Seminar 5: SMT & SPIN

CEGAR Refinement Step, SMT, Promela, SPIN

Di Marco, Okwieka

University of Rome "La Sapienza"

May 29th, 2023

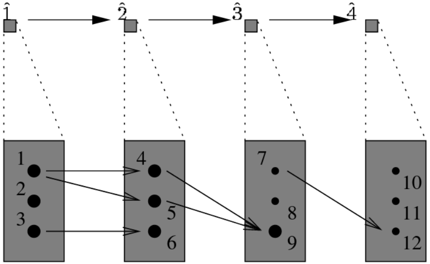
CEGAR Refinement Step

SplitPATH: Identifying spurious finite path counterexamples

Clarke et al. 2003 (figure and pseudocode copied verbatim)

Define $h^{-1}(\widehat{s}) = \{s \mid h(s) = \widehat{s}\}$ and lift to paths $\widehat{\mathrm{T}} = \langle \widehat{s_1}, \ldots, \widehat{s_n} \rangle$

$$\text{Concrete counterexample} \iff h^{-1}_{path}(\widehat{\mathrm{T}}) \neq \emptyset \iff S_i \neq \emptyset$$



Algorithm SplitPATH

$S := h^{-1}(\widehat{s_1}) \cap I$

$j := 1$

while $(S \neq \emptyset$ and $j < n)$ {

  $j := j + 1$

  $S_{\mathrm{prev}} := S$

  $S := Img(S, R) \cap h^{-1}(\widehat{s_j})$ }

if $S \neq \emptyset$ **then** output counterexample

else output j, S_{prev}

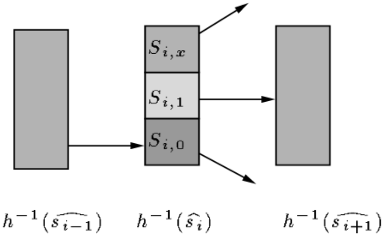
`PolyRefine`: Refinement Step

Clarke et al. 2003 (figure copied verbatim)

For i so that $Img(S_i, R) \cap h^{-1}(\widehat{s_{i+1}}) = \emptyset$ and S_i reachable, we partition $h^{-1}(\widehat{s_i})$ into $S_{i,0}$ (reachable), $S_{i,1}$ (outgoing), $S_{i,x}$ (isolated)

$$\text{Spurious } \widehat{s_i} \to \widehat{s_{i+1}} \iff S_{i,1} \neq \emptyset$$

Split abstraction relation $\equiv$ into coarsest $\equiv'$ s.t. $S_{i,0}$ and $S_{i,1} \cup S_{i,x}$ are separate.
Can be done in polynomial time if no need for optimal (coarsest) refinement.



PolyRefine: Refinement Step

Clarke et al. 2003

For every equivalence class $h^{-1}(\widehat{s}_j) = E_j$ of abstraction $\equiv$, define $\equiv_j$ as equivalence relation with one class, E_j.

Want to find refinement for each. Initially $\equiv'_j \leftarrow \equiv_j$.

For all $a, b \in E_j$: if $a \in S_{i,0}$ and $b \in S_{i,1} \cup S_{i,x}$, remove (a, b) from $\equiv'_j$.

$\implies$ Splits E_j into two subclasses, assign each to new abstract state.

SMT

Satisfiability Modulo Theories

Classic SAT: "find Boolean variable assignment s.t. Boolean formula holds"

Conjunctive Normal Form of OR-*clauses* of *literals* connected with AND:

$$(x_1 \vee \neg x_2 \vee \ldots \vee x_n) \wedge (x_{n+1} \vee \ldots \neg x_{n+m}) \wedge \ldots$$

Literal in Boolean formula: variable or its negation

Literal in SMT formula: formula in quantifier-free First-Order Logic (FOL) theory, e.g. Linear Rational Arithmetic:

$$(\neg(2v_1 + v_2 \leq 3) \vee A_1) \wedge (3v_2 - 2v_1 < 6) \vee \ldots$$

Quantifier-free First-Order (= Predicate) Logic and Theories

FOL: non-Boolean ("domain") variables, predicates over them, quantifiers

FOL Signature Σ: set of symbols (functions, predicates, constants)

Σ-FOL Theory $\mathcal{T}$: set of Σ-formulas constrained by *axioms*

$\Longrightarrow$ axioms *interpret* the symbols of the signature

Decidable Theory: there is an efficient procedure for "Is a formula included in the theory?"

Usually more interested in "Is this conjunction of $\mathcal{T}$-literals *consistent*?"

Examples of Decidable FOL Theories

Equality and Uninterpreted Functions

$$f(g(x, y)) = h(y, f(x))$$

- Equality axioms $a = a$ and $a = b \iff b = a$ and $a = b \wedge b = c \implies a = c$
- Uninterpreted functions: only defined by name and arity (argument number)

Linear Arithmetic over the Integers

$$(a + b < 3) \wedge (b = 2) \wedge (a > 0) \implies (a = 1)$$

- Clauses are (in-)equalities of arithmetic expressions
- Arithmetic expressions: constant numbers, operators, variables

Building a SMT Solver: Lazy Approach

SMT formula φ $\mathcal{T}$-SAT $\iff$ $\varphi^p \wedge \tau^p$ SAT where

- φ^p Boolean abstraction of φ: every $\mathcal{T}$-atom corresponds to a Boolean variable B_i
- τ^p Boolean abstraction of all $\mathcal{T}$-lemmas on atoms in φ (not explicitly constructed as Boolean formula)

SAT solver: find $\mu^p \models \varphi^p$, then ask $\mathcal{T}$-solver whether $\mu^p \models \tau^p$

$\mathcal{T}$**-solver(**μ**):** determine consistency of $\mathcal{T}$-literals conjunction μ, give *conflict set* $\eta \subseteq \mu$ if inconsistent (i.e. return falsified clauses)

Classic SAT: DPLL Algorithm

Backtracking algorithm with eager rules reducing search space. Observations:

$$(0 \vee x_2 \vee \ldots \vee x_n) \text{ SAT} \iff (x_2 \vee \ldots \vee x_n) \text{ SAT} \tag{1}$$
$$(1 \vee x_2 \vee \ldots \vee x_n) \text{ SAT trivially} \tag{2}$$

Unit propagation: for unit clause x_i, assign $x_i \leftarrow 1$. Conversely for $\neg x_i$.

Remove all clauses containing this literal (2), remove complement from all other clauses (1).

Pure literal elimination: if variable x_i only occurs as literal x_i or as $\neg x_i$ throughout formula, assign it appropriately and drop containing clauses (2).

Termination: SAT if all clauses satisfied, UNSAT if there is an unsatisfied clause for all variable assignments (exhaustive search)

Classic SAT: DPLL Algorithm Example (1.0)

Action to do: Choose $a = 0$.

$c_1 : \neg a \vee b \vee c$

$c_2 : a \vee c \vee d$

$c_3 : a \vee c \vee \neg d$

$c_4 : a \vee \neg c \vee d$

$c_5 : a \vee \neg c \vee \neg d$

$c_6 : \neg b \vee \neg c \vee d$

$c_7 : \neg a \vee b \vee \neg c$

$c_8 : \neg a \vee \neg b \vee c$

Classic SAT: DPLL Algorithm Example (1.1)

$c_2 : c \vee d$

$c_3 : c \vee \neg d$

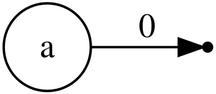
$c_4 : \neg c \vee d$

$c_5 : \neg c \vee \neg d$

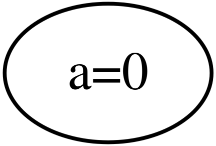
$c_6 : \neg b \vee \neg c \vee d$

Action taken: Choose $a = 0$.

Tree:



Implication Graph:



Classic SAT: DPLL Algorithm Example (1.2)

$c_2 : c \vee d$

$c_3 : c \vee \neg d$

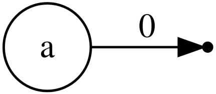
$c_4 : \neg c \vee d$

$c_5 : \neg c \vee \neg d$

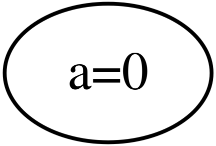
$c_6 : \neg b \vee \neg c \vee d$

Action to do: Pure literal elim. $b = 0$.

Tree:



Implication Graph:



Classic SAT: DPLL Algorithm Example (1.3)

$c_2 : c \vee d$

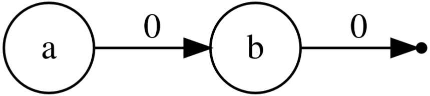
$c_3 : c \vee \neg d$

$c_4 : \neg c \vee d$

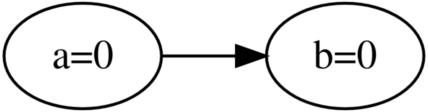
$c_5 : \neg c \vee \neg d$

Action taken: Pure literal elim. $b = 0$.

Tree:



Implication Graph:



Classic SAT: DPLL Algorithm Example (2.0)

$c_2 : c \vee d$

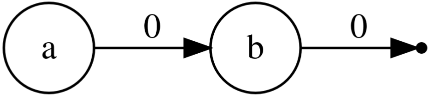
$c_3 : c \vee \neg d$

$c_4 : \neg c \vee d$

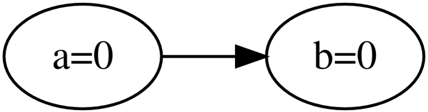
$c_5 : \neg c \vee \neg d$

Action to do: Choose $c = 0$.

Tree:



Implication Graph:



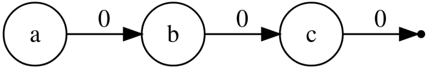
Classic SAT: DPLL Algorithm Example (2.1)

$c_2 : d$

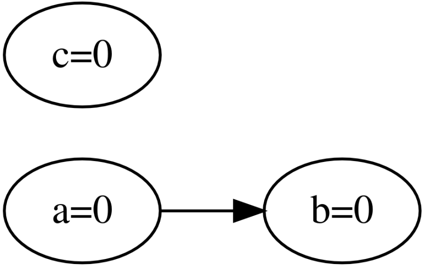
$c_3 : \neg d$

Action taken: Choose $c = 0$.

Tree:



Implication Graph:



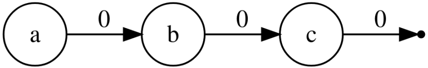
Classic SAT: DPLL Algorithm Example (2.2)

$c_2 : d$

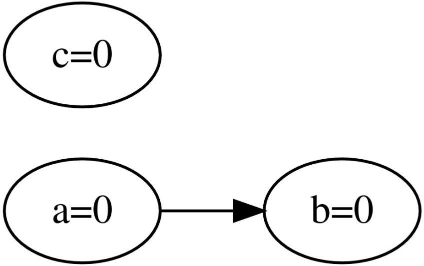
$c_3 : \neg d$

Actions to do: Unit prop. $d = 1$ and $d = 0$.

Tree:



Implication Graph:



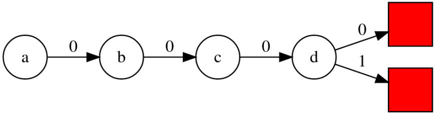
Classic SAT: DPLL Algorithm Example (2.3)

$c_4 : d$

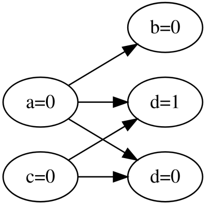
$c_5 : \neg d$

Actions not taken: Unit prop. $d = 1$ and $d = 0$.

Tree:



Implication Graph:



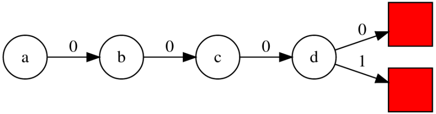
Classic SAT: DPLL Algorithm Example (2.4)

$c_2 : d$

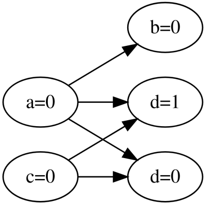
$c_3 : \neg d$

Action to do: Backtrack to last step.

Tree:



Implication Graph:



Classic SAT: DPLL Algorithm Example (3.0)

$c_2 : c \vee d$

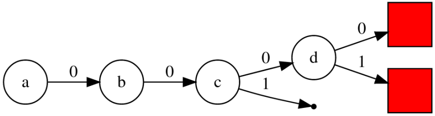
$c_3 : c \vee \neg d$

$c_4 : \neg c \vee d$

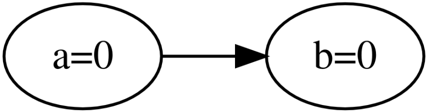
$c_5 : \neg c \vee \neg d$

Action to do: Choose $c = 1$.

Tree:



Implication Graph:



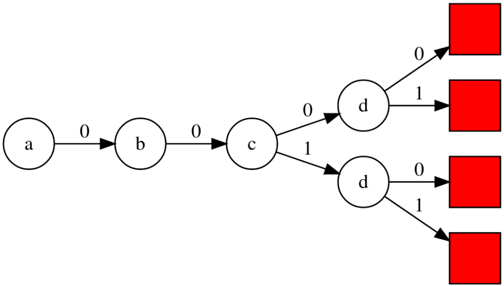
Classic SAT: DPLL Algorithm Example (3.x)

$c_4 : d$

$c_5 : \neg d$

Actions taken: Choose $c = 1$, and continue (omitted).

Tree:



Classic SAT: DPLL Algorithm Example (4.0)

$c_1 : \neg a \vee b \vee c$

$c_2 : a \vee c \vee d$

$c_3 : a \vee c \vee \neg d$

$c_4 : a \vee \neg c \vee d$

$c_5 : a \vee \neg c \vee \neg d$

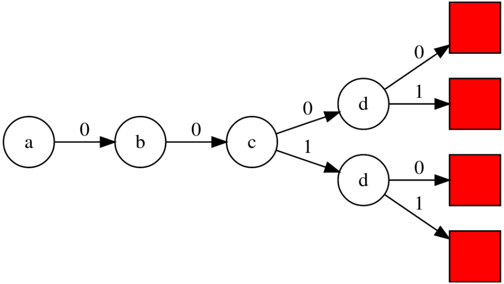
$c_6 : \neg b \vee \neg c \vee d$

$c_7 : \neg a \vee b \vee \neg c$

$c_8 : \neg a \vee \neg b \vee c$

Actions taken: Backtrack to a, since b pure.

Tree:



Classic SAT: DPLL Algorithm Example (4.1)

$c_1 : b \vee c$

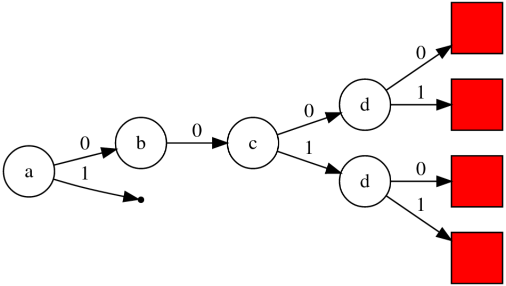
$c_6 : \neg b \vee \neg c \vee d$

$c_7 : b \vee \neg c$

$c_8 : \neg b \vee c$

Actions taken: Choose $a = 1$.

Tree:



Classic SAT: DPLL Algorithm Example (4.2)

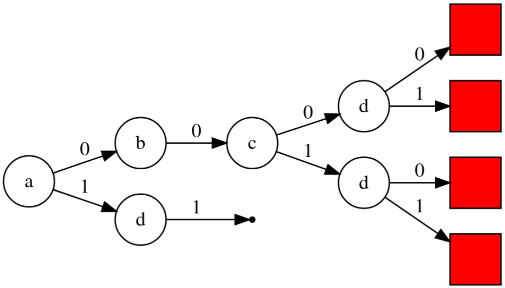
$c_1 : b \vee c$

$c_7 : b \vee \neg c$

$c_8 : \neg b \vee c$

Actions taken: Pure literal elim. $d = 1$.

Tree:



Classic SAT: DPLL Algorithm Example (5.x)

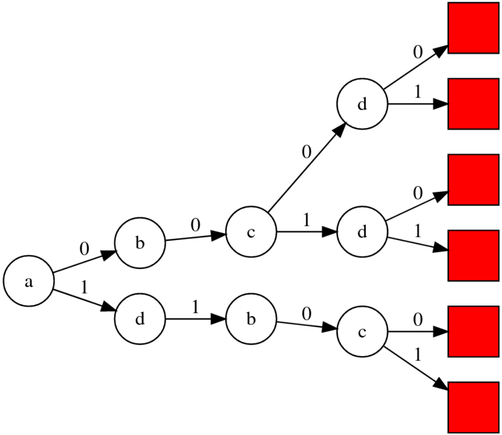
$c_1 : b \vee c$

$c_7 : b \vee \neg c$

$c_8 : \neg b \vee c$

Actions taken: Choose $b = 0$, see it is unsat., backtrack.

Tree:

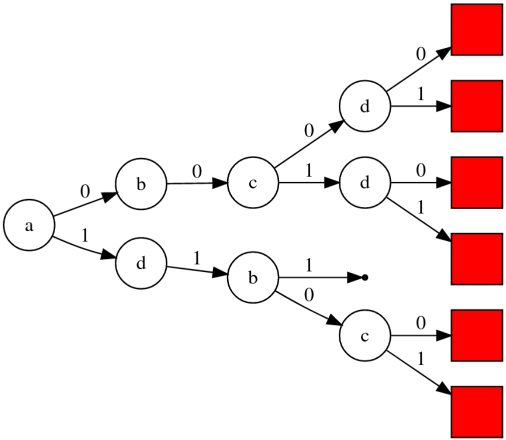


Classic SAT: DPLL Algorithm Example (6.1)

$c_8 : c$

Action taken: Choose $b = 1$.

Tree:

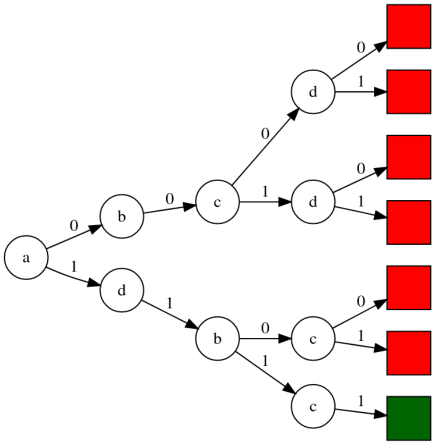


Classic SAT: DPLL Algorithm Example (6.1)

$c_8 : c$

Action taken: Unit prop. $c = 1$.

Tree:



Classic SAT: Conflict-Driven Clause Learning (CDCL)

Improvement on DPLL

1. Variable choosing, unit propagation, pure literal elim. as before.
2. Construct implication graph to find conflicts.
3. If conflict, find responsible assignments (predecessors of conflicting states).
4. Add negation of these assignments as new *conflict clause*.
5. *Non-chronological backjumping*: Jump back in tree as far as possible to a responsible assignment.

From CDCL to CDCL($\mathcal{T}$)

Early pruning: call $\mathcal{T}$-solver for partial assignments

Then: $\mathcal{T}$-solver(μ) should be incremental (reuse μ_1 results for $\mu_1 \cup \mu_2$ computation) and backtrack-capable (undo) $\implies$ Stack-based interface

$\mathcal{T}$-solver(μ) returns conflict set $\eta \subseteq \mu \implies$ conflict clauses

$\mathcal{T}$-propagation: $\mathcal{T}$-solver(μ) can give deduction clauses $\neg\mu' \models \eta$ where $\mu' \subseteq \mu$ causes assignment η to previously unassigned atom in φ.

Many more advanced techniques (pure-literal filtering, preprocessing atoms)

SPIN

Simple Promela Interpreter

- SPIN can be used for four main purposes:
  1. as a **simulator**, allowing for rapid prototyping with a random, guided, or interactive simulations
  2. as an exhaustive **verifier**, capable of proving the validity of user specified LTL properties
  3. as a **proof approximation system** that can validate large models with maximal coverage of the state space.
  4. as a driver for **swarm verification**, which can make optimal use of large numbers of available compute cores to leverage parallelism and search diversification techniques,

Simple Promela Interpreter (2)

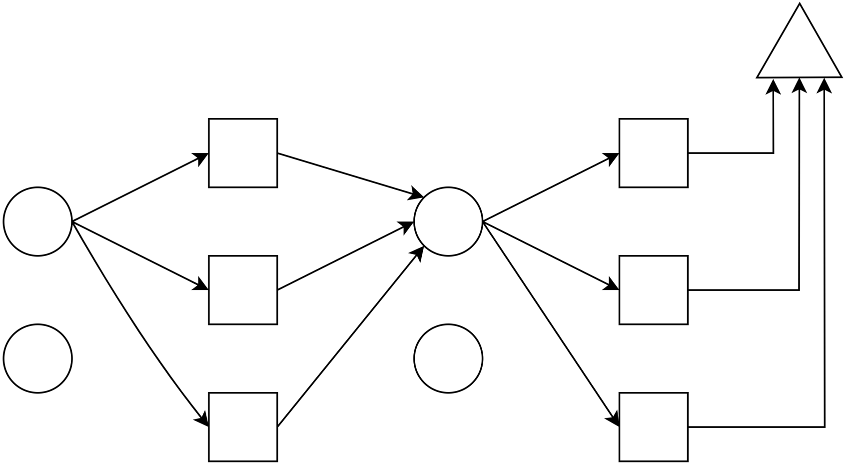
- Some of the many iteresting features of SPIN are:
  1. Spin targets efficient **software verification**, not hardware.
  2. Provides direct support for the use of **embedded C code** as part of model specifications.
  3. Works **on-the-fly**, which means that it avoids the preemptive construction of a global state graph.
  4. Spin supports both rendezvous and buffered FIFO **message passing** between processes.
  5. To optimize the verification runs, Spin exploits **partial order reduction** techniques, and (optionally) BDD-like storage techniques.

Process Meta Language

- Very similar to the C programming language
- Can specify **finite**-state systems.
  - Everything **must be bounded**
- A Promela model consists of the declaration of:

1. Types
2. Global variables
3. Channels
4. Processes
5. `init`

Case Study: Paxos



Algorithm

Algorithm 1 Paxos — Proposer p

Constants:
A, n, and f. {A is the set of acceptors. $n = |A|$ and $f = \lfloor (n-1)/2 \rfloor$.}

Init:
$crnd \leftarrow -1$ {Current round number}

on $\langle \text{PROPOSE}, val \rangle$
  $crnd \leftarrow \text{pickNextRound}(crnd)$
  $cval \leftarrow val$
  $P \leftarrow \emptyset$
  send $\langle \text{PREPARE}, crnd \rangle$ **to** A

on $\langle \text{PROMISE}, rnd, vrnd, vval \rangle$ with $rnd = crnd$ **from** acceptor a
  $P \leftarrow P \cup (vrnd, vval)$

on event $|P| \geq n - f$
  $j = \max\{vrnd : (vrnd, vval) \in P\}$
  if $j \geq 0$ **then**
    $V = \{vval : (j, vval) \in P\}$
    $cval \leftarrow \text{pick}(V)$ {Pick proposed value $vval$ with largest $vrnd$}
  send $\langle \text{ACCEPT}, crnd, cval \rangle$ **to** A

Algorithm 2 Paxos — Acceptor a

Constants:
L {Set of learners}

Init:
$rnd \leftarrow -1$
$vrnd \leftarrow -1$
$vval \leftarrow -1$

on $\langle \text{PREPARE}, prnd \rangle$ with $prnd > rnd$ **from** proposer p
  $rnd \leftarrow prnd$
  send $\langle \text{PROMISE}, rnd, vrnd, vval \rangle$ **to** proposer p

on $\langle \text{ACCEPT}, i, v \rangle$ with $i \geq rnd$ **from** proposer p
  $rnd \leftarrow i$
  $vrnd \leftarrow i$
  $vval \leftarrow v$
  send $\langle \text{LEARN}, i, v \rangle$ **to** L

Algorithm 3 Paxos — Learner l

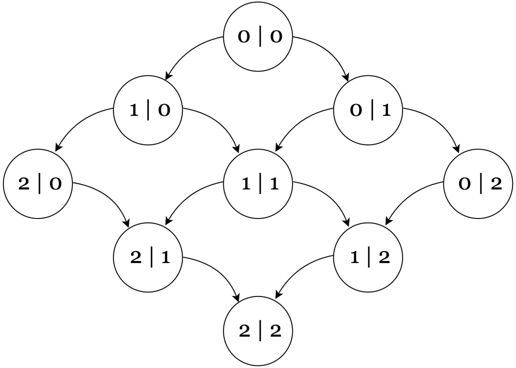
Init:
$V \leftarrow \emptyset$

on $\langle \text{LEARN}, (i, v) \rangle$ **from** acceptor a
  $V \leftarrow V \uplus (i, v)$

on event $\exists i, v : |\{(i, v) : (i, v) \in V\}| \geq n - f$
  v **is chosen**

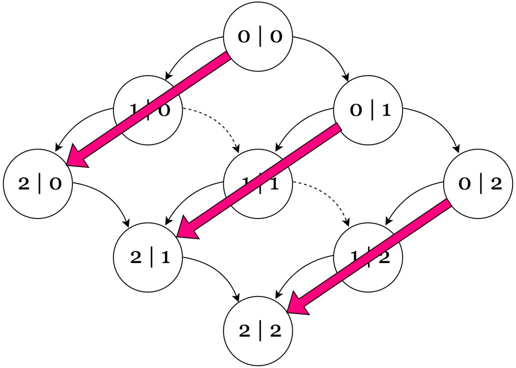
Atomic

```
active proctype P1() {
    a; b;
}
active proctype P2() {
    c; d;
}
```



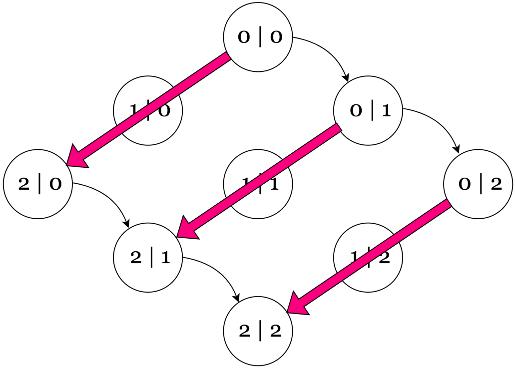
Atomic (2)

```
active proctype P1() {
    atomic { a; b; }
}
active proctype P2() {
    c; d;
}
```



Atomic (3)

```
active proctype P1() {
    d_step { a; b; }
}
active proctype P2() {
    c; d;
}
```



Fin.