Model Checker Aided Design of a Controller for a Wafer Scanner

Martijn Hendriks[1†] Barend van den Nieuwelaar[2⋆] Frits Vaandrager[1†]

[1] *University of Nijmegen, The Netherlands*

TU/e

[2] *Eindhoven University of Technology, The Netherlands*

[⋆] *ASML, Veldhoven, The Netherlands*

[†] *Supported by EC project IST-2001-35304 (AMETIST)*

Introduction

ASML builds wafer scanners

- Very complex lithographic machines used in the semiconductor manufacturing process
  - ▷ Machine is regarded as Task-Resource system (flexibility)
  - ▷ Scheduling in real-time (many things can go wrong)
  - ▷ Throughput is one of the main performance characteristics
  - ▷ Deadlock should be avoided at all costs

What is this case-study about?

- Material flow in Extreme Ultra Violet (EUV) machine
  - ▷ Compute a (least restrictive) deadlock avoidance policy
  - ▷ Compute schedules (optimal wrt throughput)

AMETIST

Class of problems considered by AMETIST: Scheduling / planning / resource allocation problems

Observation: There are many similar problems in different domains

- Scheduling production lines in factories
- Scheduling computer programs in real-time systems
- Scheduling instructions inside a processor
- Scheduling trains over limited quantities of railroad track

Many approaches (re)-invented in each domain

A unifying framework would be nice. . .

AMETIST (2)

The AMETIST approach:

- Model as dynamical system with *state space* and well-defined *dynamics*: model generates behavior (the semantics)
- Design activities (verification, synthesis etc) explore and modify system structure so that behavior is correct, optimal, etc
- Do not let modeling suffer from tools

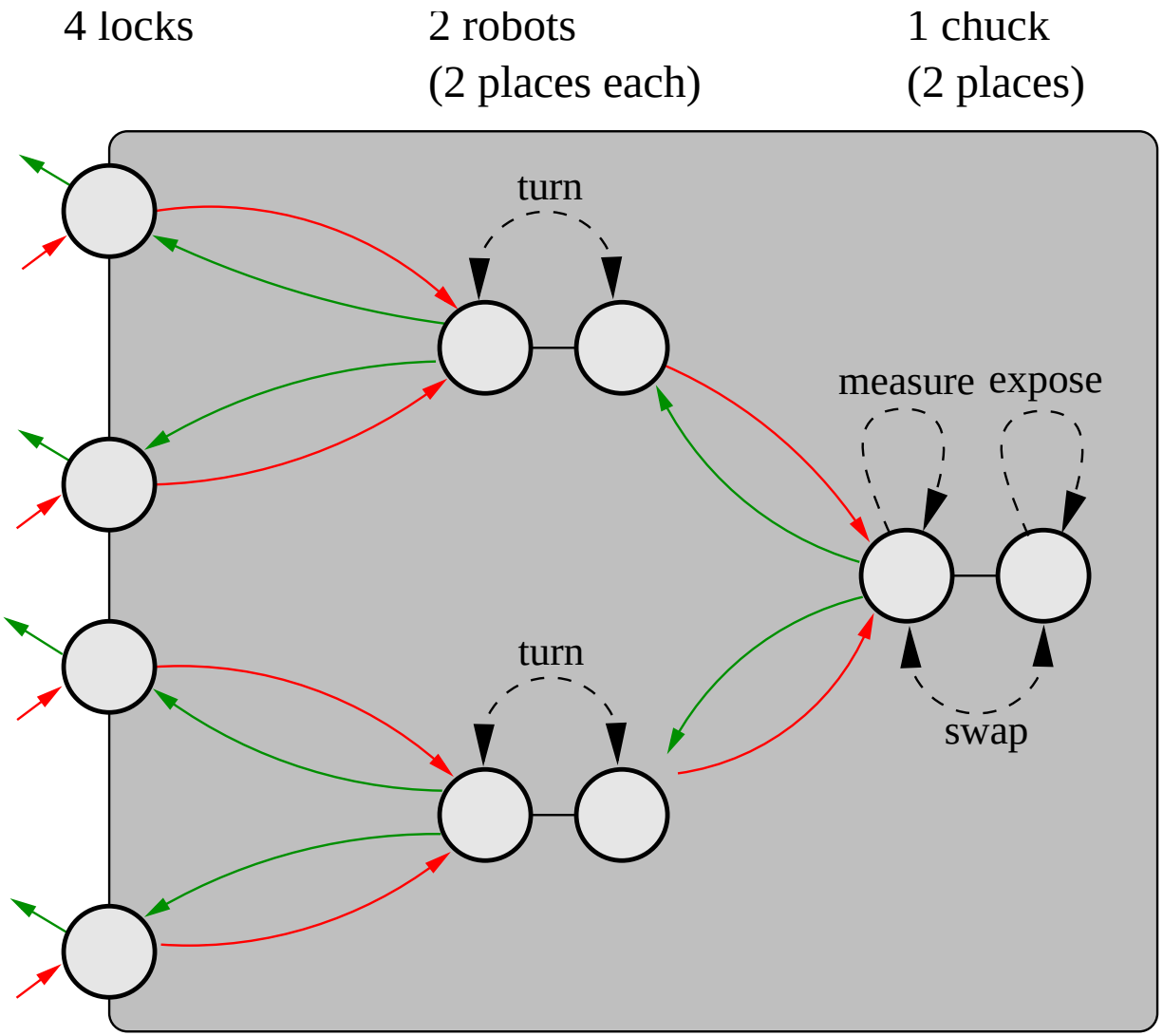
Timed automaton model as mathematical carrier

AMETIST mission:

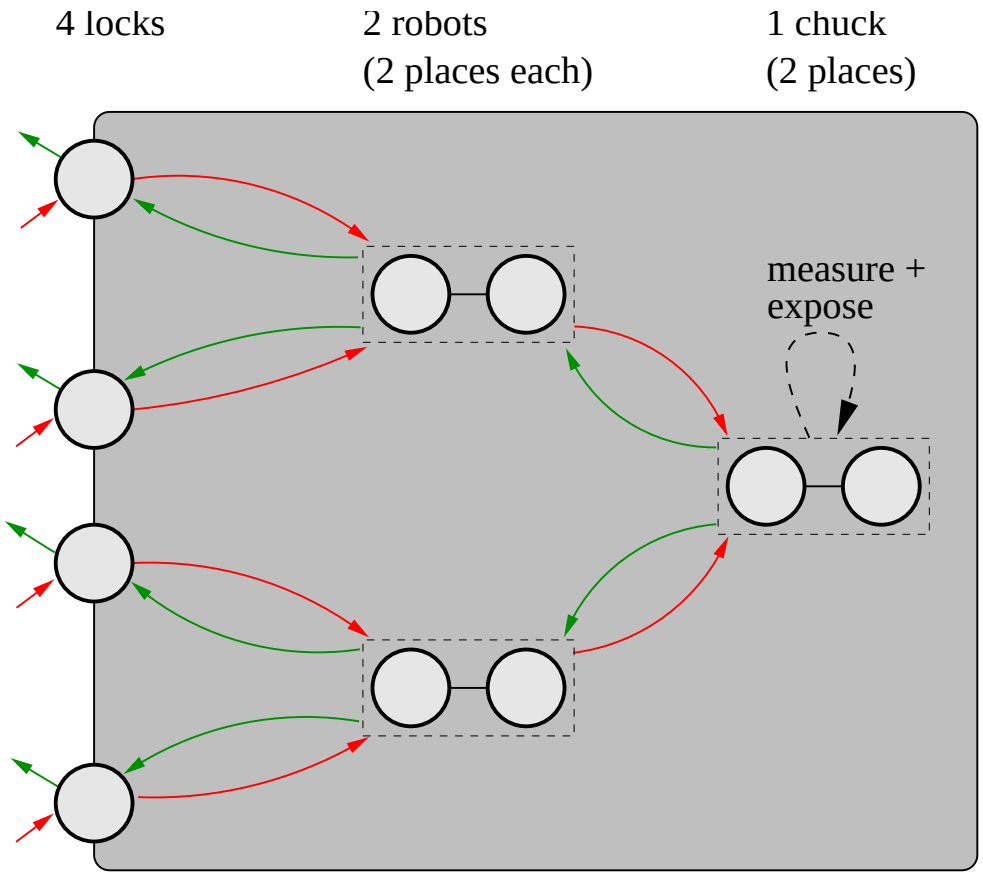
- ▷ Improve TA model checking tools
- ▷ Investigate the applicability of TA tools
- ▷ Link to dedicated tools when appropriate

Contents

Material flow in EUV machine

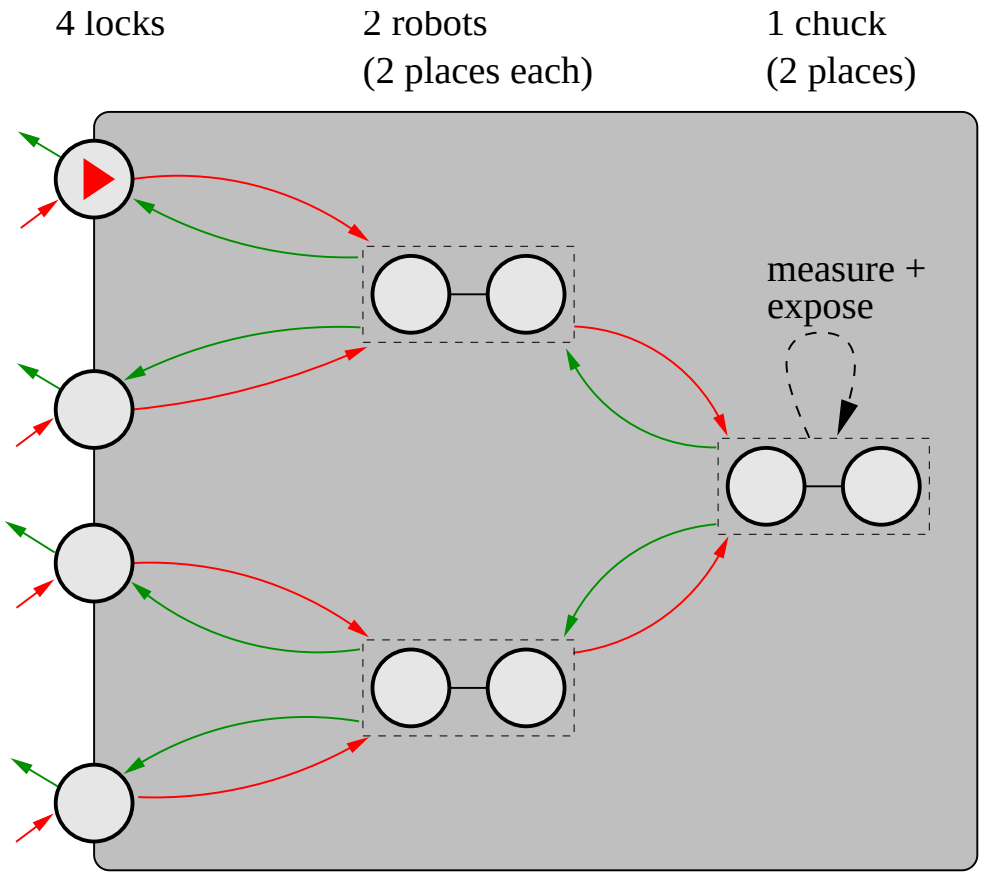


Material flow in EUV machine



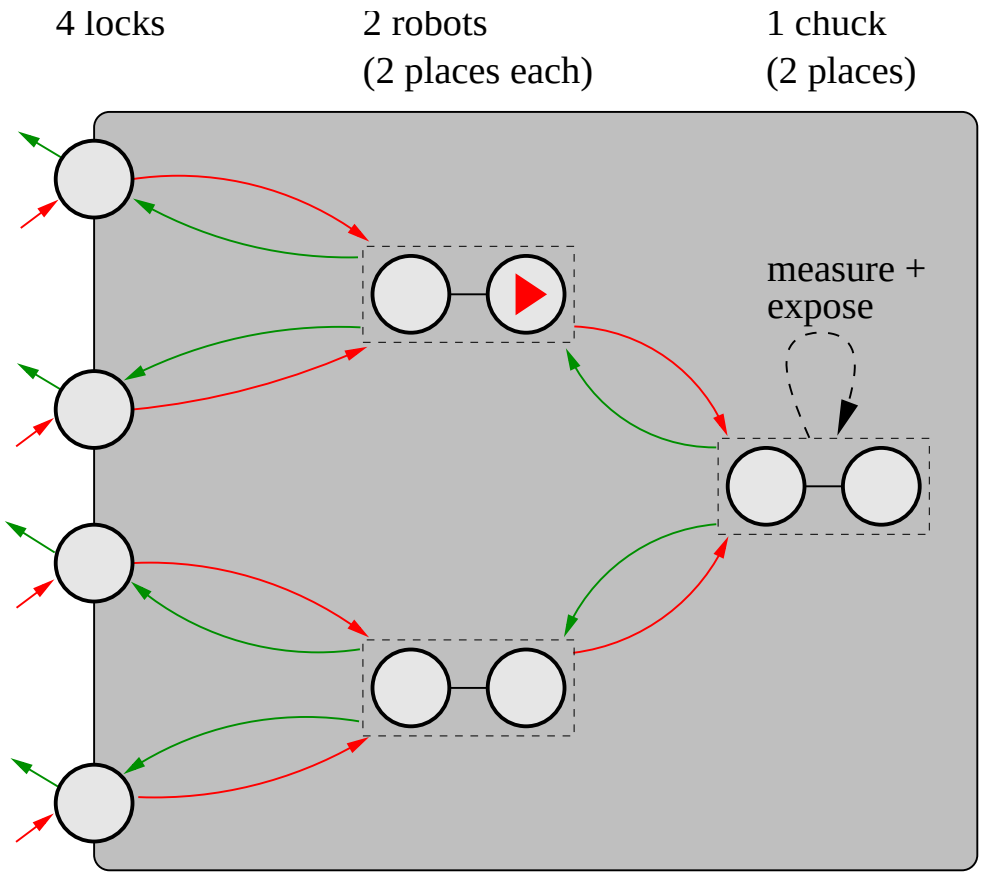
- Atomic step: entry / exit / move / measure-expose of 1 wafer
- Abstract from turning robots & chuck swap
- Acyclic 1-color material paths (to prevent "livelocks")
- A place is empty between two material instances

Material flow in EUV machine



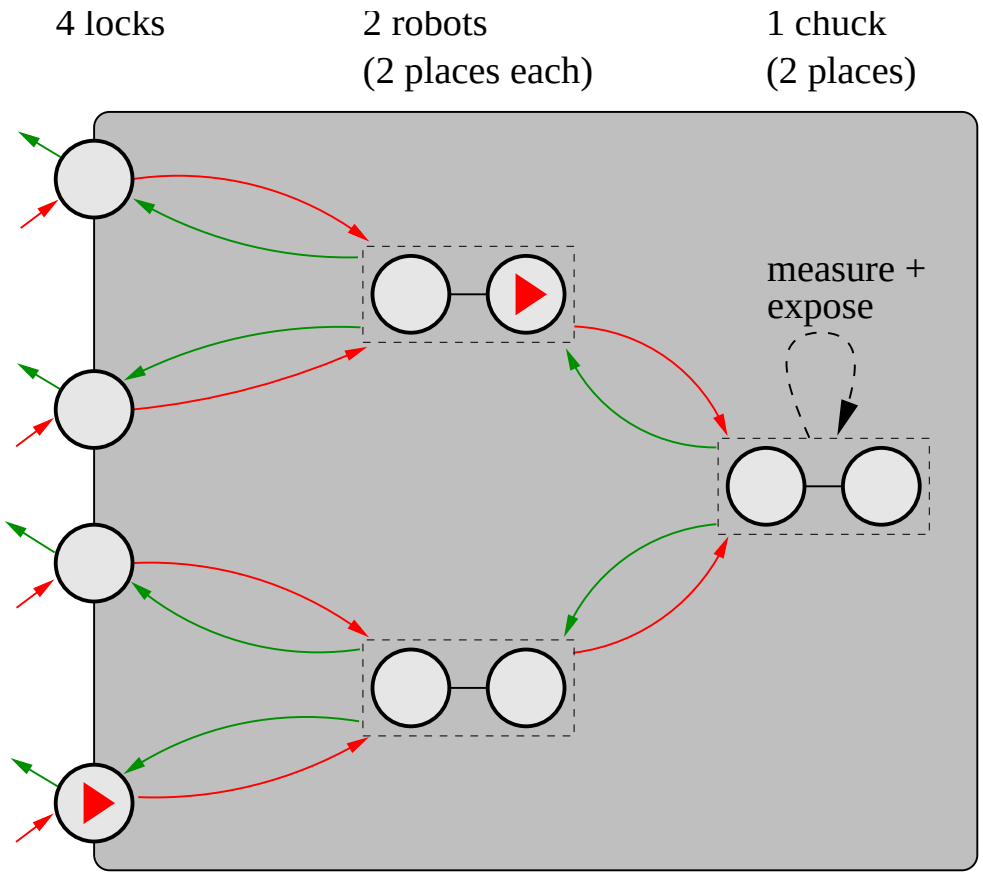
- Atomic step: entry / exit / move / measure-expose of 1 wafer
- Abstract from turning robots & chuck swap
- Acyclic 1-color material paths (to prevent "livelocks")
- A place is empty between two material instances

Material flow in EUV machine



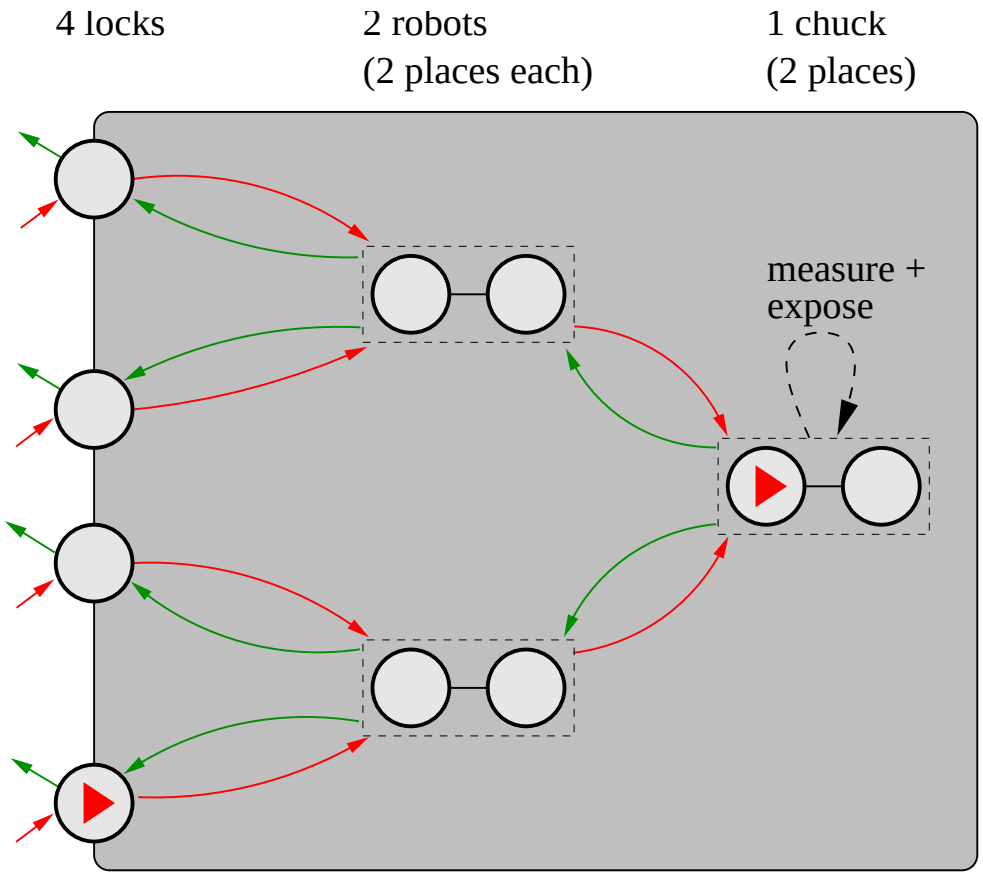
- Atomic step: entry / exit / move / measure-expose of 1 wafer
- Abstract from turning robots & chuck swap
- Acyclic 1-color material paths (to prevent "livelocks")
- A place is empty between two material instances

Material flow in EUV machine



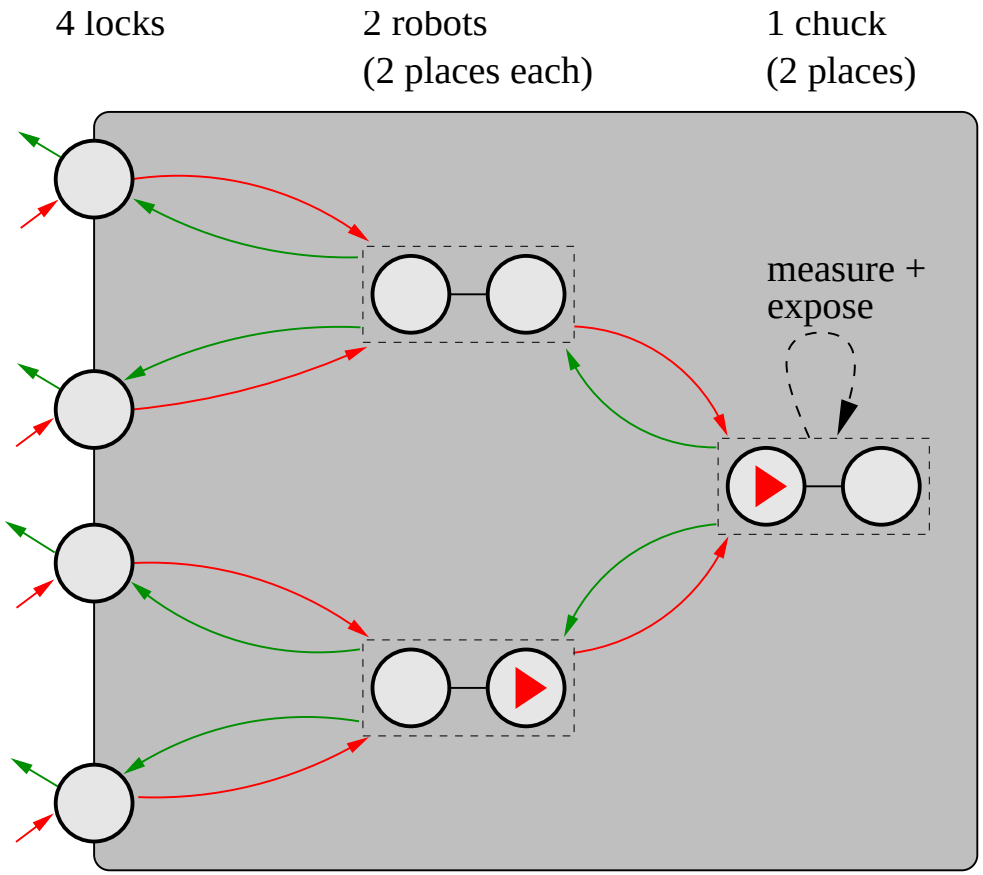
- Atomic step: entry / exit / move / measure-expose of 1 wafer
- Abstract from turning robots & chuck swap
- Acyclic 1-color material paths (to prevent "livelocks")
- A place is empty between two material instances

Material flow in EUV machine



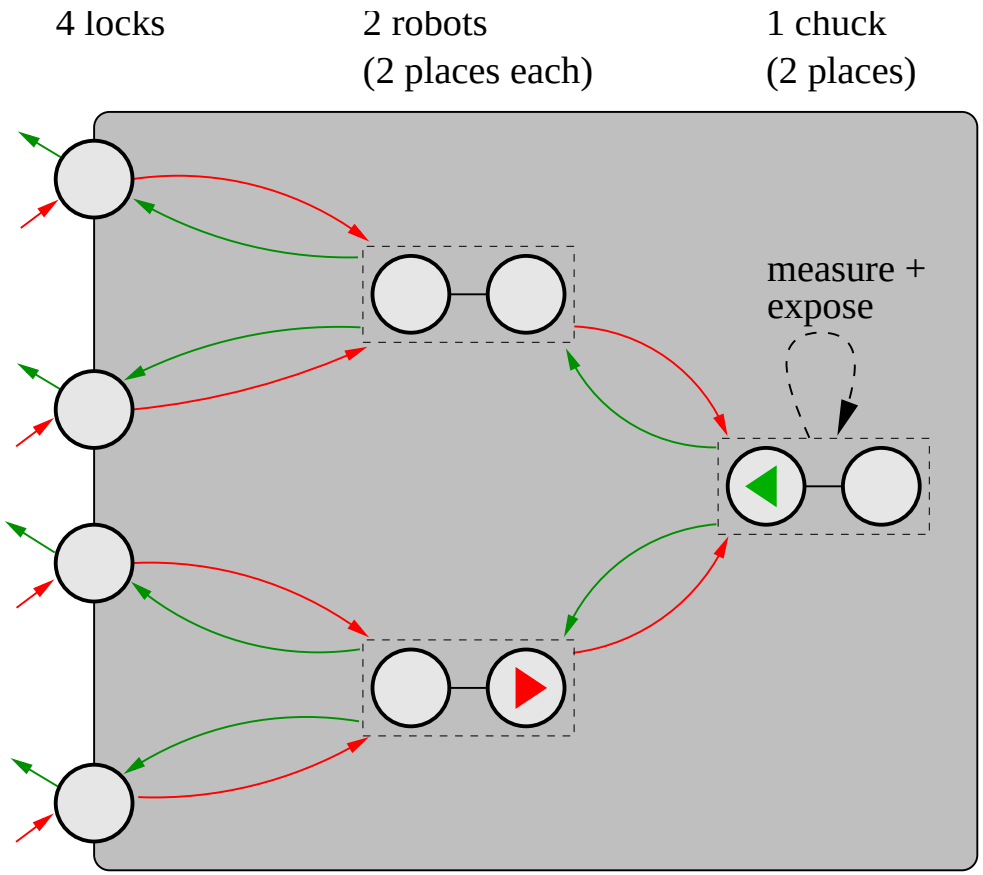
- Atomic step: entry / exit / move / measure-expose of 1 wafer
- Abstract from turning robots & chuck swap
- Acyclic 1-color material paths (to prevent "livelocks")
- A place is empty between two material instances

Material flow in EUV machine



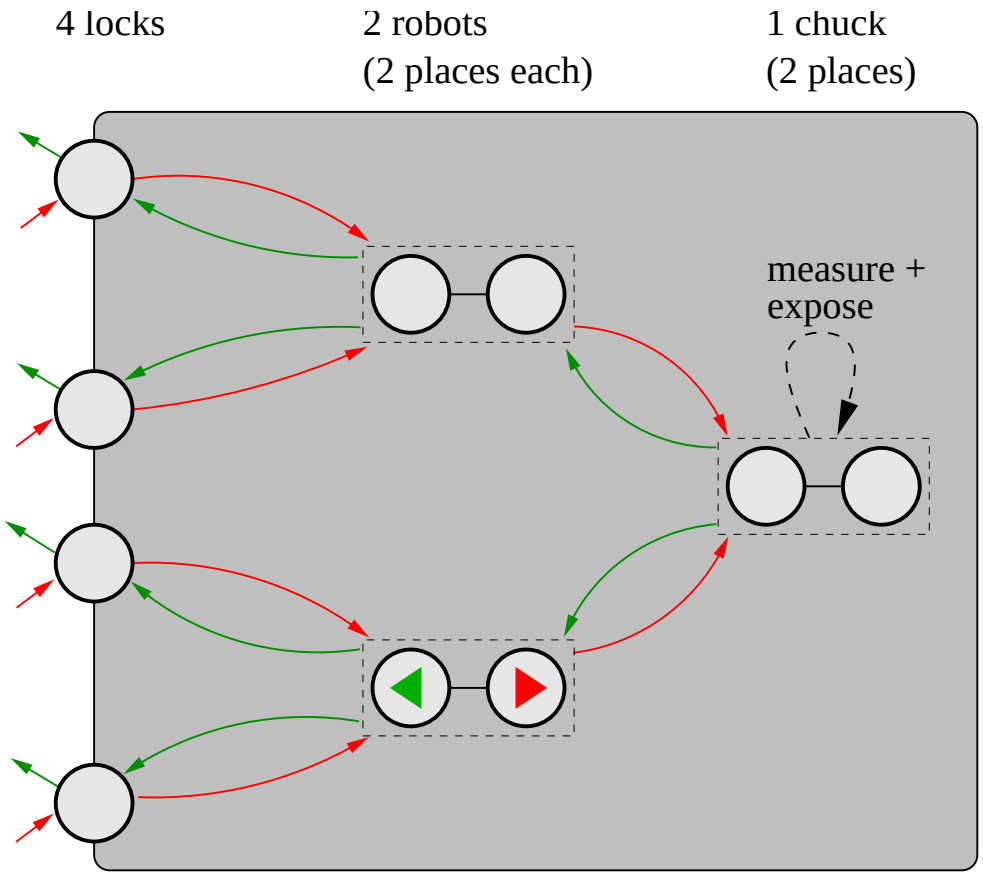
- Atomic step: entry / exit / move / measure-expose of 1 wafer
- Abstract from turning robots & chuck swap
- Acyclic 1-color material paths (to prevent "livelocks")
- A place is empty between two material instances

Material flow in EUV machine



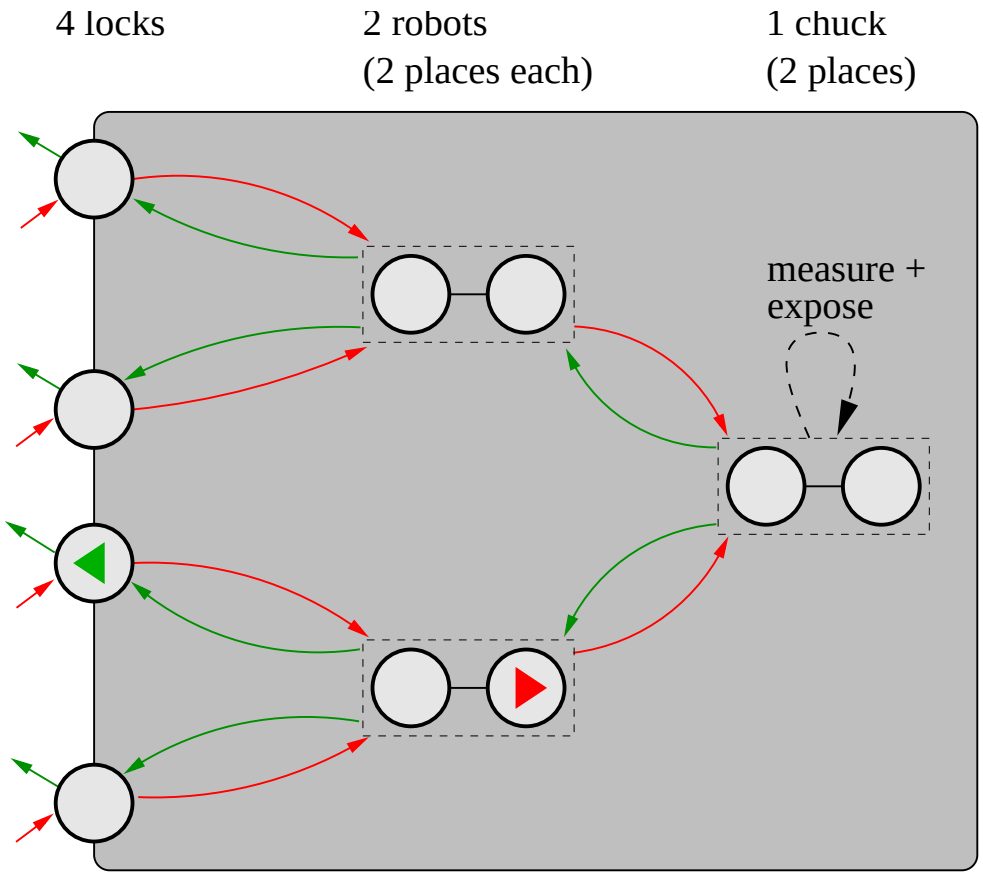
- Atomic step: entry / exit / move / measure-expose of 1 wafer
- Abstract from turning robots & chuck swap
- Acyclic 1-color material paths (to prevent "livelocks")
- A place is empty between two material instances

Material flow in EUV machine



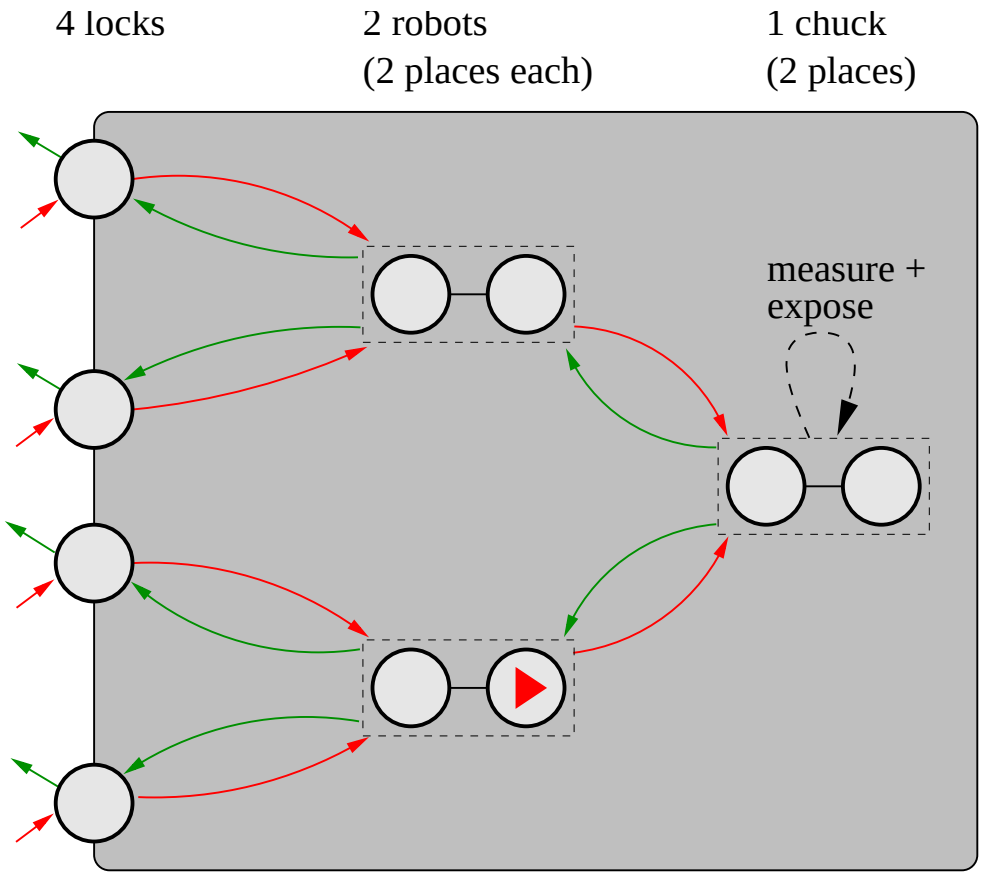
- Atomic step: entry / exit / move / measure-expose of 1 wafer
- Abstract from turning robots & chuck swap
- Acyclic 1-color material paths (to prevent "livelocks")
- A place is empty between two material instances

Material flow in EUV machine



- Atomic step: entry / exit / move / measure-expose of 1 wafer
- Abstract from turning robots & chuck swap
- Acyclic 1-color material paths (to prevent "livelocks")
- A place is empty between two material instances

Material flow in EUV machine

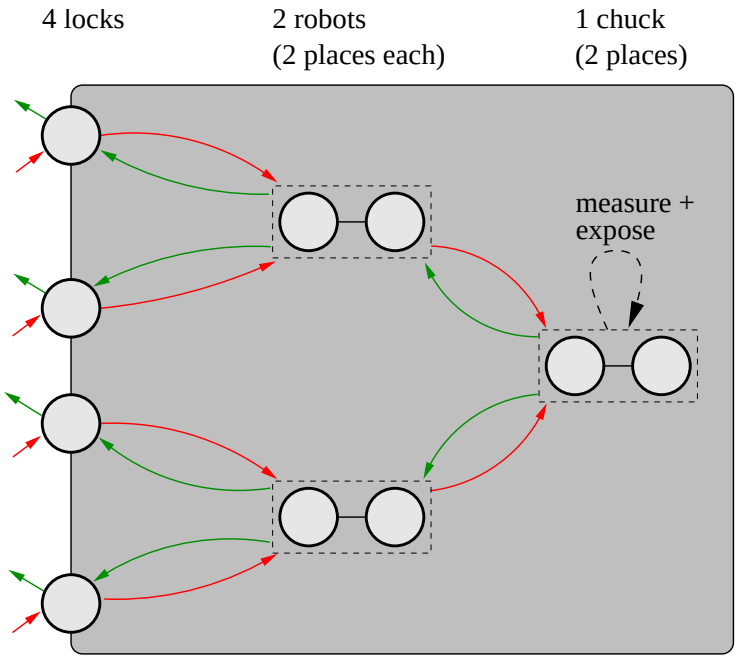


- Atomic step: entry / exit / move / measure-expose of 1 wafer
- Abstract from turning robots & chuck swap
- Acyclic 1-color material paths (to prevent "livelocks")
- A place is empty between two material instances

SMV model

Straightforward modeling:

- Every place is modeled by a state variable which can be empty (e), red (r), or green (g)
- Every pair of arrows is modeled by an asynchronous process



SMV model (2)

```
module main ()
{
  -- the places in the machine:
  l : array 03 of {e,r,g};
  c : array 01 of {e,r,g};
  rb: array 01 of
      array 01 of {e,r,g};

  -- initialization:
  for (i=0; i<4; i=i+1)
    init(l[i]):=e;
  for (i=0; i<2; i=i+1)
    for (j=0; j<2; j=j+1)
      init(rb[i][j]):=e;
  for (i=0; i<2; i=i+1)
    init(c[i]):=e;

  -- system dynamics:
  for (i=0; i<4; i=i+1)
    t2l[i]: process entry_exit(l[i]);

  for (i=0; i<4; i=i+1)
    for (j=0; j<2; j=j+1)
      l2r[i][j]: process move(l[i],rb[(i<2?0:1)][j]);

  for (i=0; i<2; i=i+1)
    for (j=0; j<2; j=j+1)
      for (k=0; k<2; k=k+1)
        r2c[i][j][k]: process move(rb[i][j],c[k]);

  for (i=0; i<2; i=i+1)
    exp[i]: process expose(c[i]);
}
```

```
module entry_exit (p)
{
  if (p=e)
    next(p):=r;
  else if (p=g)
    next(p):=e;
}

module move (lft,rgt)
{
  if (lft=r && rgt=e)
  {
    next(lft):=e;
    next(rgt):=r;
  }
  else if (lft=e && rgt=g)
  {
    next(lft):=g;
    next(rgt):=e;
  }
}

module expose (p)
{
  if (p=r)
    next(p):=g;
}
```

Avoiding Deadlock

3 ways of handling deadlock:

- Deadlock prevention: restrict system such that deadlock is a priori impossible
- Deadlock detection: detect and resolve deadlocks at runtime
- Deadlock avoidance: dynamically choose control actions to avoid deadlock

Avoiding Deadlock

3 ways of handling deadlock:

- Deadlock prevention: restrict system such that deadlock is a priori impossible
- Deadlock detection: detect and resolve deadlocks at runtime
- Deadlock avoidance: dynamically choose control actions to avoid deadlock

Deadlock avoidance:

- Keep the system in a set of *safe states* (Dijkstra, 1965)
- Questions:
  - ▷ What is deadlock and what are safe states?
  - ▷ How to express deadlock and safety in CTL?
  - ▷ How to characterize the set of safe states?

Avoiding Deadlock (2)

Conditions for deadlock:

(*Operating systems – internals and design principles*, Stallings)

- **Mutual exclusion**: only one process may use a resource at a time
- **Hold and wait**: a process may hold allocated resources while awaiting assignment of others
- **No preemption**: no resource can be forcibly removed from a process that is holding it
- **Circular wait**: a closed chain of processes exists such that each process holds at least one resource needed by the next resource in the chain

First three items hold in the model; circular wait must be formalized (there is a *choice* of which resources to use)

Avoiding Deadlock (3)

Example:

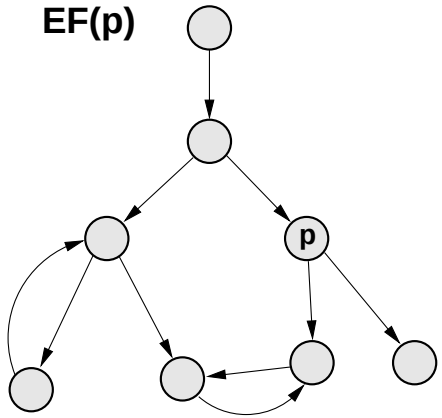
Formalization of circular wait for a state s

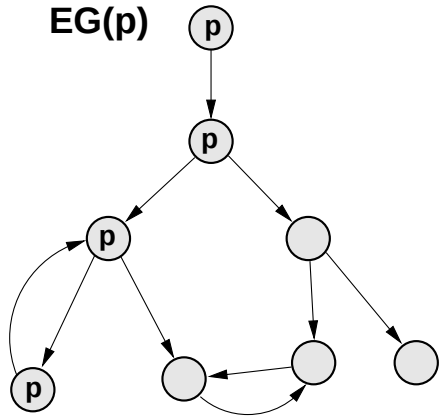
$$\exists_{Q \subseteq P}\ Q \neq \emptyset\ \wedge\ \forall_{q \in Q}\ (s(q) \neq e \wedge \emptyset \neq needs^s(q) \subseteq Q)$$

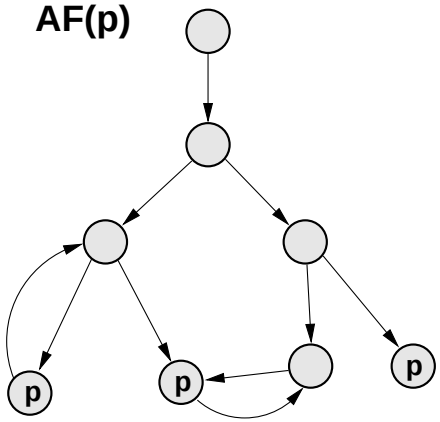
where P is the set of places, and $needs^s(q)$ gives the set of places of which *one* is needed by the wafer that currently is at place q

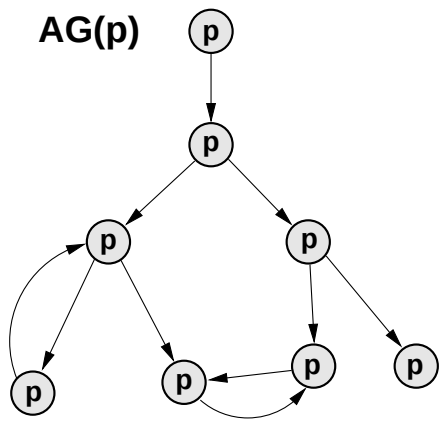
(CTL interlude)

SMV builds a transition system over which it interprets CTL









Avoiding Deadlock (4)

How to encode a circular wait situation in CTL?

- Basic idea: the wafers in a circular wait can never move again; they are *jammed*
- Observe: if in our model a transitions $s \longrightarrow s'$ moves a wafer from place p to p', then p is empty in state s'

Avoiding Deadlock (4)

How to encode a circular wait situation in CTL?

- Basic idea: the wafers in a circular wait can never move again; they are *jammed*
- Observe: if in our model a transitions $s \longrightarrow s'$ moves a wafer from place p to p', then p is empty in state s'

$$\mathbf{jammed} \equiv \bigvee_{p \in P} \mathbf{AG}(p \text{ is not empty})$$

Avoiding Deadlock (4)

How to encode a circular wait situation in CTL?

- Basic idea: the wafers in a circular wait can never move again; they are *jammed*
- Observe: if in our model a transitions $s \longrightarrow s'$ moves a wafer from place p to p', then p is empty in state s'

$$\mathbf{jammed} \equiv \bigvee_{p \in P} \mathbf{AG}(p \text{ is not empty})$$

Proposition: s has a circular wait if and only if s is jammed

lemma 1 Circular wait property is stable

lemma 2 If position p is jammed in state s and $p' \in needs^s(p)$, then position p' is also jammed in state s

Avoiding Deadlock (5)

What are safe states?

- A state is *safe* iff "all processes can be run to completion" (*Banker's algorithm*, Dijkstra, 1965)
- In our case: the wafers are the processes, and they are "run to completion" when they exit the machine

Avoiding Deadlock (5)

What are safe states?

- A state is *safe* iff "all processes can be run to completion" (*Banker's algorithm*, Dijkstra, 1965)
- In our case: the wafers are the processes, and they are "run to completion" when they exit the machine

$$\mathbf{safe} \equiv \mathbf{EF}\left(\bigwedge_{p \in P} (p \text{ is empty})\right)$$

Avoiding Deadlock (5)

What are safe states?

- A state is *safe* iff "all processes can be run to completion" (*Banker's algorithm*, Dijkstra, 1965)
- In our case: the wafers are the processes, and they are "run to completion" when they exit the machine

$$\textbf{safe} \equiv \mathbf{EF}\left(\bigwedge_{p \in P} (p \text{ is empty})\right)$$

Note:

- **jammed** $\Longrightarrow$ $\neg$**safe**
- but in general NOT: $\neg$**safe** $\Longrightarrow$ **jammed**

Avoiding Deadlock (6)

What is the connection between safe and jammed states?

- We want to show that safe states really are safe, ie, it is always possible to avoid deadlock (= circular wait = jammed)
- Furthermore, the set of safe states is the largest set from which deadlock can always be avoided

Avoiding Deadlock (6)

What is the connection between safe and jammed states?

- We want to show that safe states really are safe, ie, it is always possible to avoid deadlock (= circular wait = jammed)
- Furthermore, the set of safe states is the largest set from which deadlock can always be avoided

$$s_{\text{init}} \models \mathbf{AG}\,(\textbf{safe} \iff \textbf{EG}(\neg\textbf{jammed}))$$

Avoiding Deadlock (6)

What is the connection between safe and jammed states?

- We want to show that safe states really are safe, ie, it is always possible to avoid deadlock (= circular wait = jammed)
- Furthermore, the set of safe states is the largest set from which deadlock can always be avoided

$$s_{\text{init}} \models \mathbf{AG}\,(\textbf{safe} \iff \textbf{EG}(\neg\textbf{jammed}))$$

Least restrictive deadlock avoidance policy for EUV machine:

- Keep it within the set of safe states!

Avoiding Deadlock (7)

Characterizing the set of safe states:

set $C = true$

while $s_{\text{init}} \not\models \mathbf{AG}(\mathbf{safe} \iff C)$ do:

Update C to exclude counterexample (involves thinking)

This case: 4 iterations to get 4 unsafe situations (mod symmetry)

Avoiding Deadlock (7)

Characterizing the set of safe states:

set $C = true$

while $s_{\text{init}} \not\models \mathbf{AG}(\mathbf{safe} \iff C)$ do:

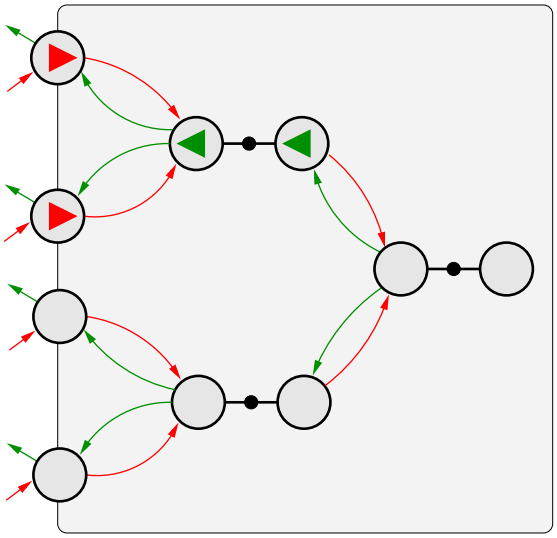
Update C to exclude counterexample (involves thinking)

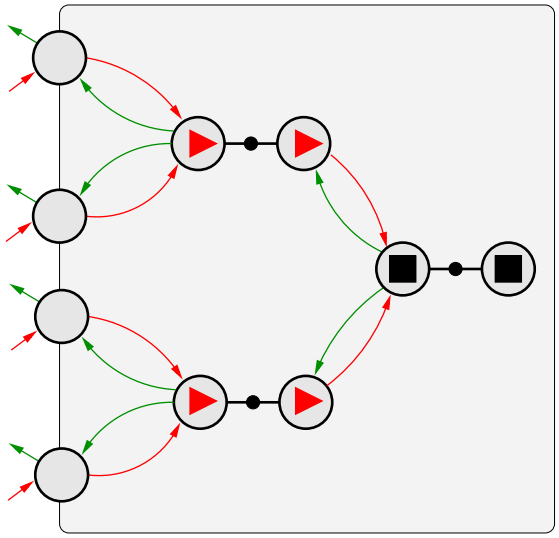
This case: 4 iterations to get 4 unsafe situations (mod symmetry)

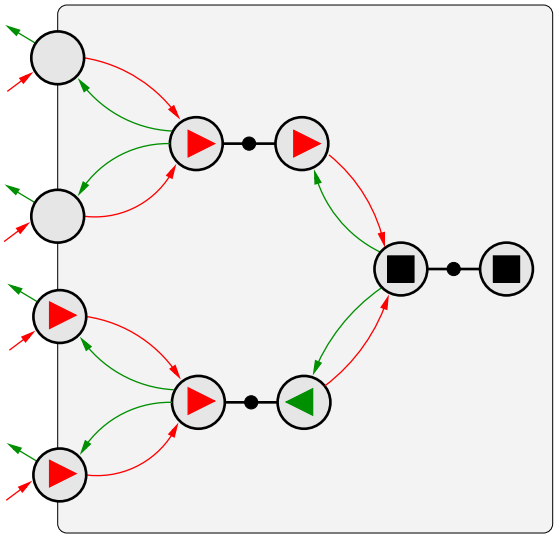
Note:

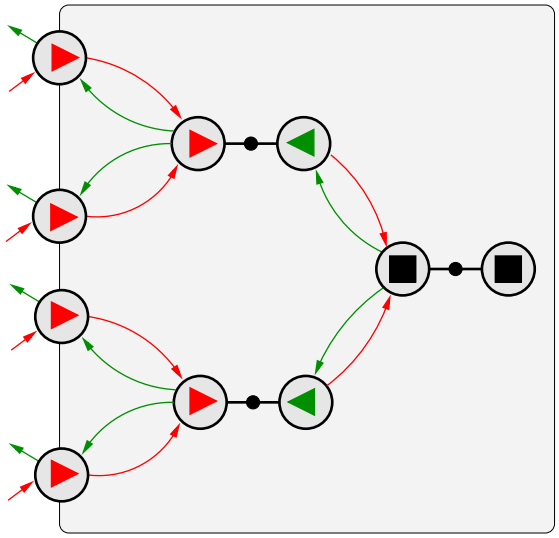
- Creative step is not needed: SMV internally builds a BDD representation of the set of safe states if you ask whether $s_{\text{init}} \models \mathbf{safe}$
- However, the iterative process gives nice pictures (how to draw a picture from a BDD?)

Avoiding Deadlock (8)









Avoiding Deadlock (9)

Predicate C that exactly characterizes the set of safe states:

```
~( (l[0]=r & l[1]=r & rb[0][0]=g & rb[0][1]=g)
   |
   (l[2]=r & l[3]=r & rb[1][0]=g & rb[1][1]=g)
   |
   (~c[0]=e & ~c[1]=e & rb[0][0]=r & rb[0][1]=r & rb[1][0]=r & rb[1][1]=r)
   |
   (~c[0]=e & ~c[1]=e & rb[0][0]=r & rb[0][1]=r &
   ((rb[1][0]=r & rb[1][1]=g) | (rb[1][0]=g & rb[1][1]=r)) & l[2]=r & l[3]=r)
   |
   (~c[0]=e & ~c[1]=e & rb[1][0]=r & rb[1][1]=r &
   ((rb[0][0]=r & rb[0][1]=g) | (rb[0][0]=g & rb[0][1]=r)) & l[0]=r & l[1]=r)
   |
   (~c[0]=e & ~c[1]=e & ((rb[0][0]=r & rb[0][1]=g) | (rb[0][0]=g & rb[0][1]=r)) &
   ((rb[1][0]=r & rb[1][1]=g) | (rb[1][0]=g & rb[1][1]=r)) &
   l[0]=r & l[1]=r & l[2]=r & l[3]=r)
)
```

SMV demo

Contents

Uppaal model

Adaptation of the SMV model:

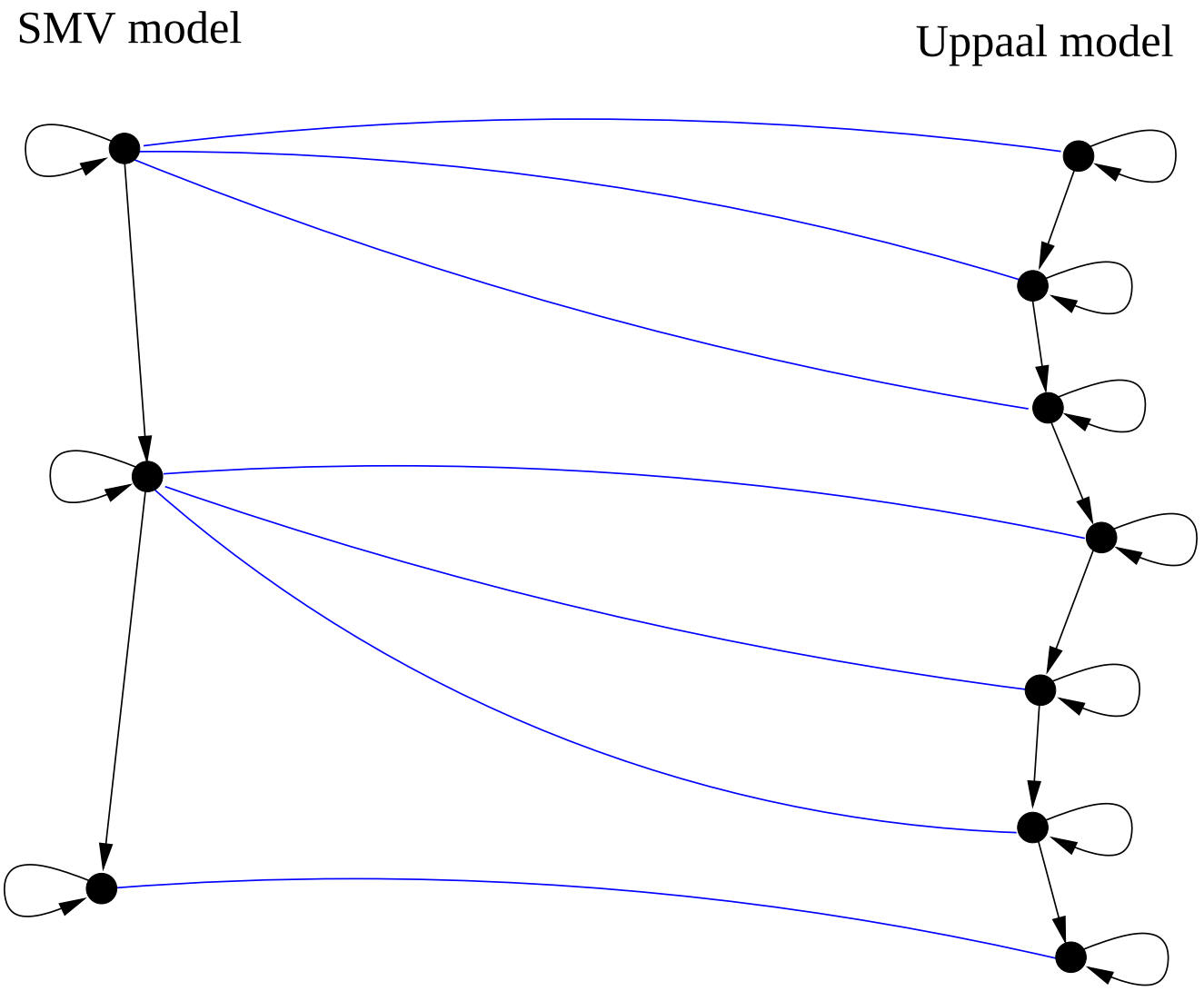
- Refine the processes that modify wafers (TrackRobot, 2 Internal Robot & Chuck processes) to add timing
- Additional processes to model constraints (4 Lock processes)
- Additional Observer process for throughput optimization

Relation with the SMV model:

- There is a *stuttering bisimulation* between the Uppaal model and the SMV model Thus, CTL\X formulas are preserved (Browne, Clarke & Grümberg, 1988)

Uppaal model (2)

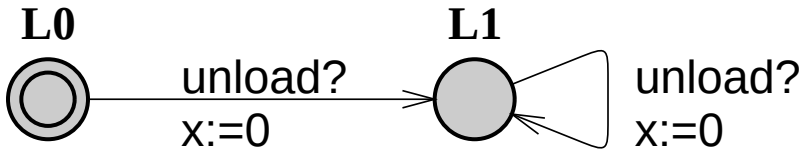
Stuttering bisimulation R



Uppaal model (3)

Throughput optimization

- Observer process (has a local clock x):



- Ask Uppaal whether

$$s_{\text{init}} \models \mathbf{EG} \left(\begin{array}{c} Observer.L0 \Longrightarrow Observer.x \leq H \\ \wedge \\ Observer.L1 \Longrightarrow Observer.x \leq S \end{array} \right)$$

Uppaal demo

Adding heuristics

The state space is too large

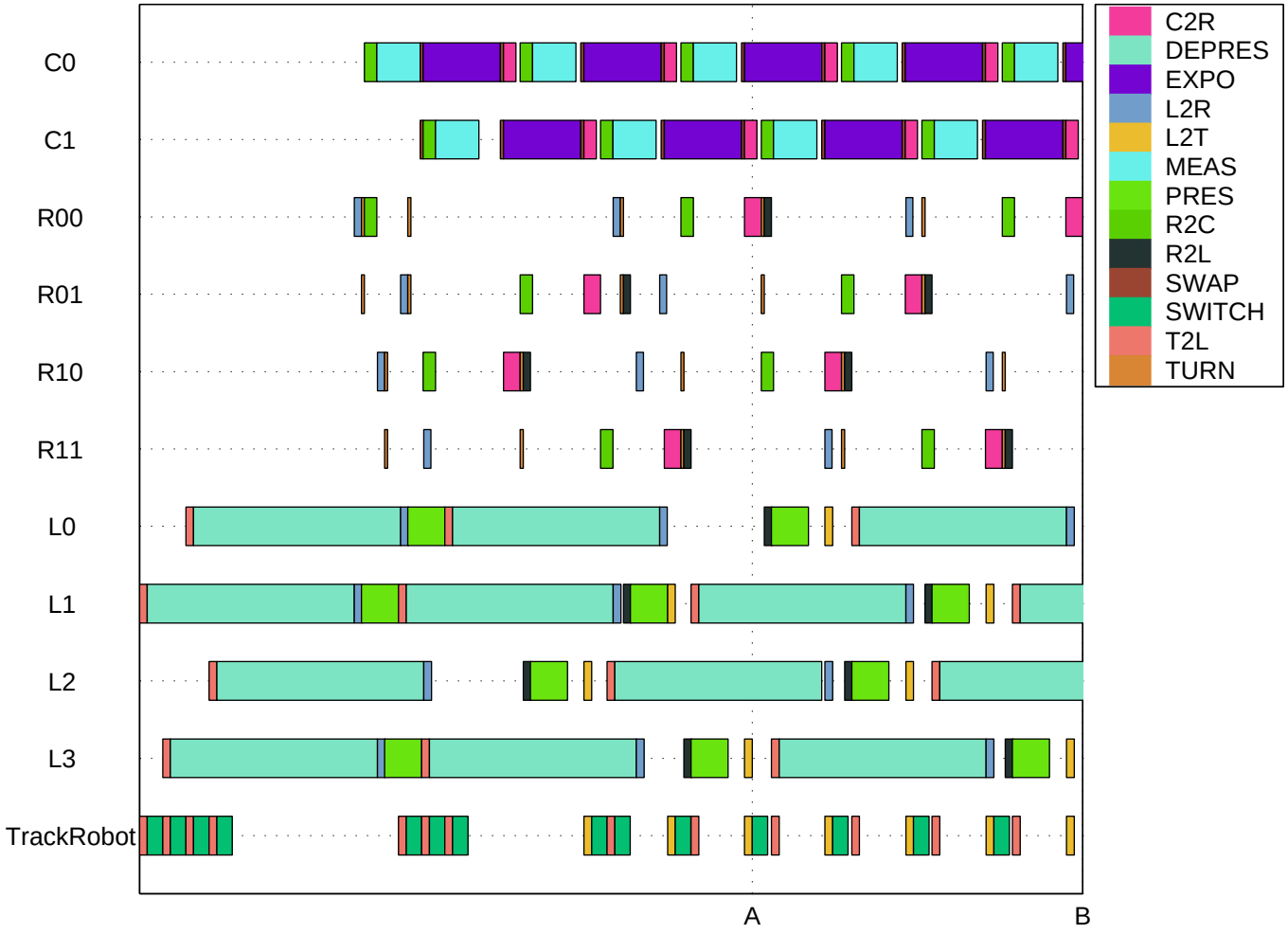
- Locks can depressurize or pressurize (almost) any time
- Internal robots can turn (almost) any time
- Chuck can swap (almost) any time
- Large differences in time scale: 670 (lock depres) vs 10 (turn)

Solution: add heuristics

- Avoid unsafe material configurations
- Avoid useless transitions (turns, swaps, etc)
- Make some transitions greedy/urgent

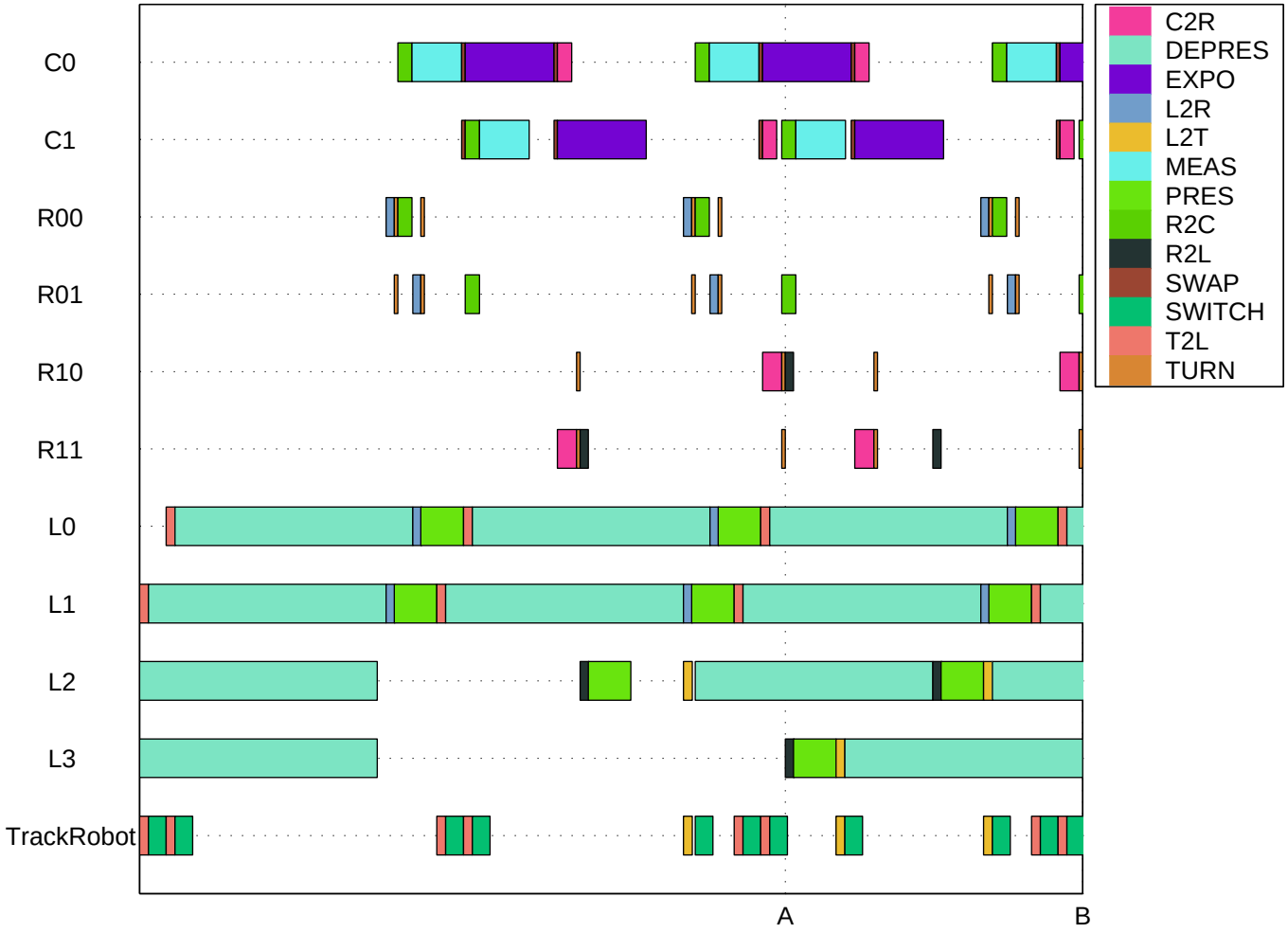
Adding heuristics (2)

Optimal schedule can be found easily:



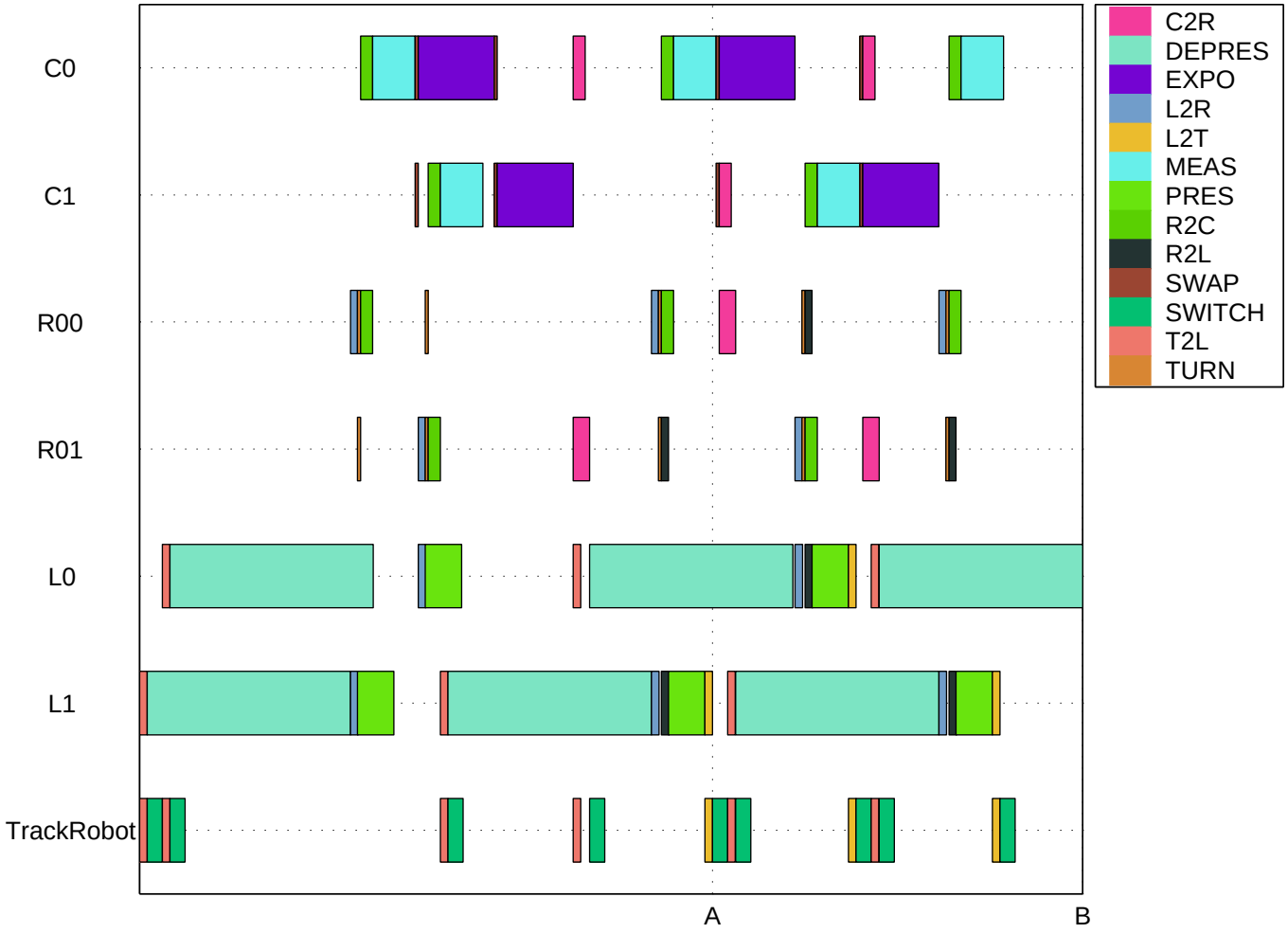
Adding heuristics (3)

Optimal schedule for no crossing wafer paths



Adding heuristics (4)

Some schedule for 2 locks and 1 internal robot:



Contents

Conclusions

Short and *exact* characterization of safe states (either by iterative process or by extracting a BDD from SMV)

Synthesis of a schedule that optimizes throughput; analysis of an alternative configuration and control policy

It took us approx 2 weeks to build the models and to obtain our results

Our work confirms once more that formal modeling and analysis may help to improve the design process; our work is referred to in a patent filed by ASML

Scalability?