Partial-order reduction

State-space explosion

We have learnt that there is a practical solution to LTL model checking in $\mathcal{O}(|\mathcal{K}| \cdot 2^{|}\phi|)$.

In practice: ϕ small, but $\mathcal{K}$ very large, often exponentially larger than an underlying high-level description. Reasons: data, concurrency, . . .

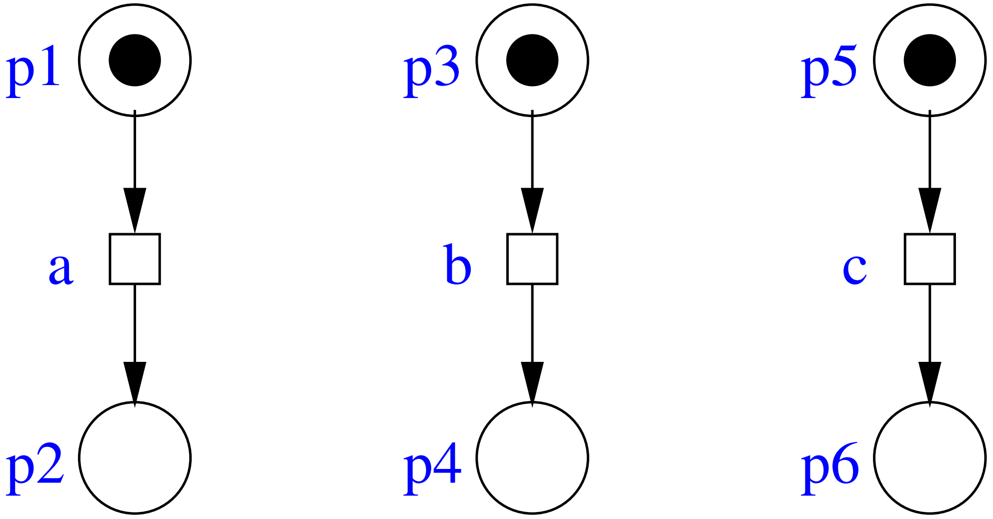
We shall explore a method that tackles state-space explosion for concurrent systems and LTL.

Input: a high-level description of a system (e.g., a description or programming language) and an LTL formula ϕ

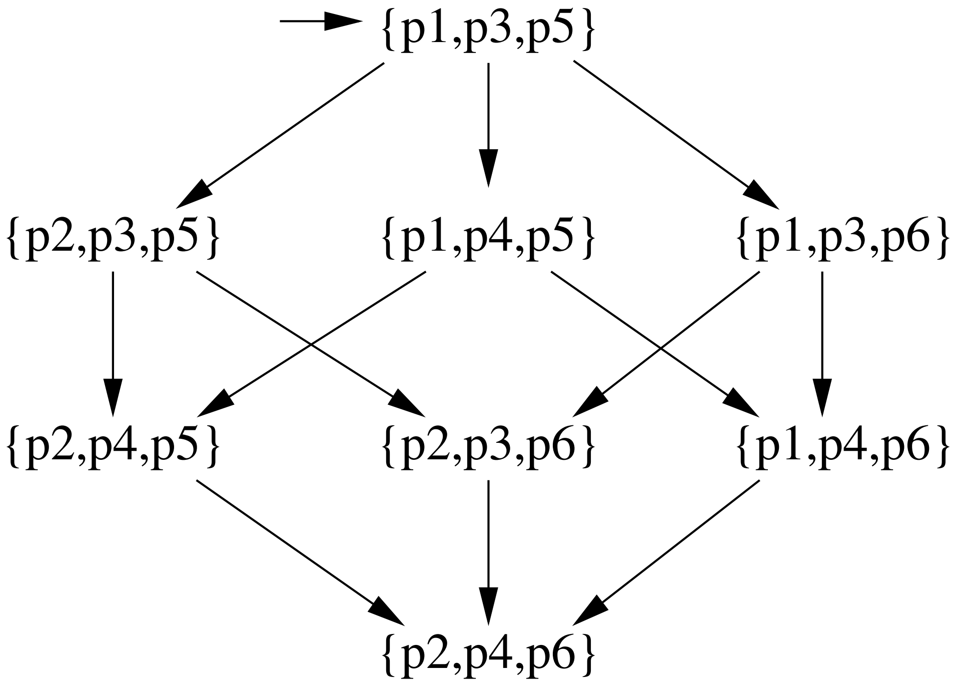
Goal: Determine whether ϕ holds without actually constructing the entire Kripke structure in the first place.

Example

Consider three processes in parallel, each of which can make one step.

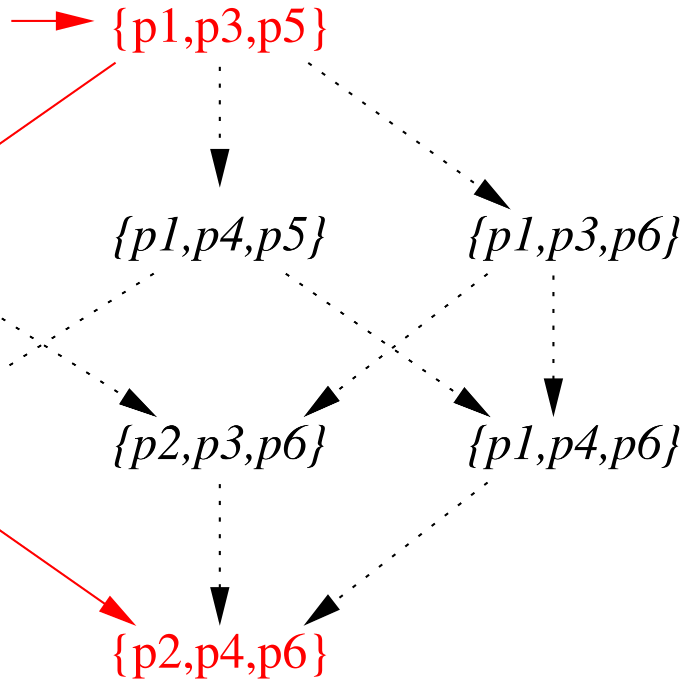


The reachability graph has got $8 = 2^3$ states and $6 = 3!$ possible paths.



For n components we have 2^n states and $n!$ paths.

All paths lead to $\{p_2, p_4, p_6\}$.



Idea: reduce size by considering only one path

Only possible if nothing "interesting" happens in those paths!

On-the-fly Model Checking

Important: It would be pointless to construct $\mathcal{K}$ first and then reduce its size.

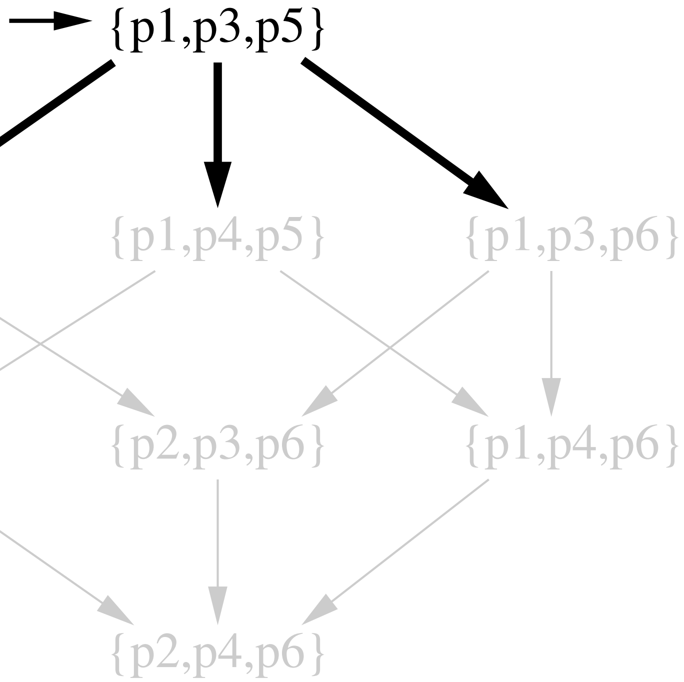
(does not save space, we can analyze $\mathcal{K}$ during construction anyway)

Thus: The reduction must be done “on-the-fly”, i.e. while $\mathcal{K}$ is being constructed (from a compact description such as a Promela model) and during analysis.

$\Rightarrow$ combination with depth-first search

Reduction and DFS

We must decide which paths to explore *at this moment*.



I.e. before having constructed (or “seen”) the rest!
→ only possible with additional information!

Partial-order techniques

Partial-order techniques aim to reduce state-space explosion due to concurrency.

One tries to exploit *independences* between transitions, e.g.

Assignments of variables that do not depend upon each other:

$$\mathtt{x} := \mathtt{z} + 5 \quad \| \quad \mathtt{y} := \mathtt{w} + \mathtt{z}$$

Send and receive on FIFO channels that are neither empty nor full.

Idea: avoid exploring all interleavings of independent transitions

correctness depends on whether the property of interest does not distinguish between different such interleavings

may reduce the state space by an exponential factor

Labelled Kripke structures

We extend our model with actions:

$$\mathcal{K} = (S, A, \rightarrow, r, AP, \nu)$$

> S, r, AP, ν as before, A is a set of actions, and $\rightarrow \subseteq S \times A \times S$.

We assume forthwith that transitions are deterministic, i.e. for each $s \in S$ and $a \in A$ there is at most one $s' \in S$ such that $(s, a, s') \in \rightarrow$.

$en(s) := \{\, a \mid \exists s' \colon (s, a, s') \in \rightarrow \,\}$ are called the enabled actions in s.

Independence

$I \subseteq A \times A$ is called independence relation for $\mathcal{K}$ if:

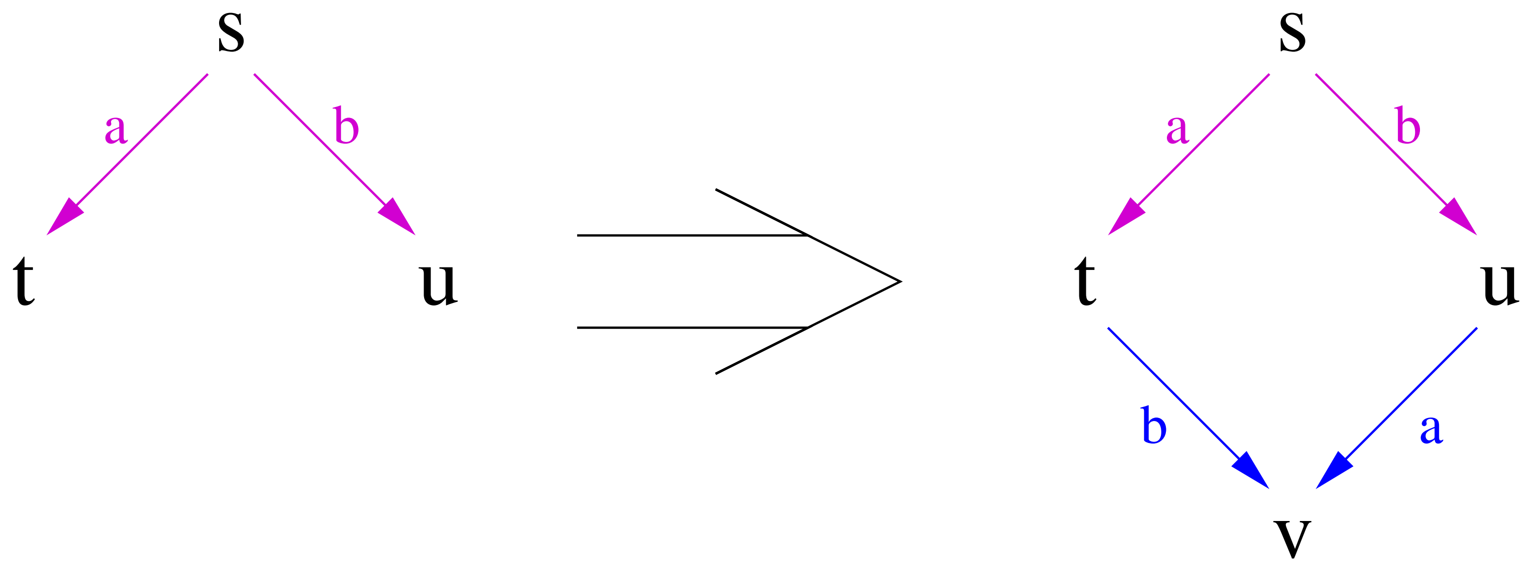
for all $a \in A$ we have $(a, a) \notin I$ (irreflexivity);

for all $(a, b) \in I$ we have $(b, a) \in I$ (symmetry);

for all $(a, b) \in I$ and all $s \in S$ we have:

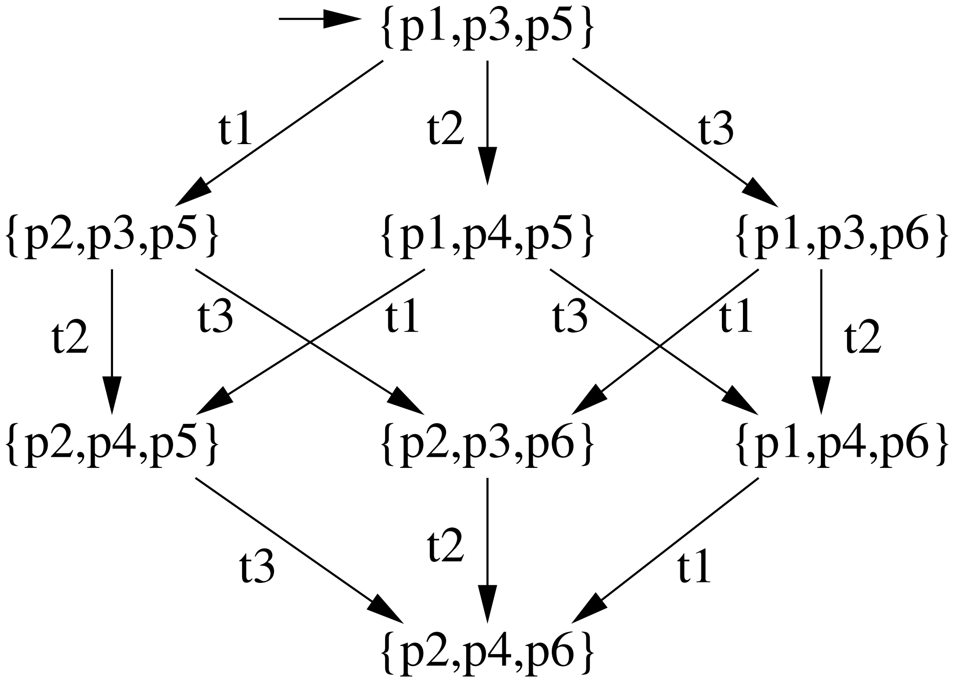
if $a, b \in en(s)$, $s \xrightarrow{a} t$, and $s \xrightarrow{b} u$,
then there exists v such that $a \in en(u)$, $b \in en(t)$, $t \xrightarrow{b} v$ and $u \xrightarrow{a} v$.

Independence



Independence: Example

In the example below all pairs of actions are independent.



Remark: In general, independence relations are not transitive!

(In-)Visibility

$U \subseteq A$ is called an invisibility set, if all $a \in U$ have the following property:

for all $(s, a, s') \in \rightarrow$ we have: $\nu(s) = \nu(s')$.

I.e., no action in U ever changes the validity of an atomic proposition.

Motivation: Interleavings of visible actions may not be eliminated because they might influence the validity of LTL properties.

Remarks

Sources for I and U: “external” knowledge about the model and the actions possible in it

e.g. addition is commutative, structural information about a Petri net,…

will *not* be obtained from first constructing all of $\mathcal{K}$!

Every (symmetric) subset of an independence relation remains an independence relation, every subset of an invisibility set remains an invisibility set.

$\rightarrow$ conservative approximation possible

But: The bigger I and U are, the more information we have at hand to improve the reduction.

Stuttering equivalence

Definition: Let σ, ρ be infinite sequences over 2^{AP}. We call σ and ρ stuttering equivalent iff there are integer sequences

$$0 = i_0 < i_1 < i_2 < \cdots \text{ and } 0 = k_0 < k_1 < k_2 < \cdots,$$

such that for all $\ell \geq 0$:

$$\begin{aligned}
&\sigma(i_\ell) = \sigma(i_\ell + 1) = \cdots = \sigma(i_{\ell+1} - 1) = \\
&\rho(k_\ell) = \rho(k_\ell + 1) = \cdots = \rho(k_{\ell+1} - 1)
\end{aligned}$$

(I.e., σ and ρ can be partitioned into “blocks” of possibly differing sizes, but with the same valuations.)

We extend this notion to Kripke structures:

Let $\mathcal{K}, \mathcal{K}'$ be two Kripke structures with the same set of atomic propositions AP.

$\mathcal{K}$ and $\mathcal{K}'$ are called stuttering equivalent iff for every sequence in $[\![\mathcal{K}]\!]$ there exists a stuttering equivalent sequence in $[\![\mathcal{K}']\!]$, and vice versa.

I.e., $[\![\mathcal{K}]\!]$ and $[\![\mathcal{K}']\!]$ contain the same equivalence classes of runs.

Invariance under stuttering

Let ϕ be an LTL formula. We call ϕ invariant under stuttering iff for all stuttering-equivalent pairs of sequences σ and ρ:

$$\sigma \in [\![\phi]\!] \quad \text{iff} \quad \rho \in [\![\phi]\!].$$

Put differently: ϕ cannot distinguish stuttering-equivalent sequences. (And neither stuttering-equivalent Kripke structures.)

Theorem: Any LTL formula without $\mathbf{X}$ is invariant under stuttering.
Proof: Exercise.

Strategy

We replace $\mathcal{K}$ by a stuttering-equivalent, smaller structure $\mathcal{K}'$.

Then we check whether $\mathcal{K}' \models \phi$, which is equivalent to $\mathcal{K} \models \phi$
(if ϕ does not contain any $\mathbf{X}$).

We generate $\mathcal{K}'$ by performing a DFS on $\mathcal{K}$, and in each step eliminating certain successor states, based on the knowledge about properties of actions that is imparted by I and U.

The method presented here is called the ample set method.

Ample sets

For every state s we compute a set $red(s) \subseteq en(s)$; $red(s)$ contains the actions whose corresponding successor states will be explored.

(partially conflicting) goals:

$red(s)$ must be chosen in such a way that stuttering equivalence is guaranteed.

The choice of $red(s)$ should reduce $\mathcal{K}$ strongly.

The computation of $red(s)$ should be efficient.

Conditions for Ample Sets

C0: $red(s) = \emptyset$ iff $en(s) = \emptyset$

C1: Every path of $\mathcal{K}$ starting at a state s satisfies the following: no action that depends on some action in $red(s)$ occurs before an action from $red(s)$.

C2: If $red(s) \neq en(s)$ then all actions in $red(s)$ are invisible.

C3: For all cycles in $\mathcal{K}'$ the following holds: if $a \in en(s)$ for some state s in the cycle, then $a \in red(s')$ for some (possibly other) state s' in the cycle.

Idea

C0 ensures that no additional deadlocks are introduced.

C1 and C2 ensure that every stuttering-equivalence class of runs is preserved.

C3 ensures that enabled actions cannot be omitted forever.

Example

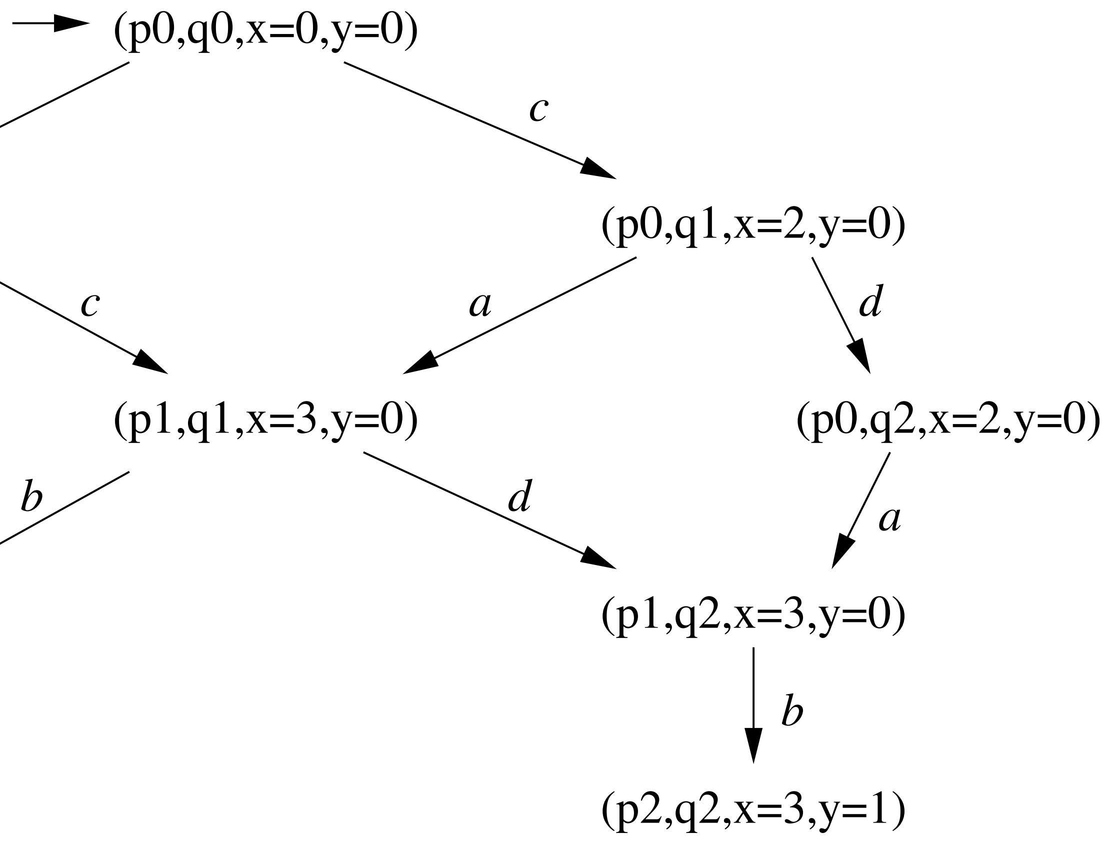
Pseudocode program with two concurrent processes:

```
**int** $x, y$ **init** 0;
**cobegin** $\{P \parallel Q\}$ **coend**
```

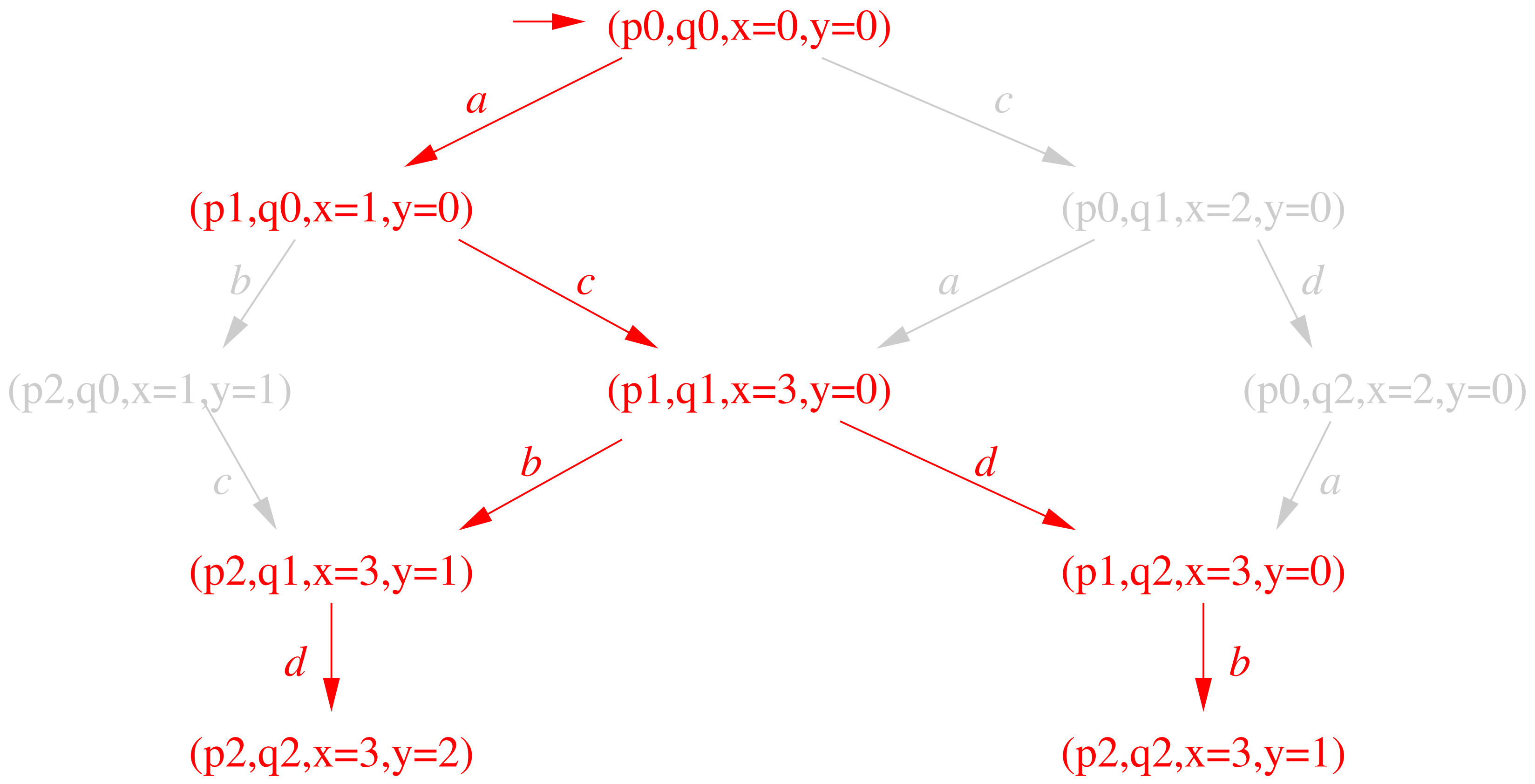
```
$P =$  p0: $x := x + 1$; (action $a$)      $Q =$  q0: $x := x + 2$; (action $c$)
       p1: $y := y + 1$; (action $b$)             q1: $y := y * 2$; (action $d$)
       p2: **end**                                q2: **end**
```

b and *d* cannot be independent.

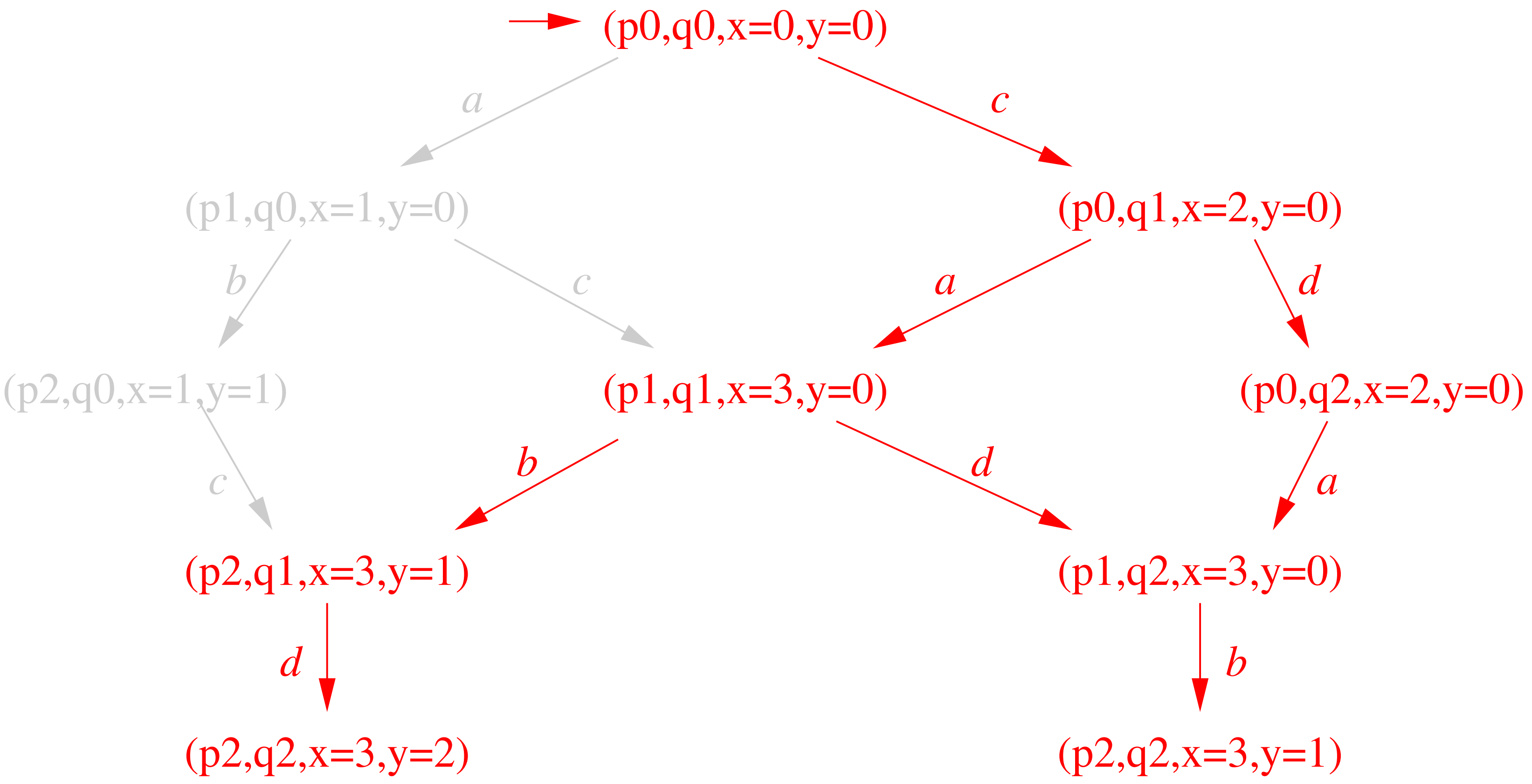
Kripke structure of the example:



Possible reduced structure if all actions are invisible:



Possible reduced structure if *a*, *d* are visible:

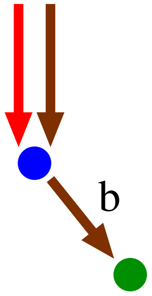


Correctness

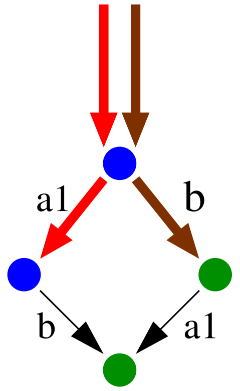
Claim: If red satisfies conditions C0 through C3, then $\mathcal{K}'$ is stuttering-equivalent to $\mathcal{K}$.

Proof (idea): Let σ be an infinite path in $\mathcal{K}$. We show that in $\mathcal{K}'$ there exists an infinite path τ such that $\nu(\sigma)$ and $\nu(\tau)$ are stuttering-equivalent.

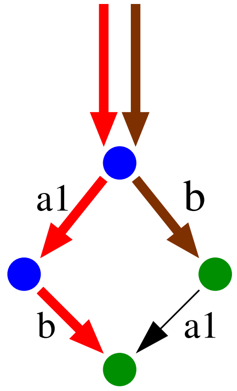
In the following, σ is shown in brown and τ in red. States known to fulfil the same atomic propositions are drawn in the same colours.



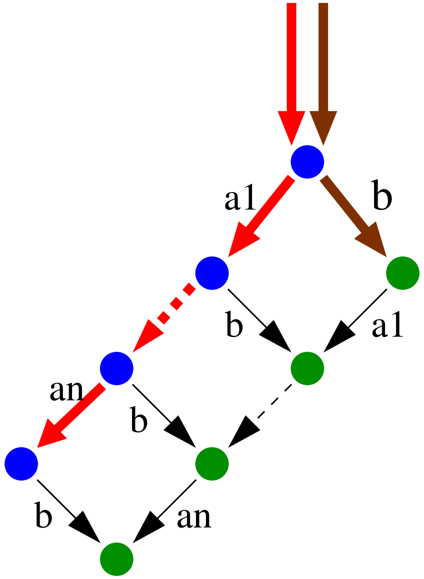
Suppose that the transition labelled with b is the first in σ that is not present in $\mathcal{K}'$.



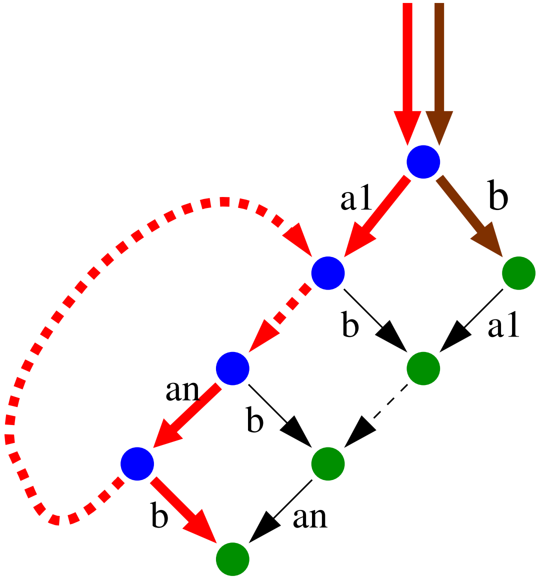
Because of C0 the blue state must have another enabled action, let us call it a_1. a_1 is independent of b (C1) and invisible (C2).



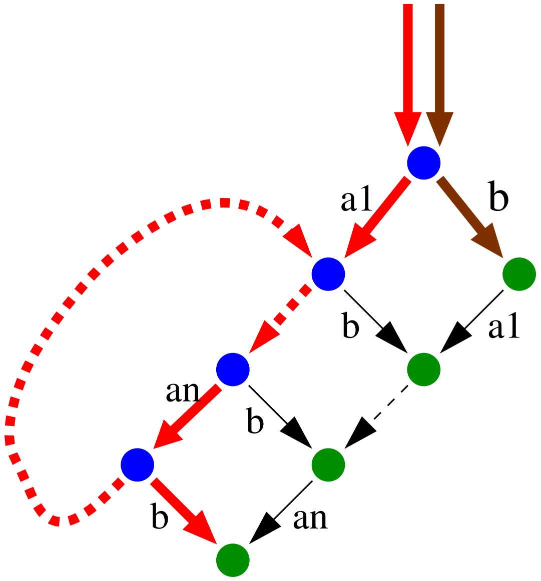
Either the second b-transition is in $\mathcal{K}'$, then we take τ to be the sequence of red edges. . .



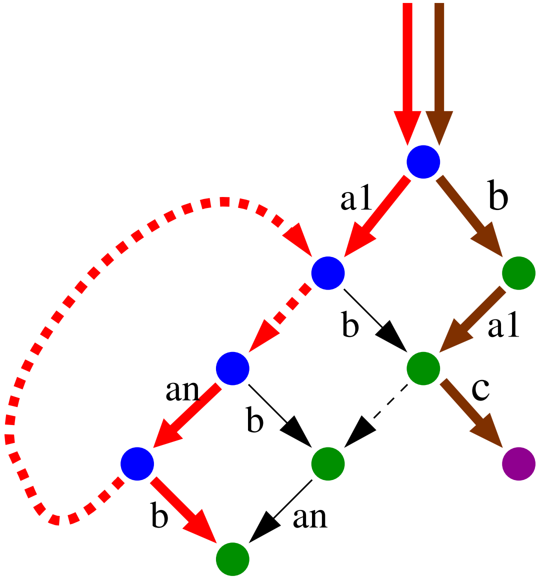
. . . or b will be “deferred” in favour of $a_2, \ldots, a_n$, all of which are also invisible and independent of b.



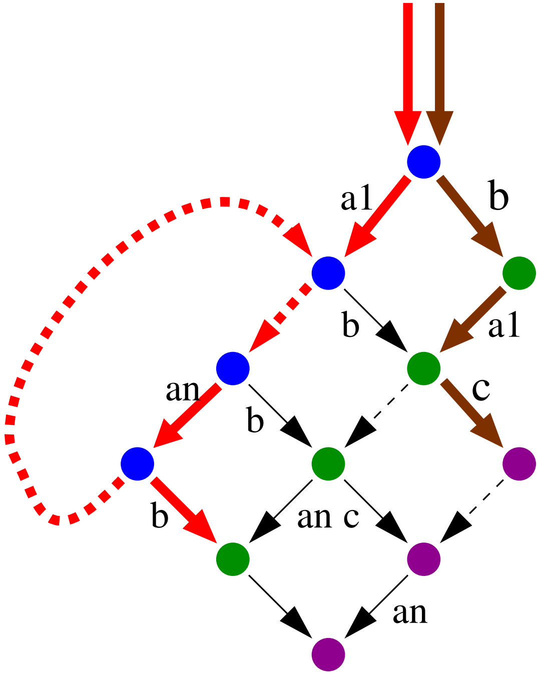
Since $\mathcal{K}$ is finite, this process must either end or create a cycle (in $\mathcal{K}'$). Because of C3, b must be activated in some state along the cycle.



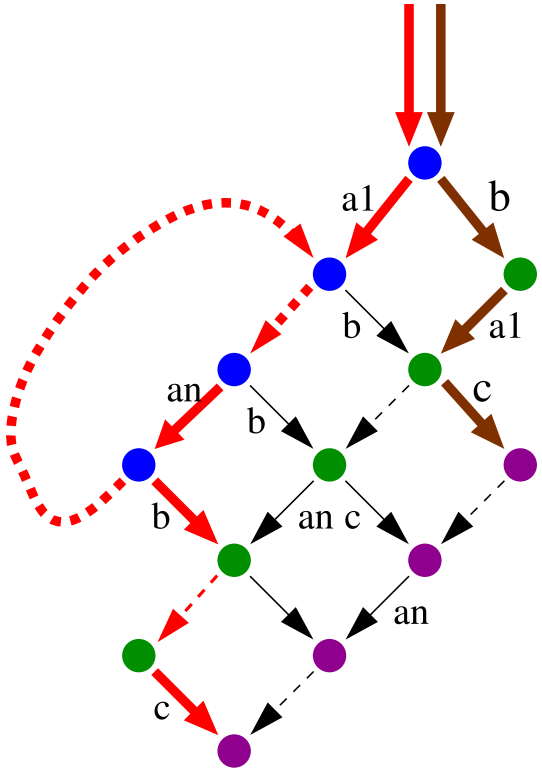
Both σ and τ contain blue states followed by green ones.



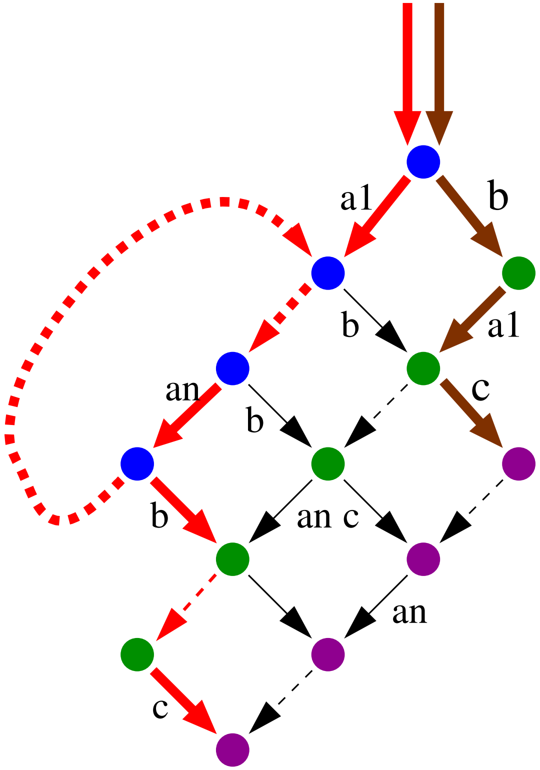
σ either continues with $a_1, \ldots, a_n$ until the paths “converge”, or it “diverges” again with an action c.



Then, however, c must be independent from $a_2, \ldots, a_n$.



Repeating the previous arguments we can conclude that $\mathcal{K}'$ also has a c-labelled transition along the red path.



Both the red and the brown path again contain blue, green, and purple states, in that order. The previous arguments can be repeated ad infinitum.

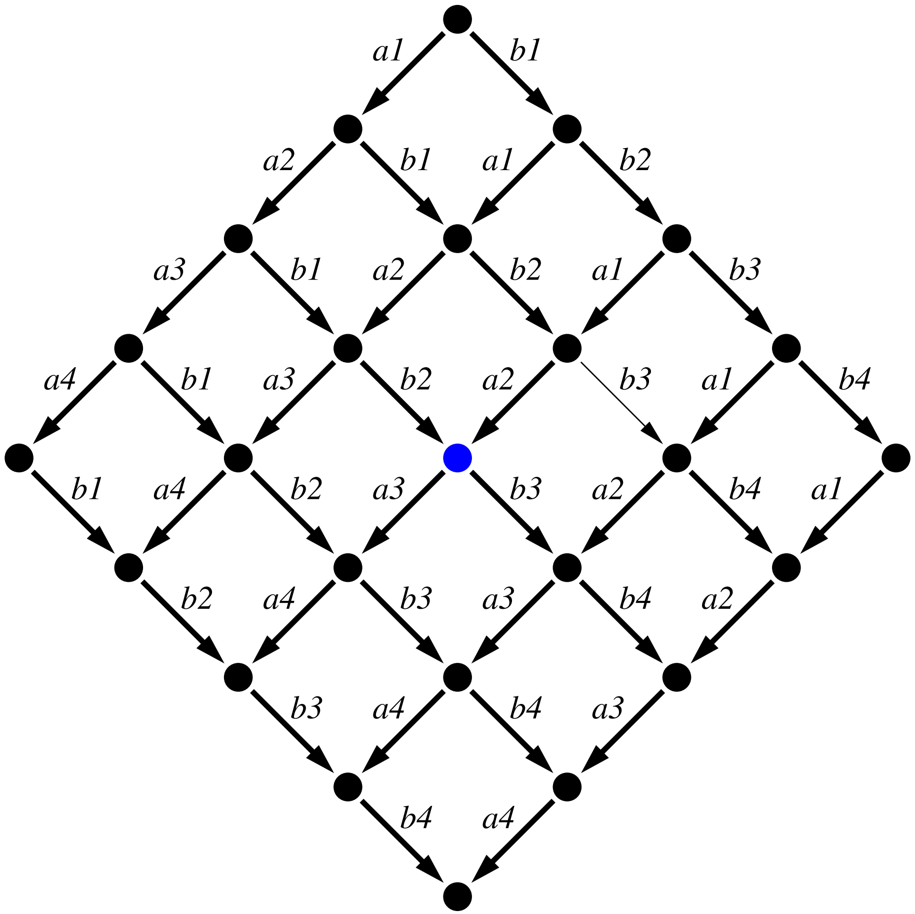
(Non-)Optimality

An ideal reduction would retain only one execution from each stuttering equivalence class.

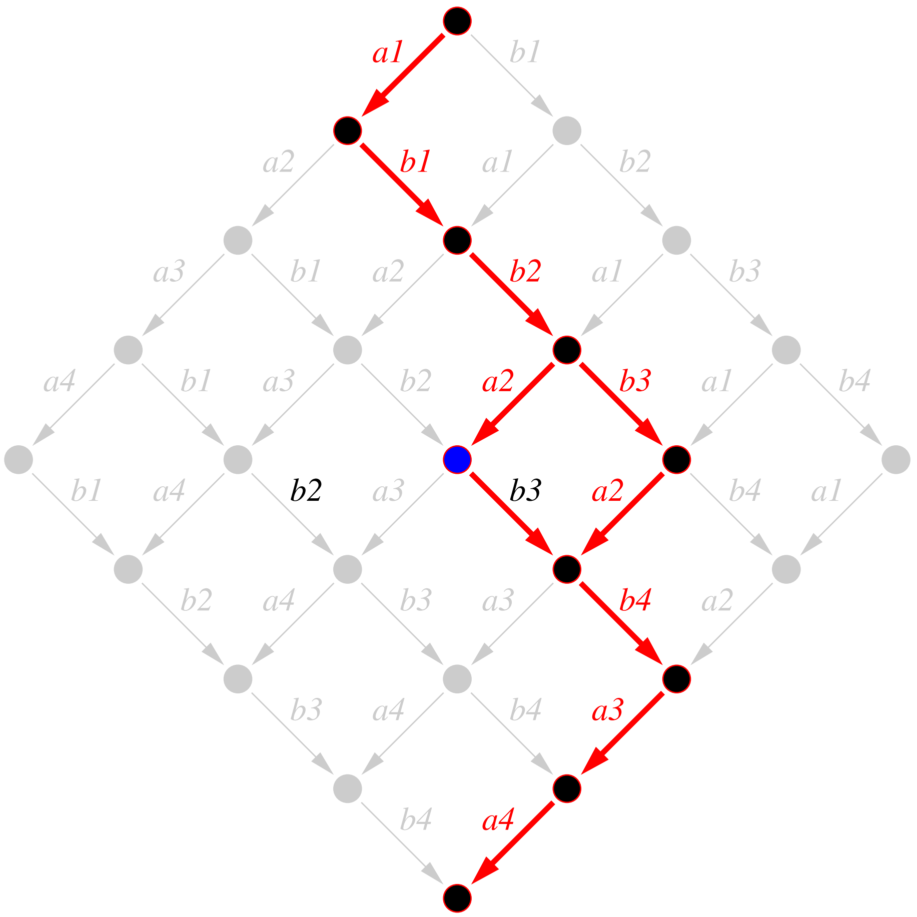
C0–C3 do *not* ensure such an ideal reduction, i.e. the resulting reduced structure is not minimal in general.

Example (see next slide): two parallel processes with four actions each ($a_1, \ldots, a_4$ or $b_1, \ldots, b_4$, resp.), all independent.

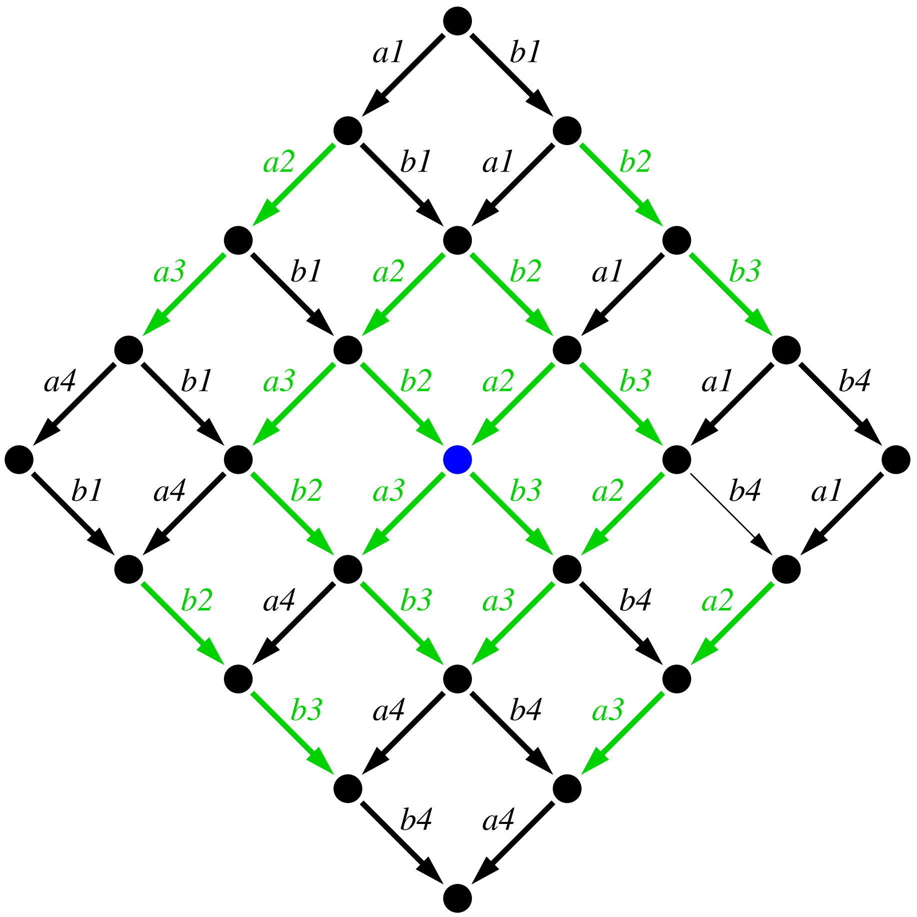
Assume that the valuation of the blue state differs from the others:



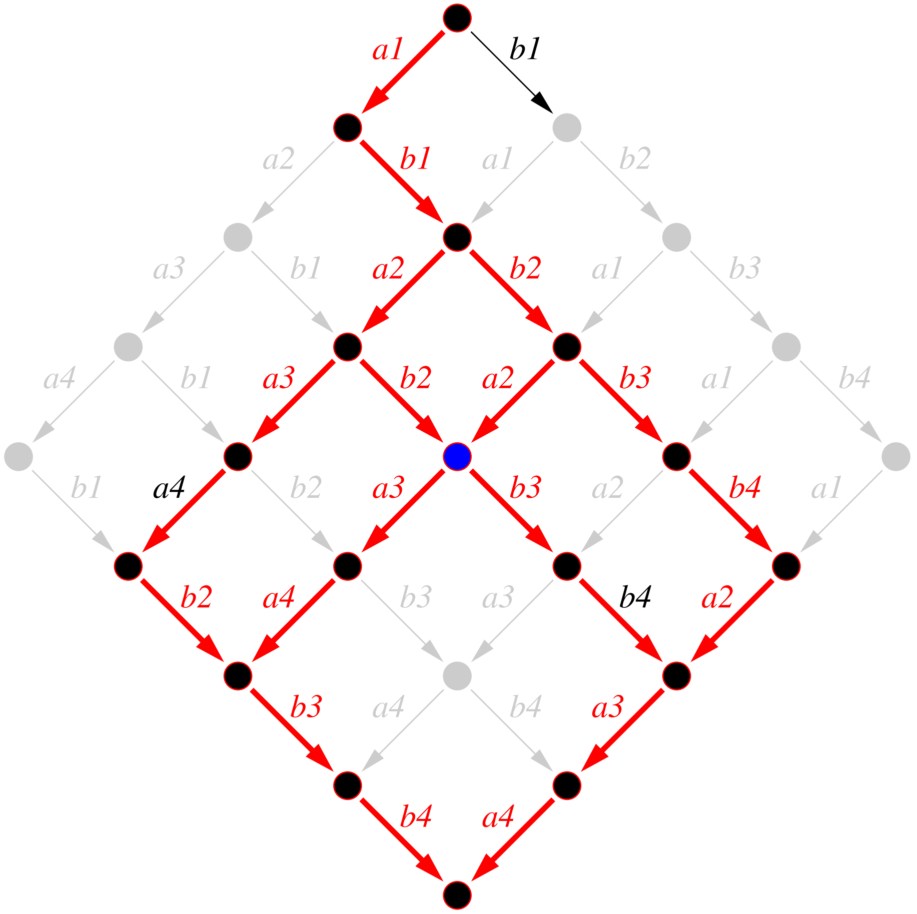
Minimal stuttering-equivalent structure:



Visible actions: a_2, a_3, b_2, b_3 (in green):



Smallest structure satisfying C0–C3:



Implementation issues

C0 and C2: obvious

C1 and C3 depend on the complete state graph

We shall find conditions that are stronger than C1 and C3. These will exclude certain reductions but can be efficiently implemented during DFS.

Replace C3 by C3':

> If $red(s) \neq en(s)$ for some state s, then no action of $red(s)$ may lead to a state that is currently on the search stack of the DFS.

Heuristic for C1

Depends on the underlying description of the system; here, we will discuss what Spin is doing for Promela models.

In general, a Promela model will contain multiple concurrent processes $P_1, \ldots, P_n$ communicating via global variables and message channels.

Let $pc_i(s)$ be the control-flow label of process P_i in state s.

Heuristic for C1 (in Spin)

Let $pre(a)$ be the set of actions that might activate a, i.e. (some overapproximation of) the set

$$\{\, b \mid \exists s\colon a \notin en(s),\ b \in en(s),\ a \in en(b(s)) \,\}$$

In Spin: Let a be an action in process P_i; then $pre(a)$ contains, e.g.,

all actions of P_i leading to a control-flow label in which a can be executed;

if the guard of a uses global variables, all actions in other processes that modify these variables;

if a reads from a message channel q or writes into it, then all actions in other processes that do the opposite (write/read) on q.

Heuristic for C1 (in Spin)

$dep(a) := \{\, b \mid (a, b) \notin I \,\}$ contains the actions dependent on a.

In Spin, if a is an action in process P_i, then $dep(a)$ will be overapproximated by:

all other actions in P_i;

actions in other processes that write to a variable from which a reads, or vice versa;

if a reads from a message channel q or writes to it, then all actions in other processes doing the same to q.

Heuristic for C1 (in Spin)

Let A_i denote the possible actions in process P_i.

$E_i(s)$ denotes the actions of process P_i activated in s:

$$E_i(s) = en(s) \cap A_i$$

$C_i(s)$ denotes the actions possible in P_i at label $pc_i(s)$:

$$C_i(s) = \bigcup_{\{\, s' | pc_i(s) = pc_i(s') \,\}} E_i(s')$$

(Some of these may not be activated in s itself because guards are not satisfied...)

The approach in Spin

Spin will test the sets $E_i(s)$, for $i = 1, \ldots, n$, as candidates for $\mathit{red}(s)$. If all these candidates fail, it takes $\mathit{red}(s) = \mathit{en}(s)$.

A violation of C1 implies that some action a depending on $E_i(s)$ might be executed before an action from $E_i(s)$ itself. To check whether such a violation is possible, Spin checks the following two conditions:

Either a belongs to some other process P_j, $j \neq i$.
⇒ check whether $A_j \cap dep(E_i(s)) \neq \emptyset$

Otherwise $a \in C_i(s) \setminus E_i(s)$.
⇒ check whether $C_i(s) \setminus E_i(s)$ may be activated by some other process.

Test for C1

```
function $\mathit{check\_C1}(s, P_i)$
    for all $P_j \neq P_i$ do
        if $\mathit{dep}(E_i(s)) \cap A_j \neq \emptyset \vee$
          $\mathit{pre}(C_i(s) \setminus E_i(s)) \cap A_j \neq \emptyset$ then
            return $\mathit{False};$
        end if;
    end for all;
    return $\mathit{True};$
end function
```