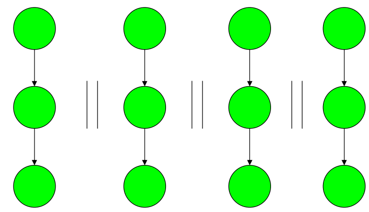
Partial Order Reduction

Ralf Huuck

Australian Government
Department of Communications, Information Technology and the Arts
Australian Research Council
NICTA Members
NICTA Partners

The Problem

Many concurrent components:



Trying to build the product state space ...

State Explosion

What to do?

Try minimizing the effect by reduction heuristics, e.g.: Partial Order Reduction

Worst case: number of states increases exponentially with number of processes.

Overview

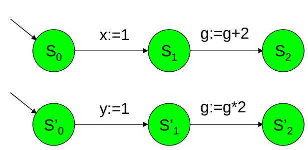
- Informal explanation
- Framework for partial order reduction (POR)
- POR in SPIN
- Summary

The imagination driving Australia's ICT future.

Introduction

The imagination driving Australia's ICT future.

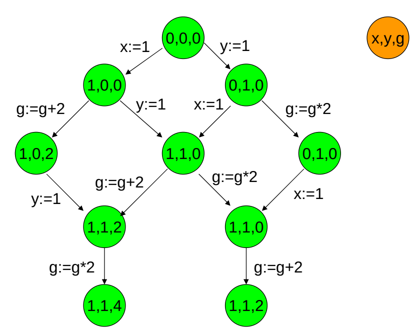
Motivation



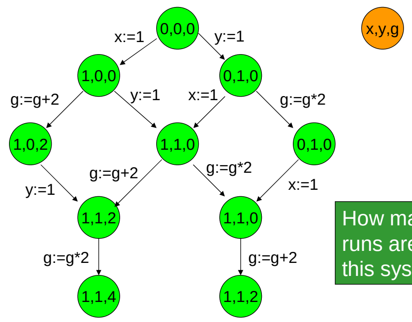
consider interleaving execution,
what are the possible runs?

The imagination driving Australia's ICT future.

Expanded Asynchronous Product

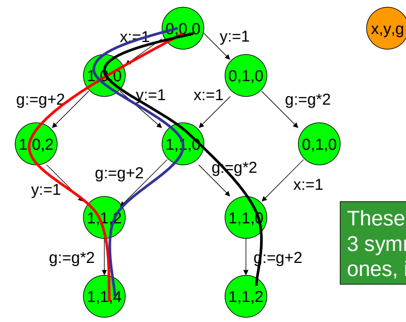


Expanded Asynchronous Product



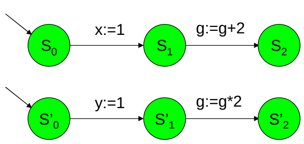
How many runs are in this system?

Possible Runs



These 3 plus 3 symmetric ones, i.e., 6

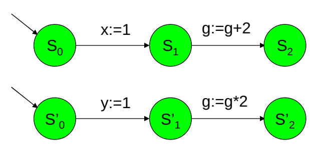
Dependencies (1)



assume x, y are local variables,
g is a global variable

Which operations are actually dependent and which are independent?

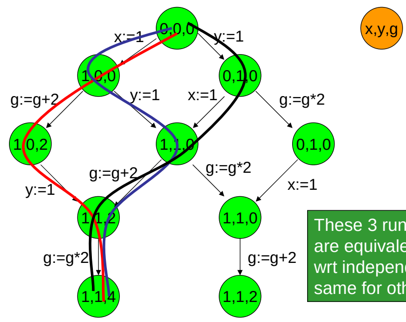
Dependencies (2)



Dependent:
- g:=g+2, g:=g*2 share same object
- x:=1, g:=g+2 ordered in same automaton
- y:=1, g:=g*2 ordered in same automaton

Independent:
- x:=1, y:=1
- x:=1, g:=g*2
- y:=1, g:=g+2

Equivalent Runs

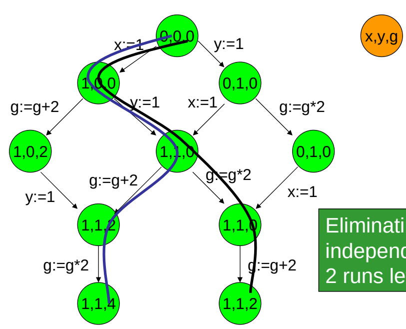


These 3 runs are equivalent wrt independencies, same for other 3 runs

Idea

- partitioning into equivalent classes
- we have to select one run in each class only

Necessary Runs



Eliminating all independencies. 2 runs left

Proving Properties

- $G(g=0 \vee g>x)$
- $F(g\geq 2)$
- $(g=0)U(x=1)$

all hold in reduced graph, i.e., considering only 2 necessary runs

Proving Properties

- G(g=0 ∨ g>x)
- F(g≥2)
- (g=0)U(x=1)

all hold in full and reduced graph, with states of the 2 necessary runs

- G(x≥y)

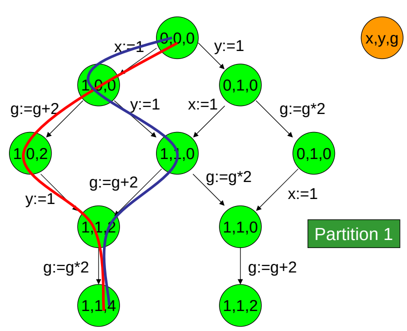
holds in reduced graph, but not full graph

WHY?

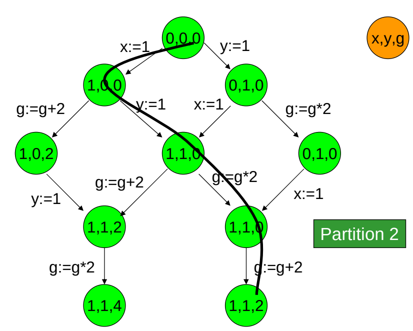
Visibility

- introduces dependency that was not assumed to exist
- dependencies not only from data objects but also formula
- remove x:=1, y:=1 from independencies

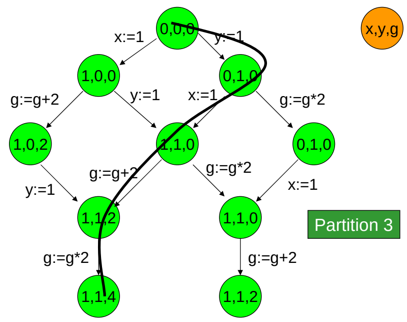
Equivalent Runs



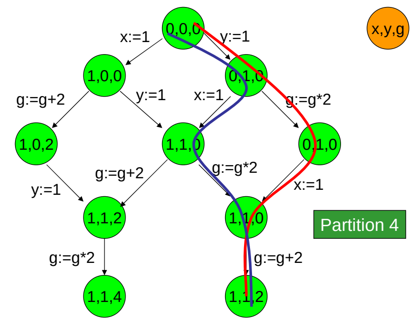
Equivalent Runs



Equivalent Runs



Equivalent Runs



Questions

- Given a set of processes how can we automatically identify classes of equivalent runs?
- How to avoid full construction upfront, but deciding on-the-fly which states and transitions are necessary?

Such techniques are addressed as partial order reduction, which, e.g., SPIN makes use of.

Theory

Labeled Transition System

(S,s_0,A,τ,Π,L) is labeled transition system where

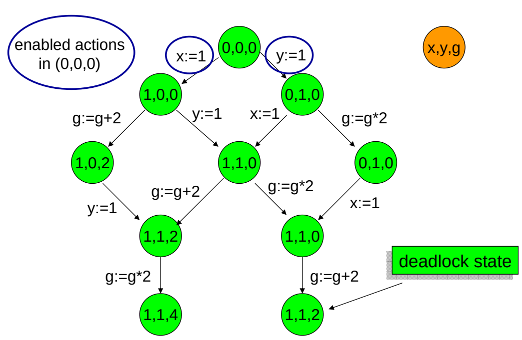
- S finite set of states
- s_0 initial state
- A finite set of actions
- $\tau: S\times A\rightarrow S$ (partial) transition function
- Π finite set of Boolean propositions
- $L:S\rightarrow 2^{\Pi}$ labeling function

(similar to a Kripke structure with symbols on transitions)

enabled/reachable

- action $a\in A$ is enabled in state $s\in S$ iff $\tau(a,s)$ is defined
- enabled(s) denotes set of all actions enabling in transition from state s
- sate s is deadlock state iff enabled(s)=$\emptyset$
- execution sequence is sequence of subsequent transitions
- state s is reachable iff there exists an execution sequence from s_0 to s

Example



Partial Order Reduction

- avoid construction including "unnecessary" interleavings if possible
- decide per state which outgoing transitions to include
- reduction function $r:S\rightarrow 2^A$, i.e., which actions have to be taken care of in a certain state

Reduced LTS

smallest $(S_r, s_{0r}, A_r, \tau_r, \Pi_r, L_r)$ such that

- $S_r \subseteq S$,
- $s_0 = s_{0r}$,
- $L_r = L \cap (S_r \times 2^{\Pi})$
- for any $s \in S_r$ and $a \in r(s)$ where $\tau(s,a)$ is defined, $\tau_r(s,a)$ is defined

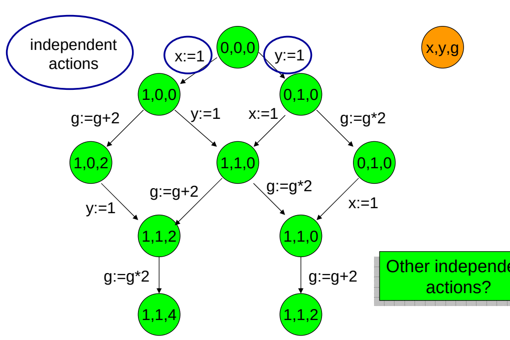
Independence

two actions $a,b \in A$ $(a \neq b)$ are independent
iff for all states $s \in S$ where $\{a,b\} \subseteq enabled(s)$

1. $b \in enabled(\tau(s,a))$ and $a \in enabled(\tau(s,b))$
2. $\tau(\tau(s,a),b) = \tau(\tau(s,b),a)$

This means actions do not disable each other (1) and their permutation leads to the same state (2).

Example



Other independent actions?

Proving Properties

Properties

POR is typically done with respect to certain classes of properties, e.g.:

- absence of deadlock,
- local property, depends on state of a single process or state of single shared object
- next-free LTL property, i.e., LTL with until operator only

Preserving Deadlock

To preserve deadlock states the reduction function must satisfy:

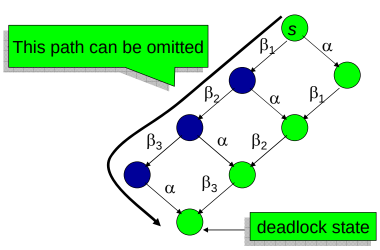
C0 $r(s)=\emptyset$ iff $enabled(s)=\emptyset$

C1 (persistency) for any execution sequence

$$s = s_0 \stackrel{a_0}{\rightarrow} s_1 \stackrel{a_1}{\rightarrow} \ldots \stackrel{a_{n-1}}{\rightarrow} s_n$$

with all $a_i \notin r(s)$ $(0 \leq i < n)$, a_{n-1} is independent of all $a_i \in r(s)$

Example



Theorem

Any reduced system satisfying C0 and C1 preserves deadlocks.

Local Properties

property ϕ is local

iff

for all $s \in S$ and independent actions $a,b \in A$

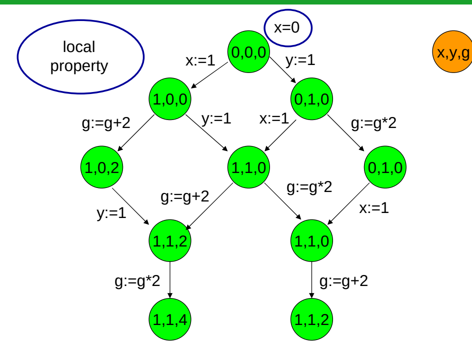
if $\{a,b\} \subseteq enabled(s)$ then:

if ϕ holds in s but not in $\tau(s,a)$

then ϕ holds in $\tau(s,b)$ but not in $\tau(\tau(s,b)a)$.

Intuition: ϕ cannot be changed by the combined effect of two independent actions, it only depends on local changes.

Example



Preserving Local Properties

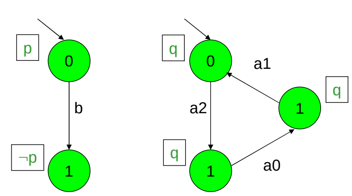
To preserve local properties the reduction function must satisfy:

C2 (cycle) for any cyclic execution sequence

$$s = s_0 \stackrel{a_0}{\longrightarrow} s_1 \stackrel{a_1}{\longrightarrow} \ldots \stackrel{a_{n-1}}{\longrightarrow} s_n$$

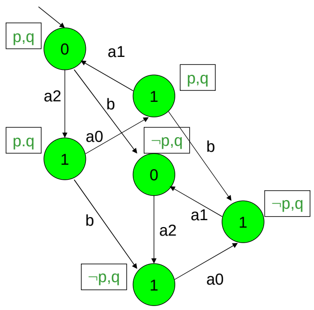
where, $s_n=s_0$ there is an s_i ($0 \leq i<n$) such that $r(s_i)=enabled(s_i)$

Example



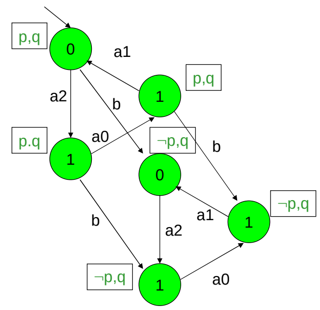
Two concurrent processes,
a's and b are independent

Full State Graph



Full State Graph

a's and b are independent, whenever having the choice between them, why not choosing some a?

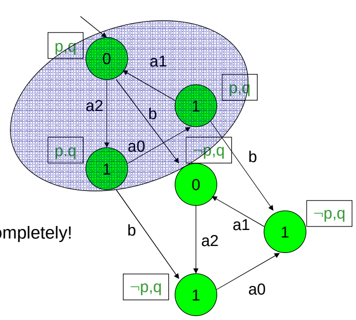


Reduced State Graph?

This means, we never see b and never ¬p.

C2 requires in any cycle there is an s_i ($0 \leq i < n$) such that $r(s_i) = enabled(s_i)$.

Therfore, cannot hide ¬p completely!



Theorem

Any reduced system satisfying C0, C1, and C2 preserves local properties.

Next-free LTL

- only allows Until as temporal operator,
- strict subset of LTL
- cannot, e.g., distinguish between the next and the second next state
- closed under stuttering

Invisibility

prop(ϕ) set of propositions in ϕ

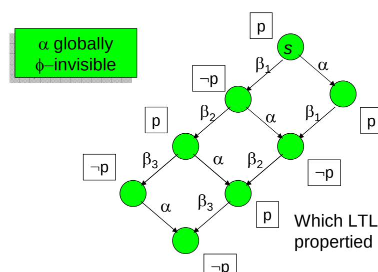
- action a is ϕ-invisible in s **iff** τ(s,a) is undefined or $\pi \in L(s) \Leftrightarrow \pi \in L(\tau(s,a))$ for all $\pi \in$ prop(ϕ)
- a is globally ϕ-invisible **iff** it is ϕ-invisible for all s$\in$S

This means some action cannot change some truth value.

Preserving Next-free LTL

C3 (invisibility) for any state s$\in$S,
all actions are globally ϕ-invisible or r(s)=enabled(s)

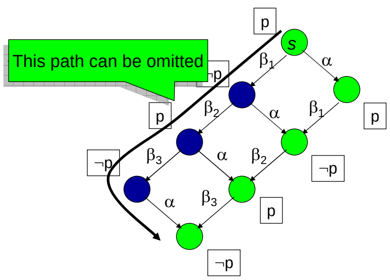
Example (1)



Which LTL and/or next-free LTL propertied do (not) hold here?

More sophisticated examples?

Example (2)



Theorem

Any reduced system satisfying
C0, C1, C2, and C3
preserves next-free LTL properties.

Well, yes but ...

- We defined constraints such that a reduced system still satisfies certain properties.
- But: How to find a suitable reduction?
- Also: building full state graph and then reducing is inefficient.

Challenging!

Let's have a look at SPIN ...

POR in SPIN

System Construction in SPIN

1. depth first search
2. reduction function based on process structure

Preliminaries

(S,s_0,A,τ,Π,L) full LTS from set of processes $\mathcal{P}$
each process $P\in\mathcal{P}$ is set of actions, i.e., $P\subseteq A$

we assume: $\mathcal{P}$ is a partitioning of A, i.e,

1. $P,Q\in\mathcal{P}$, $P\neq Q \Rightarrow P\cap Q=\emptyset$, and
2. $A=\bigcup_{P\in\mathcal{P}}P$

$Pid:A\rightarrow\mathcal{P}$ returns process (ID) for a given action

Restriction of Process Structure

We do not allow concurrency within a process:

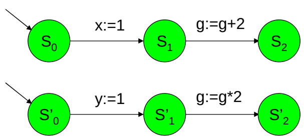
for all $a,b\in P$, $a\neq b$, $s\in S$:
$a,b\in enabled(s) \Rightarrow b\notin enabled(\tau(s,a))$

This means we still have choice (if-then-else) in a process, but no processes within processes.

Safety

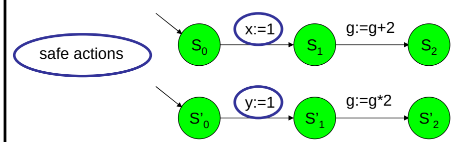
Action a is safe
iff
it is independent from any b where $Pid(a)\neq Pid(b)$

Safety Example



Which actions are safe in this example?

Safety Example



They are independent of any action in other process.

Next-free Safety

Action a is safe

iff

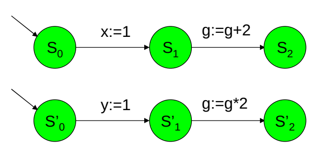
it is independent from any b where Pid(a)≠Pid(b)

Action a is next-free safe for some $\phi \in LTL_{-X}$

iff

- it is independent from any b where Pid(a)≠Pid(b), and
- globally ϕ-invisible

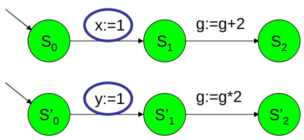
Next-free Safe Example



Which actions are next-free safe for:

- G (g=2)
- G (x<g)

Next-free Safe Example



next-free safe actions for ϕ=G (g=2)

Other (counter)examples?

Reduction Function Ample (part 1)

Let s∈S be a state. Let P∈$\mathcal{P}$ be a process such that

1. enabled(s)∩ P ≠∅
2. for all a∈enabled(s)∩P, a is (next-free) safe
3. for all a∈enabled(s)∩P, τ(s,a) is not on DFS stack

Reduction Function Ample

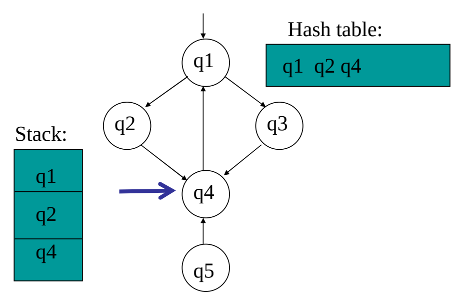
Let s∈S be a state. Let P∈$\mathcal{P}$ be a process such that

1. enabled(s)∩ P ≠∅
2. for all a∈enabled(s)∩P, a is (next-free) safe
3. for all a∈enabled(s)∩P, τ(s,a) is not on DFS stack

.

Remember DFS algorithm? Stack keeps record of states we have seen before, but not fully explored.

Reminder: DFS Algorithm



Reduction Function Ample (part 2)

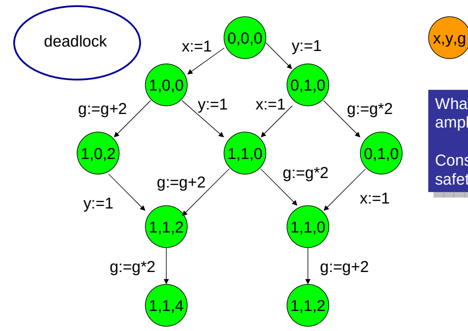
Let s∈S be a state. Let P∈$\mathcal{P}$ be a process such that

1. enabled(s)∩ P ≠∅
2. for all a∈enabled(s)∩P, a is (next-free) safe
3. for all a∈enabled(s)∩P, τ(s,a) is not on DFS stack

We define a reduction function ample as follows:

- if there is no such process then ample(s)=enabled(s).
- otherwise choose arbitrary P satisfying above requirements and define ample(s)=enabled(s)∩P.

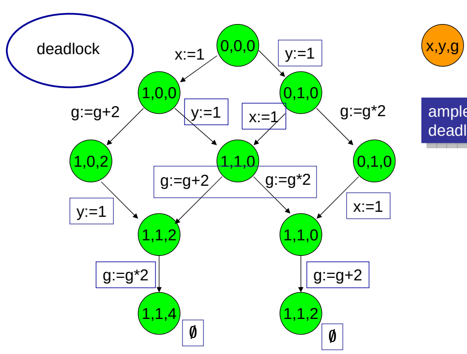
Example (POR deadlock)



What are the ample sets?

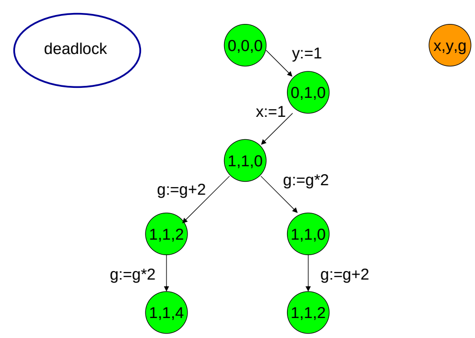
Consider simple safety only.

Example (POR deadlock)

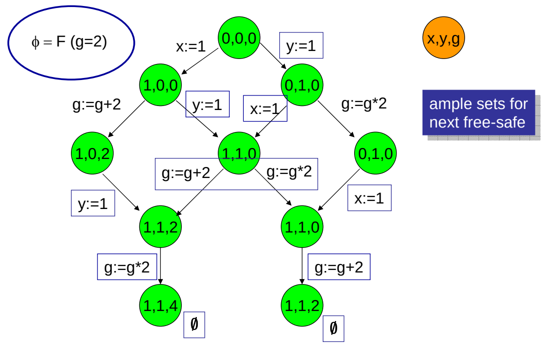


ample sets for deadlock

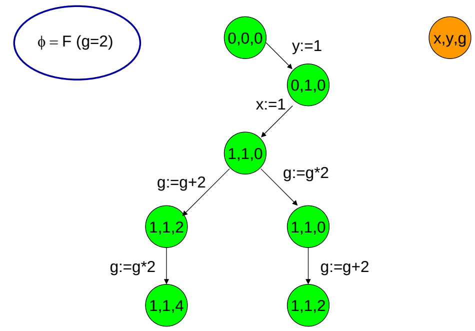
Reduction (POR deadlock)



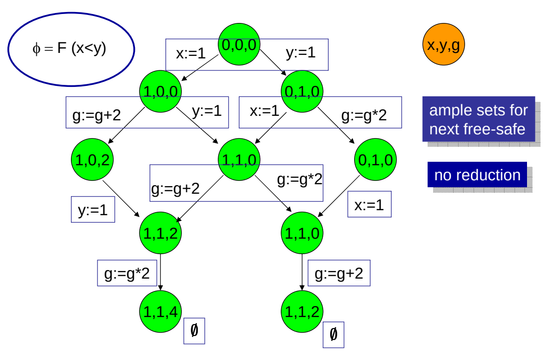
Example (2)



Reduction (2)



Example (3)



On-the-fly Construction

Constructing full state space first and then reducing it is not very smart, but:

- We can do POR while construction the state space

Basically, use DFS algorithm for state space construction and only follow the paths in the ample sets.

POR does not always help, but the more independent actions the better.

Summary

The imagination driving Australia's ICT future.

Partial Order Reduction

- tackles state explosion
- general framework for reduction
- SPIN example for implementation of reduction function
- other methods out there, e.g., symmetry reduction, automata minimizations, abstractions etc.

The imagination driving Australia's ICT future.

Good news ☺

We are done with “standard” model checking.