff \text{en}_s = \emptyset$

(C1) if $\text{ample}_s \neq \text{en}_s$ then every $\alpha \in \text{ample}_s$ is invisible

(C2) ? $\big(\text{en}_s \setminus \text{ample}_s\big)\ I\ \text{ample}_s$

(C3) ...

Idea: Instead of doing sth now, do it in future!

(C2)

(C2) a transition dependent on some transition from ample_s

can not be enabled before some transition from ample_s is executed

(C2) a transition dependent on some transition from ample_s

can not be enabled before some transition from ample_s is executed

(C2) for every path Π starting in s:

if $\alpha \in \mathrm{ample}_s,\ \beta \notin \mathrm{ample}_s,\ \alpha D \beta$

then β does not appear in Π

before some transition from ample_s is executed

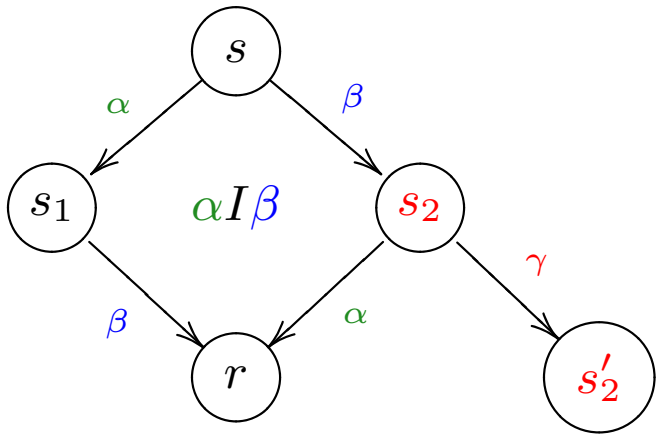
Lemma: (C2) implies $\left(\mathrm{en}_s \setminus \mathrm{ample}_s\right) I \, \mathrm{ample}_s$.

Proof: Let $\beta \in \mathrm{en}_s \setminus \mathrm{ample}_s$, $\alpha \in \mathrm{ample}_s$, $\alpha D \beta$.

$s \xrightarrow{\beta} \beta(s) \rightarrow \ldots$ contradiction with (C2) .

Correctness?

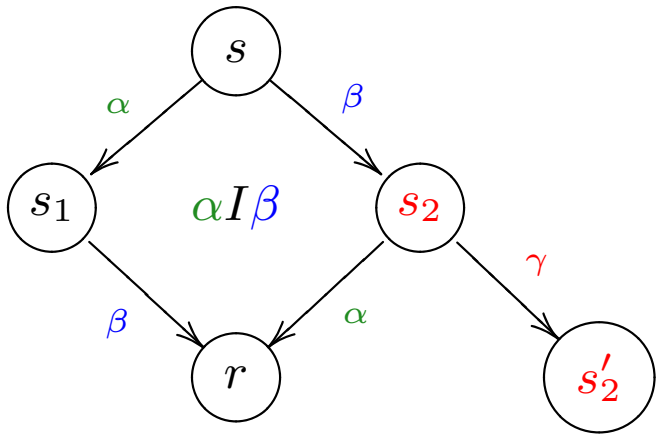
Problem 2: s_2 –successors unreachable otherwise.



e.g., let $\alpha \in \text{ample}_s$, $\beta \notin \text{ample}_s$

Correctness?

Problem 2: s_2 –successors unreachable otherwise.

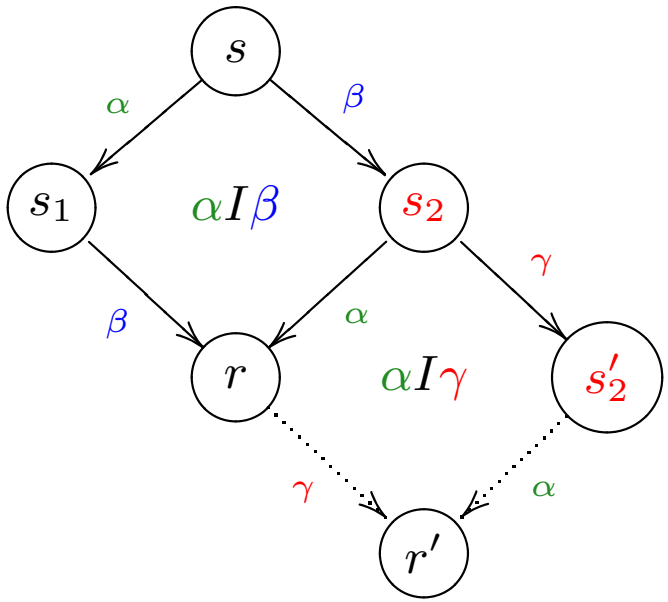


e.g., let $\alpha \in \text{ample}_s, \beta \notin \text{ample}_s$

by **(C2)** applied to $\beta\gamma \ldots$, we deduce $\gamma I \alpha$

Problems?

Problem 2: s_2 –successors unreachable otherwise.



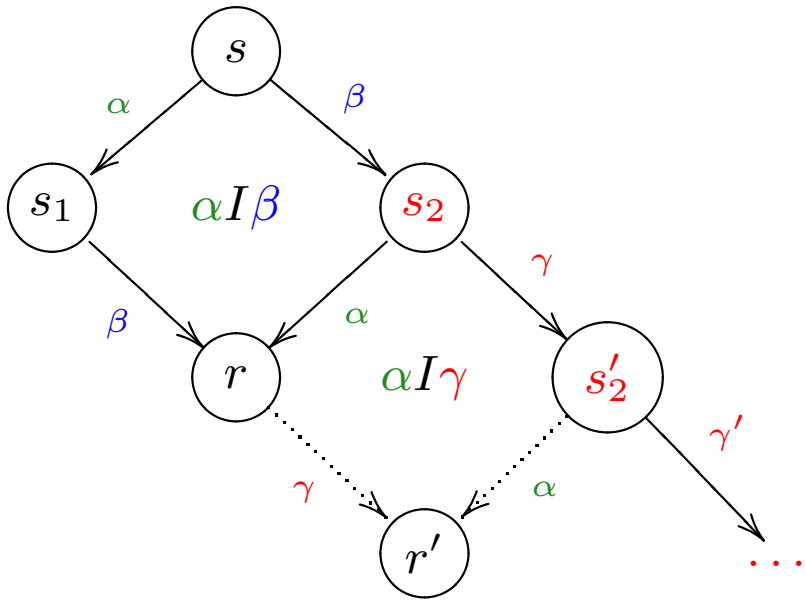
e.g., let $\alpha \in \text{ample}_s$, $\beta \notin \text{ample}_s$

by **(C2)** applied to $\beta\gamma \ldots$, we deduce $\gamma I \alpha$

α invisible, thus $ss_1rr' \equiv ss_2s_2'$

Problems?

Problem 2^∞ : (s_2) –path unreachable otherwise.

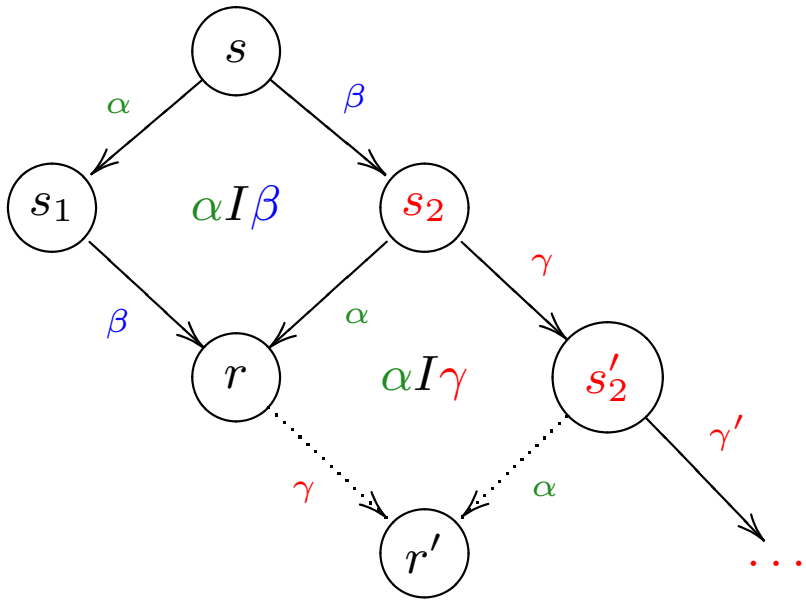


by **(C2)** we deduce $\gamma I \alpha$, $\gamma' I \alpha$, $\ldots$

α invisible, thus $s s_1 r r' \ldots \equiv s s_2 s_2' \ldots$

Problems?

Problem 2^∞ : (s_2) –path unreachable otherwise.



by **(C2)** we deduce $\gamma I \alpha$, $\gamma' I \alpha$, $\ldots$

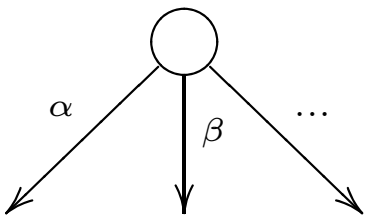
α invisible, thus $s s_1 r r' \ldots \equiv s s_2 s_2' \ldots$

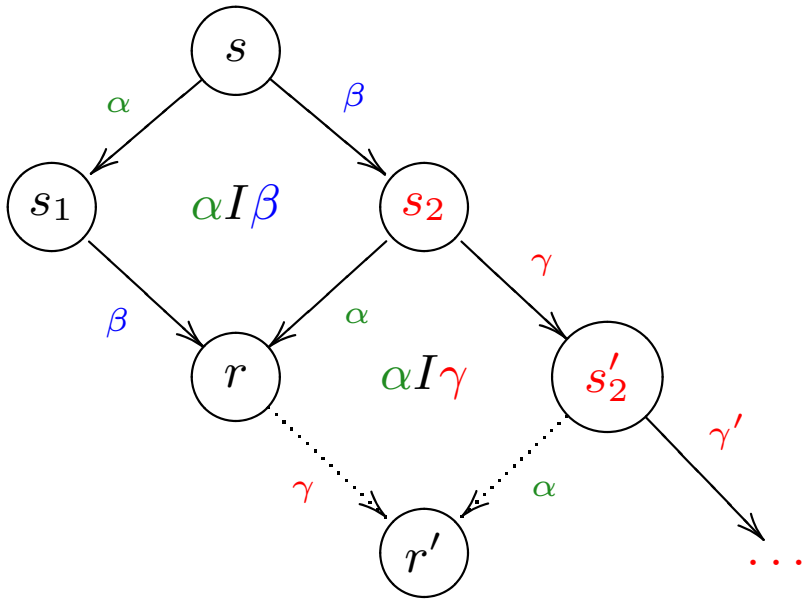
Problem 2^∞ does not appear under weak fairness

Fairness

Def. (weak fairness): if α enabled from some point on then α eventually executed.

Corollary: for every reachable state s, if $\alpha \in \mathrm{en}_s$ then eventually some β will be executed such that $\alpha D \beta$.





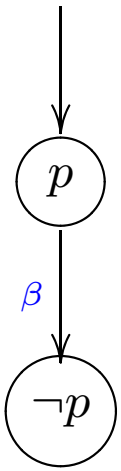
Enough?

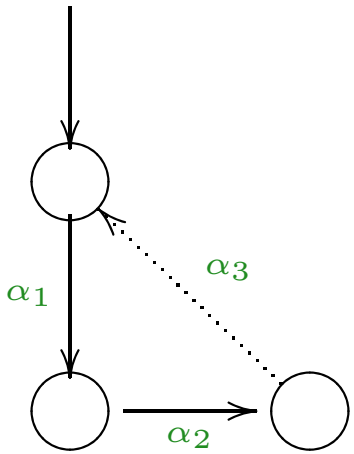
Are **(C0)** – **(C2)** sufficient?

Enough?

Are **(C0)** – **(C2)** sufficient?

No!

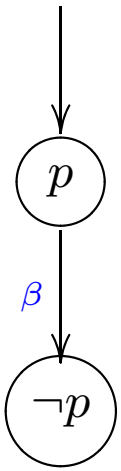


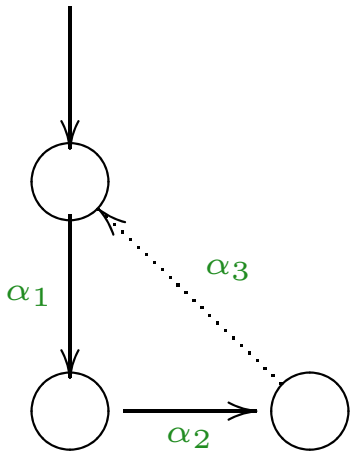


Enough?

Are **(C0)** – **(C2)** sufficient?

No!





(C3) we forbid cycles C such that $\exists \beta \;\; \forall s \in C \;\; \beta \in \mathrm{en}_s \setminus \mathrm{ample}_s$

Sufficient condition for correctness

(C0) $\text{ample}_s = \emptyset \iff \text{en}_s = \emptyset$

(C1) if $\text{ample}_s \neq \text{en}_s$ then every $\alpha \in \text{ample}_s$ is invisible

(C2) for every path Π starting in s:

if $\alpha \in \text{ample}_s,\ \beta \notin \text{ample}_s,\ \alpha D \beta$

then β does not appear in Π

before some transition from ample_s is executed

(C3) we forbid cycles C such that $\exists \beta\ \ \forall s \in C\ \ \beta \in \text{en}_s \setminus \text{ample}_s$

How to implement **(C1)** – **(C3)** ?

Sufficient condition for correctness

(C1) easy

(C2) hard, implemented in an approximate manner

- an over-approximation of D is computed
- condition **(C2)** is monotonic
- static analysis only

(C3) replaced by another condition which is easier but stronger:

(C3') if $\mathrm{ample}_s \neq \mathrm{en}_s$ then $\forall \alpha \in \mathrm{ample}_s \;\; \alpha(s) \notin$ stack

Implementation

Implementation decision:

$\mathrm{ample}_s =$ all transitions of some process i enabled in s

Implementation

Implementation decision:

ample_s = all transitions of some process i enabled in s

whenever

- they are independent from all operations of all other processes
- no operation of any other process may enable

  any other operation of process i

β enabling α (over-approximation)

– if β modifies pc so that α may be executed

– if Promela enabling condition for α depends on global variables, then any β that modifies these variables

– if α is reading from/writing to a buffer then any β that reads from/writes to this buffer

$\alpha D \beta$ (over-approximation)

– α and β refer to the same global variable

  and at least one of them modifies the variable

– α and β belong to the same process; synchronous communication

  is understood as belonging to both processes

– α and β write to/read from the same buffer

However reading from a buffer is independent from writing to the same buffer!

What remains independent?

Example:

Operations independent from all operations of other processes:

- operating on local variables
- reading from a buffer with xr flag set
- writing to a bugger with xs flag set
- test `nempty(q)` if xr flags is set for `q`
- test `nfull(q)` if xs flag is set for `q`