Automatic State Reaching for Debugging Reactive Programs

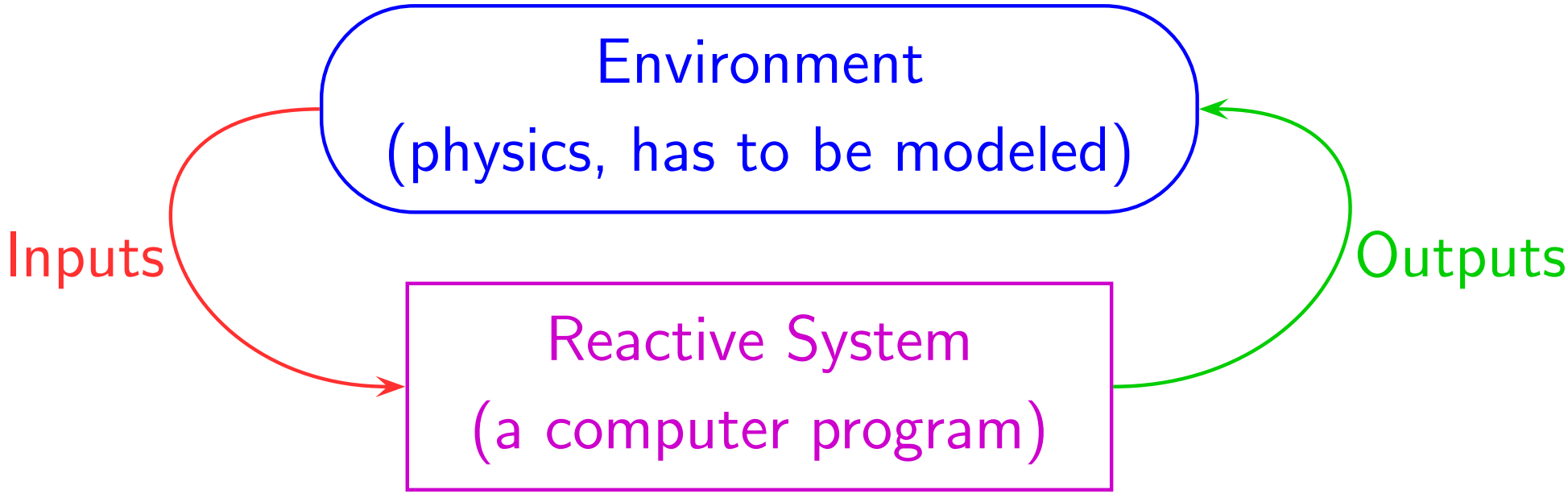
Fabien Gaucher, Erwan Jahier, Bertrand Jeannet &
Florence Maraninchi

Reactive Systems in the Synchronous Approach

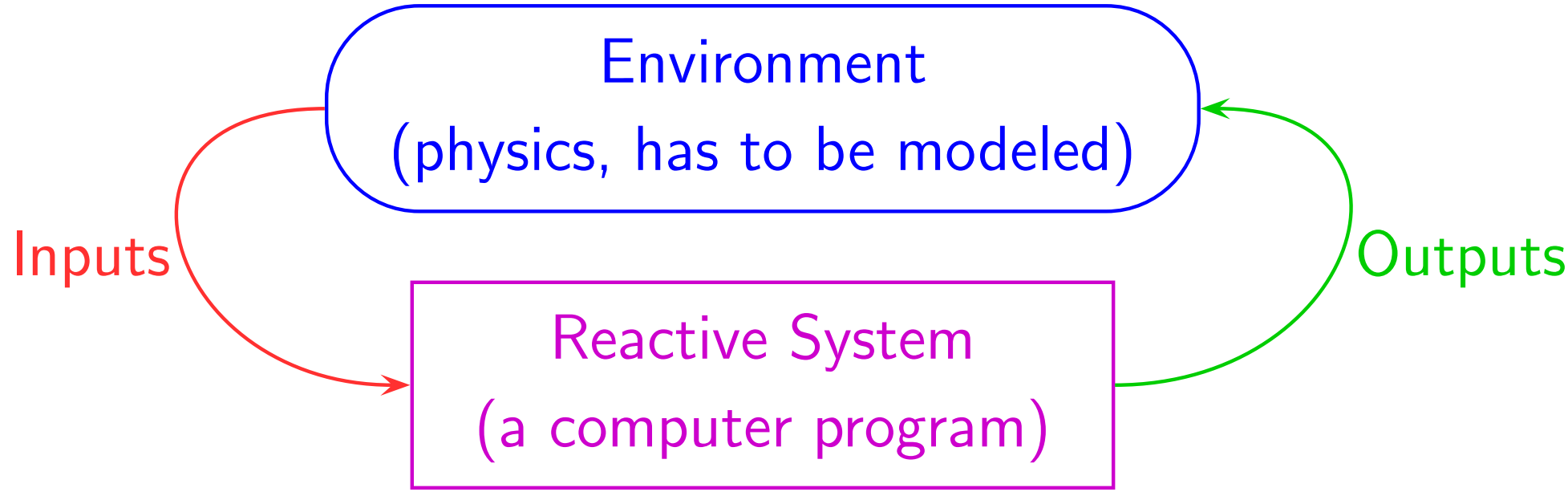
Environment
(physics, has to be modeled)

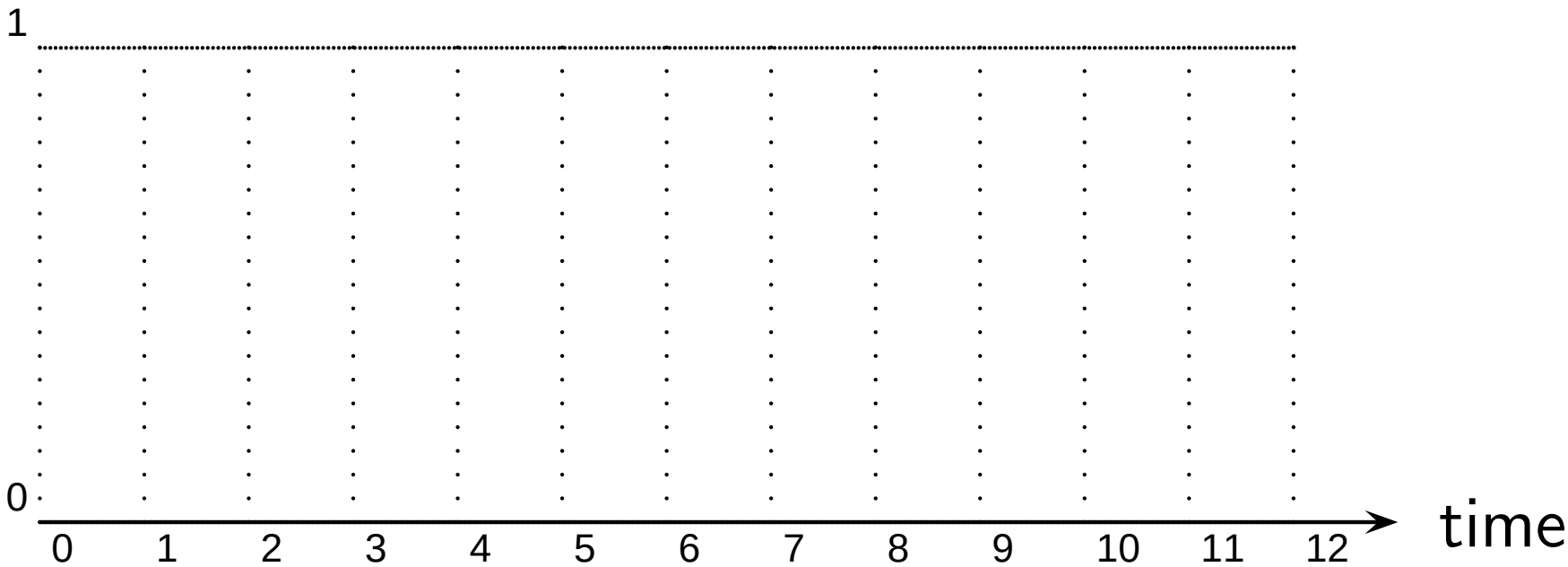
Reactive System
(a computer program)

Reactive Systems in the Synchronous Approach

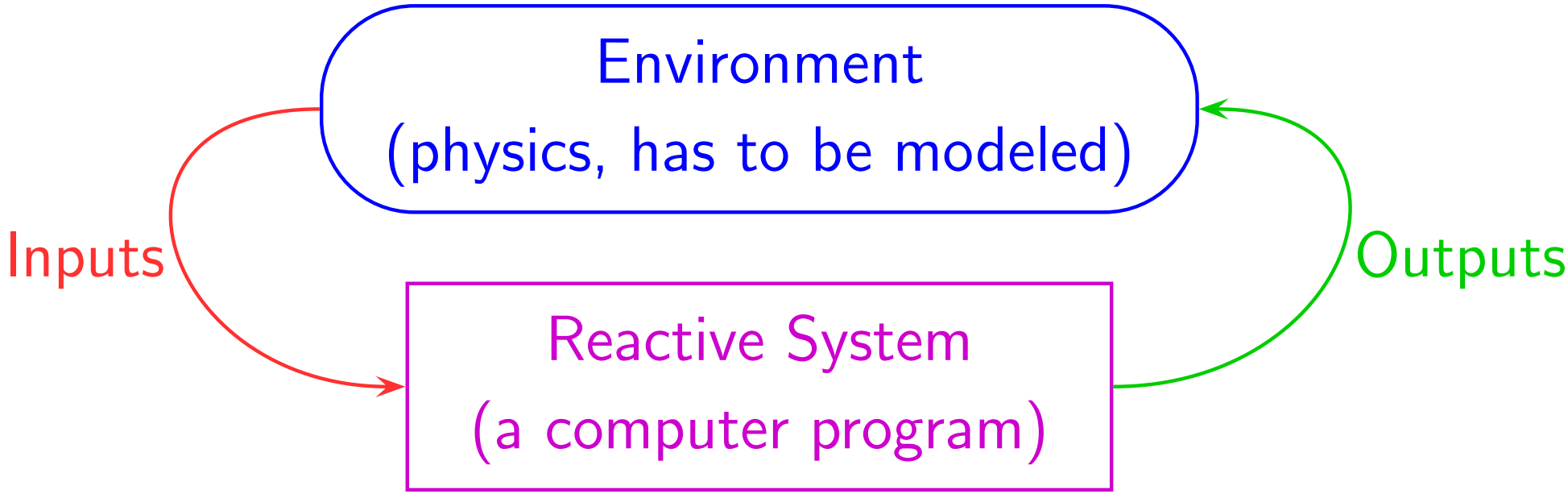


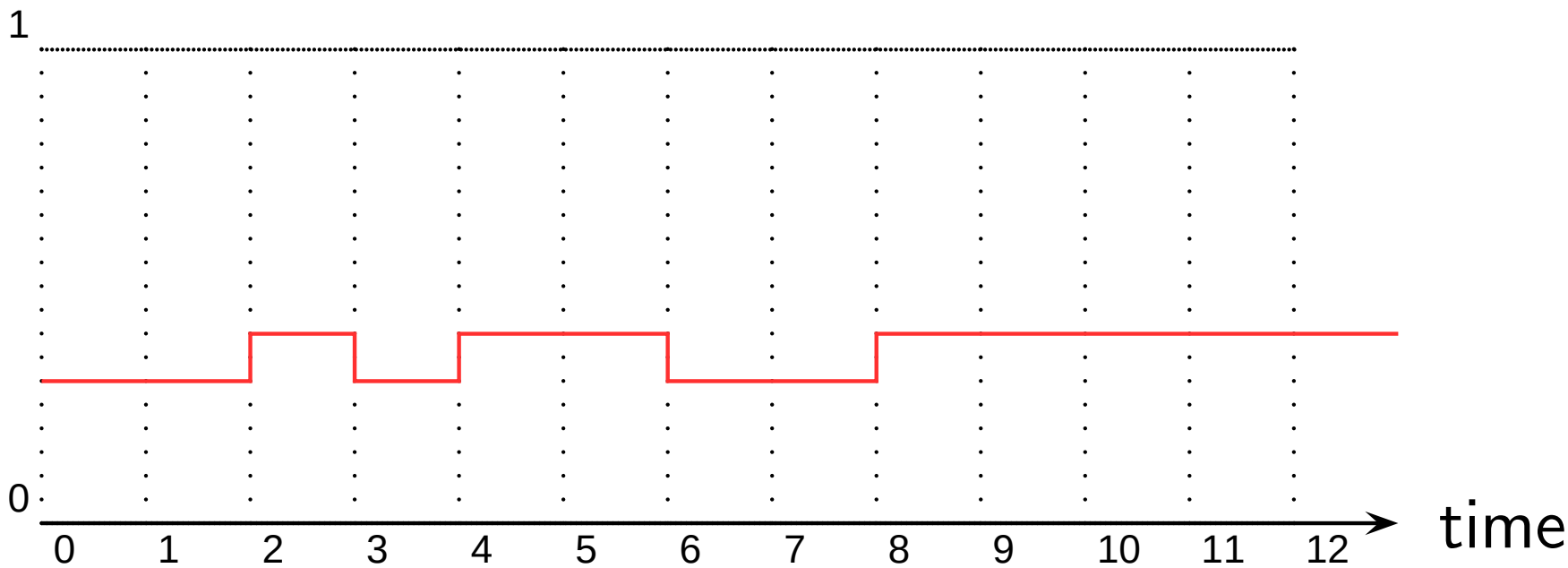
Reactive Systems in the Synchronous Approach



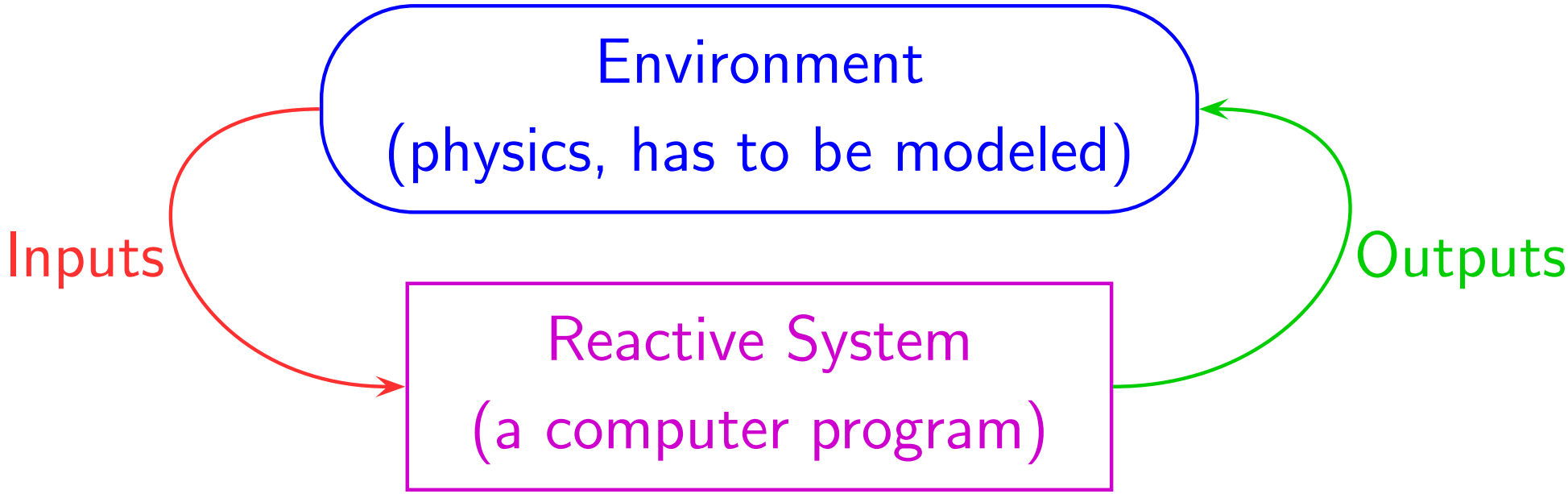


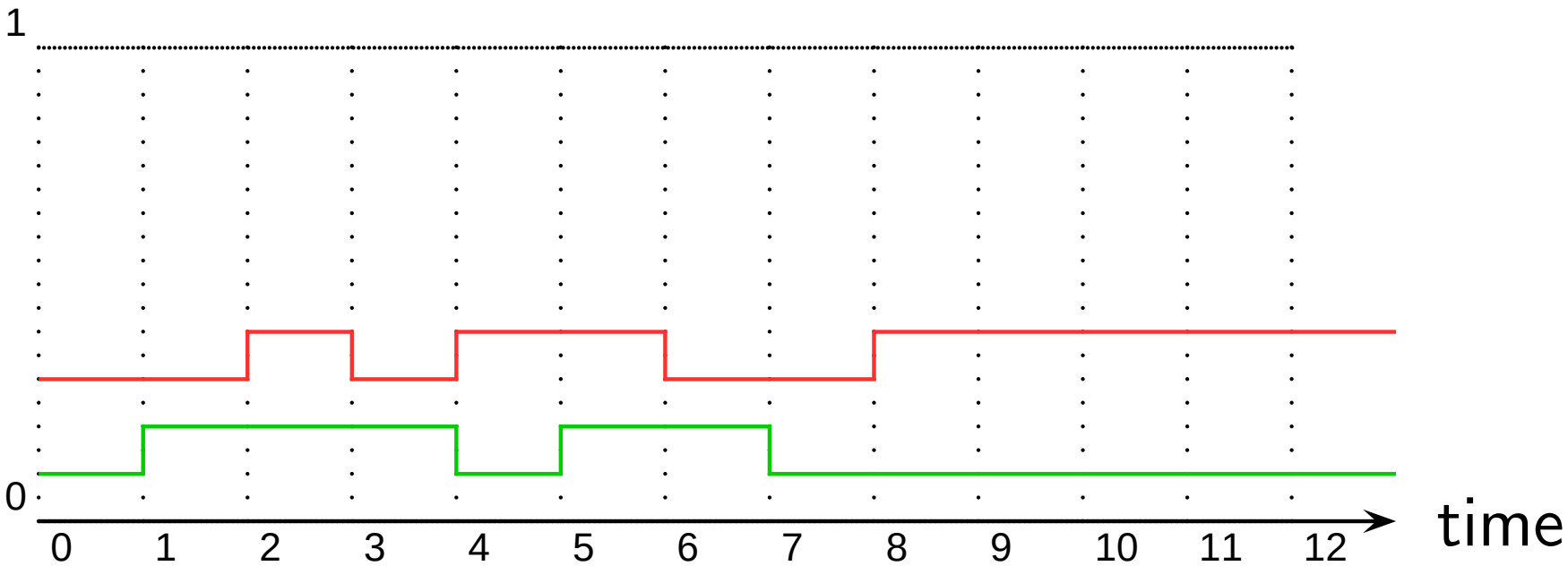
Reactive Systems in the Synchronous Approach



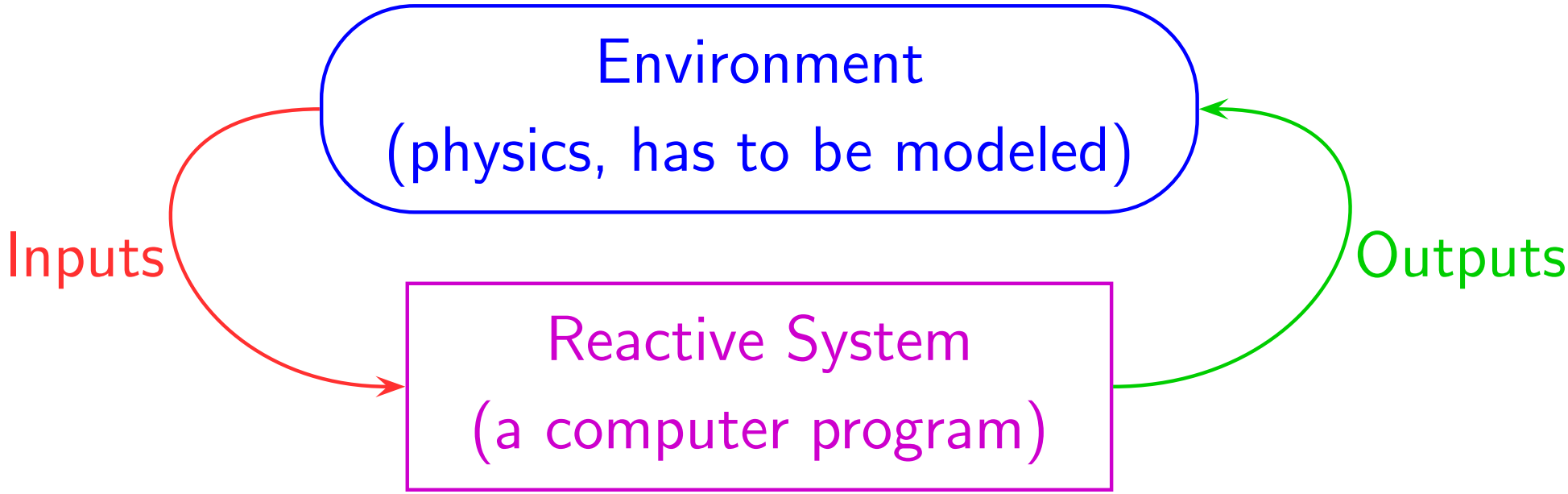


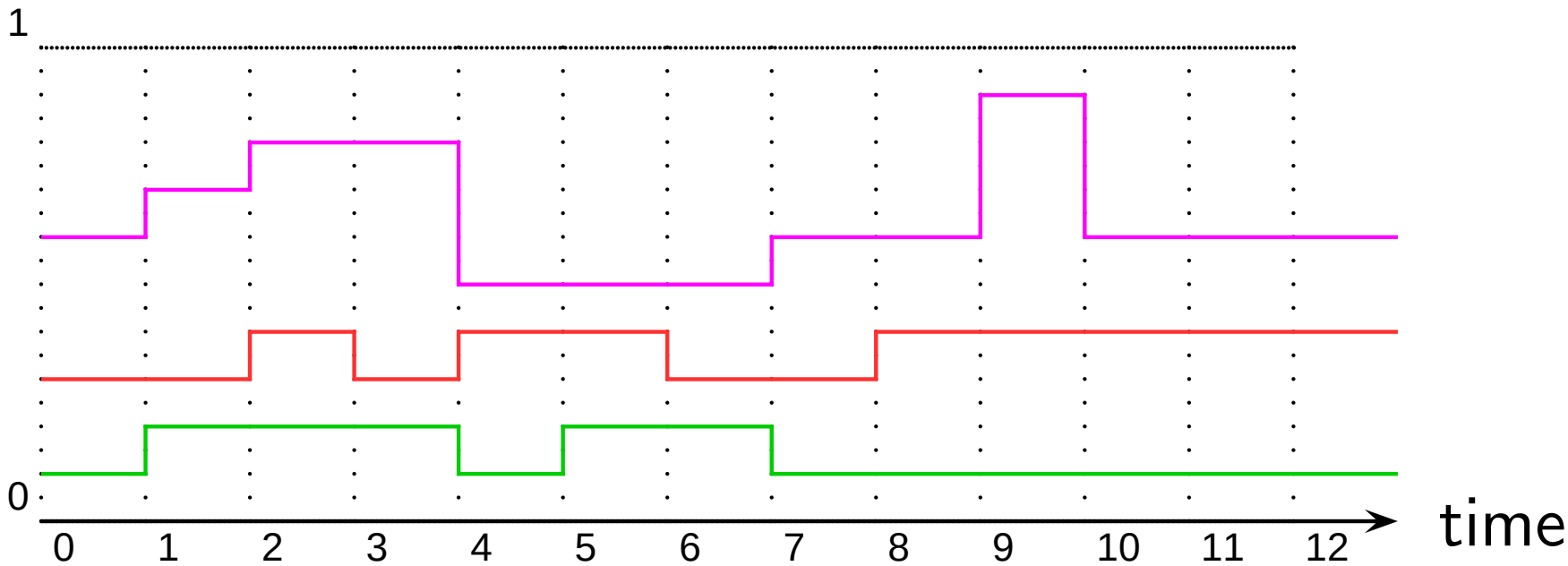
Reactive Systems in the Synchronous Approach





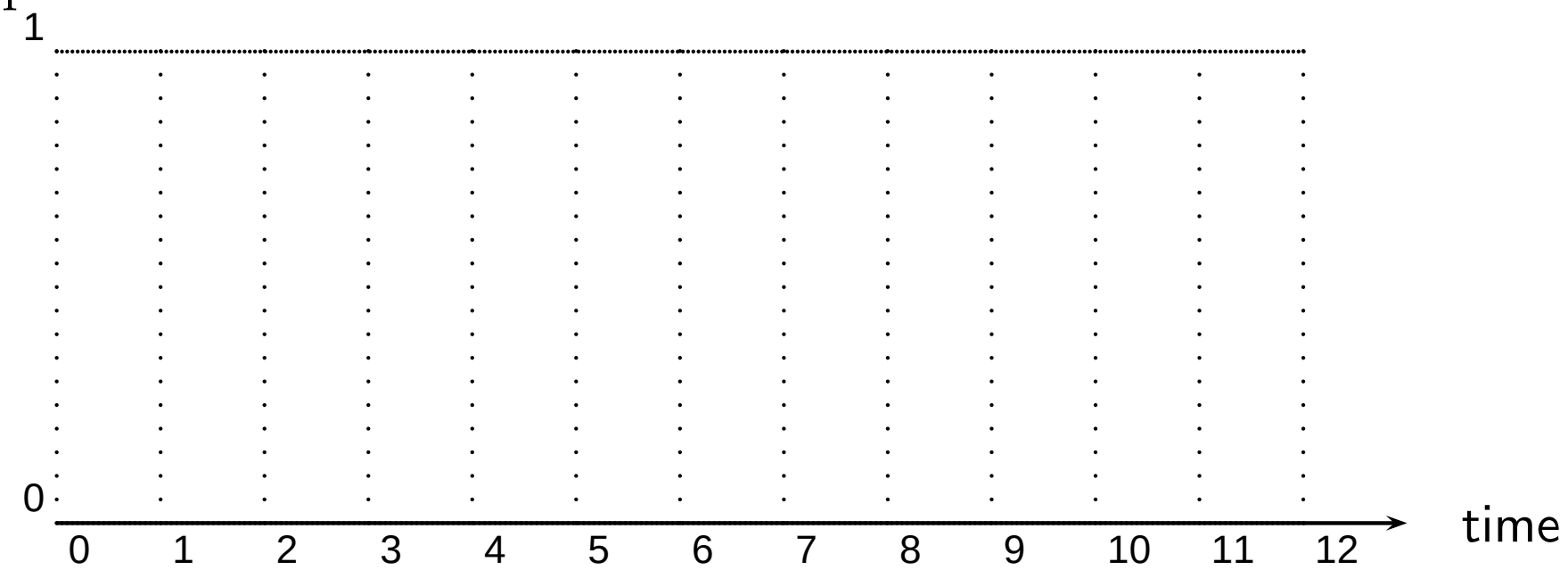
Reactive Systems in the Synchronous Approach





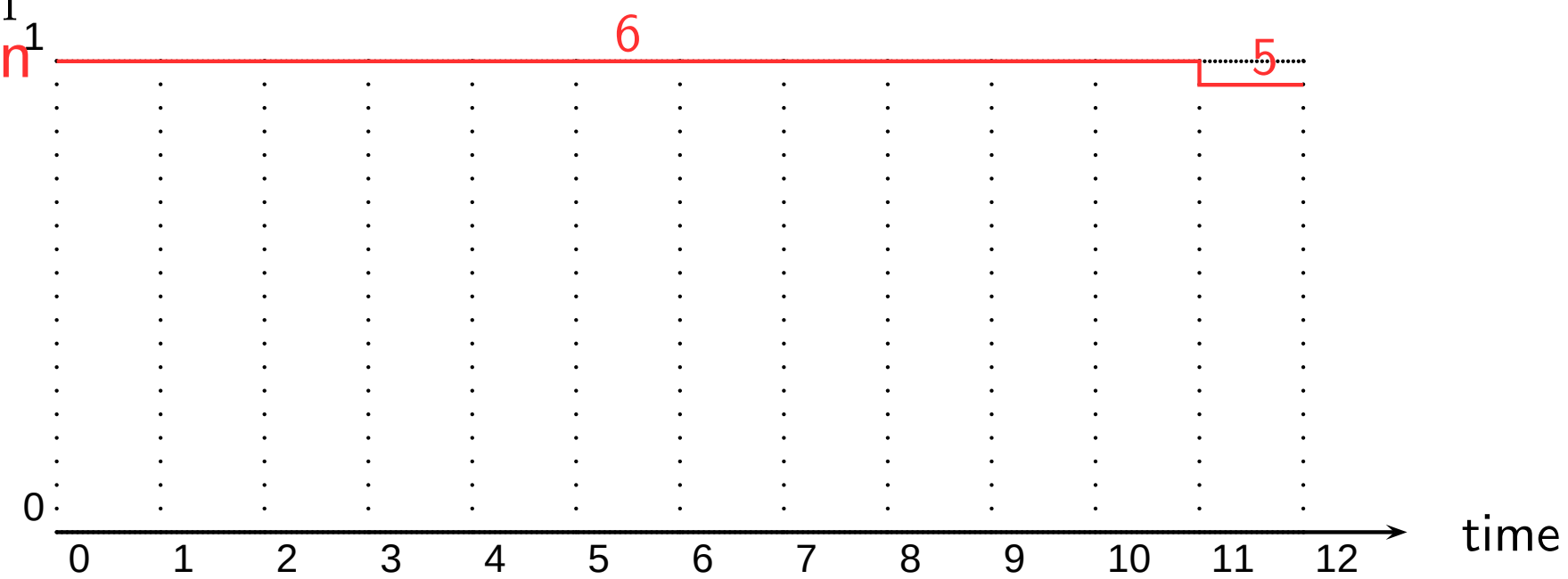
Our running example

```
node maintain(n:int; val:bool) returns (m:bool);
var cpt:int;
let cpt = if val then (0 -> pre(cpt)) + 1 else 0;
    m = cpt >= n;
    assert(n >= 5);
tel
```



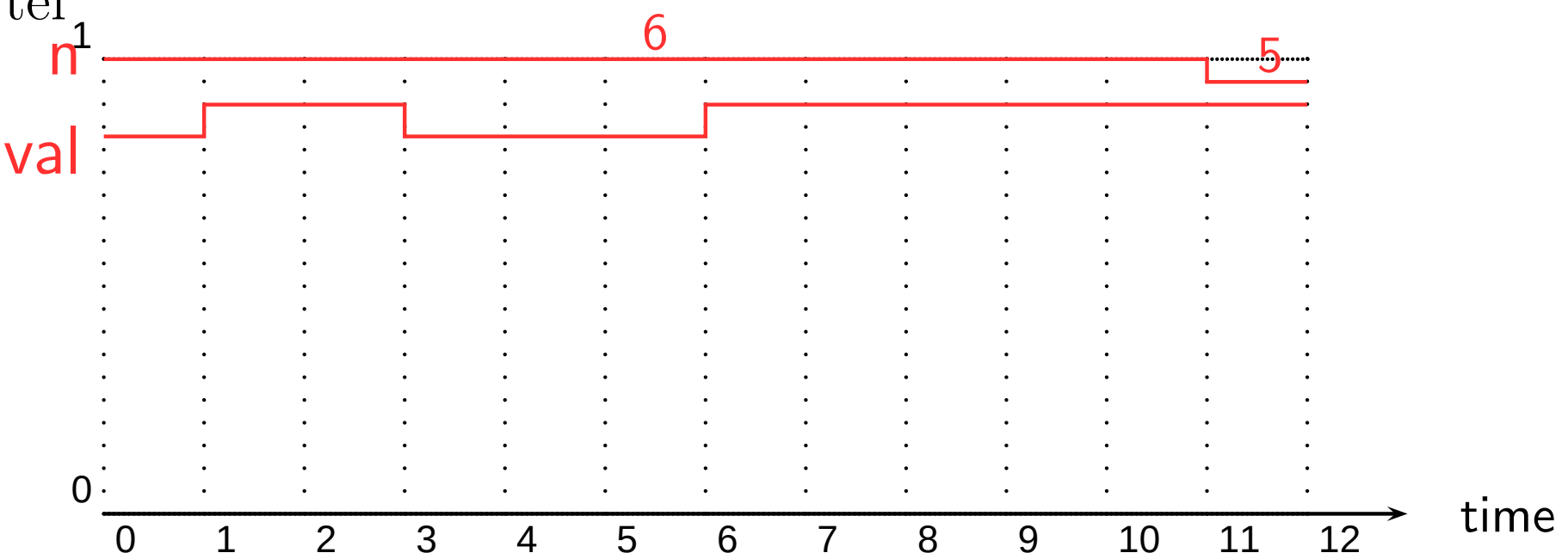
Our running example

```
node maintain(n:int; val:bool) returns (m:bool);
var cpt:int;
let cpt = if val then (0 -> pre(cpt)) + 1 else 0;
    m = cpt >= n;
    assert(n >= 5);
tel
```



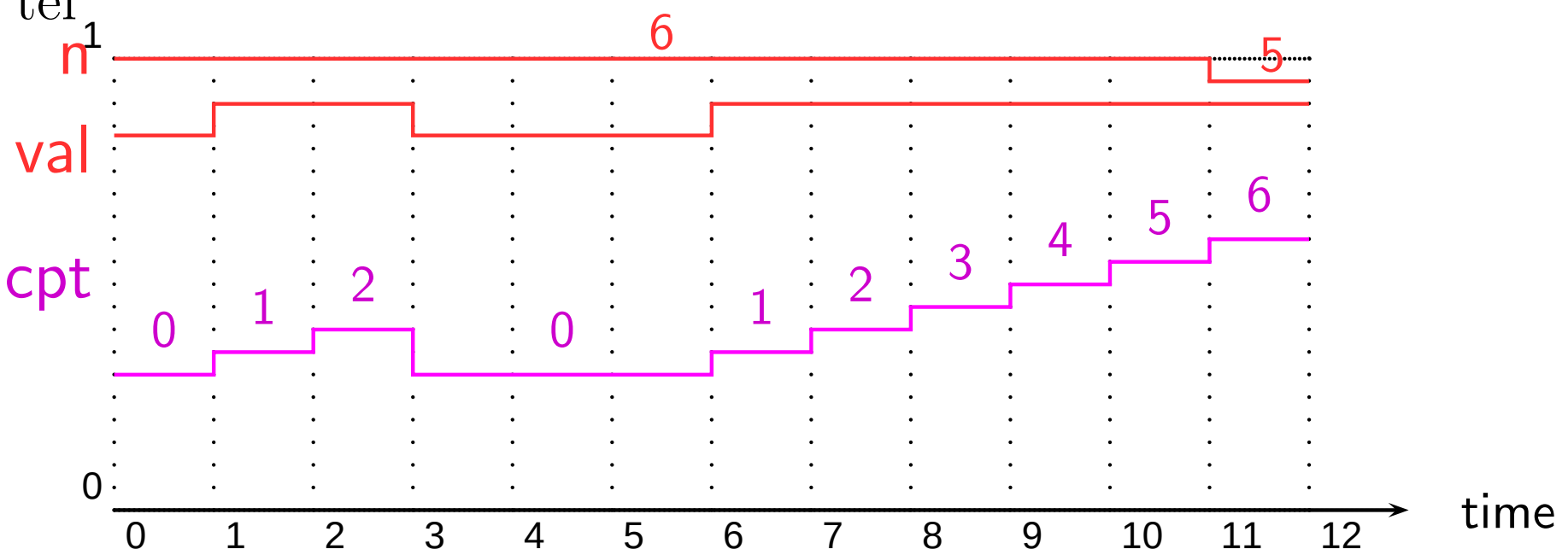
Our running example

```
node maintain(n:int; val:bool) returns (m:bool);
var cpt:int;
let cpt = if val then (0 -> pre(cpt)) + 1 else 0;
    m = cpt >= n;
    assert(n >= 5);
tel
```



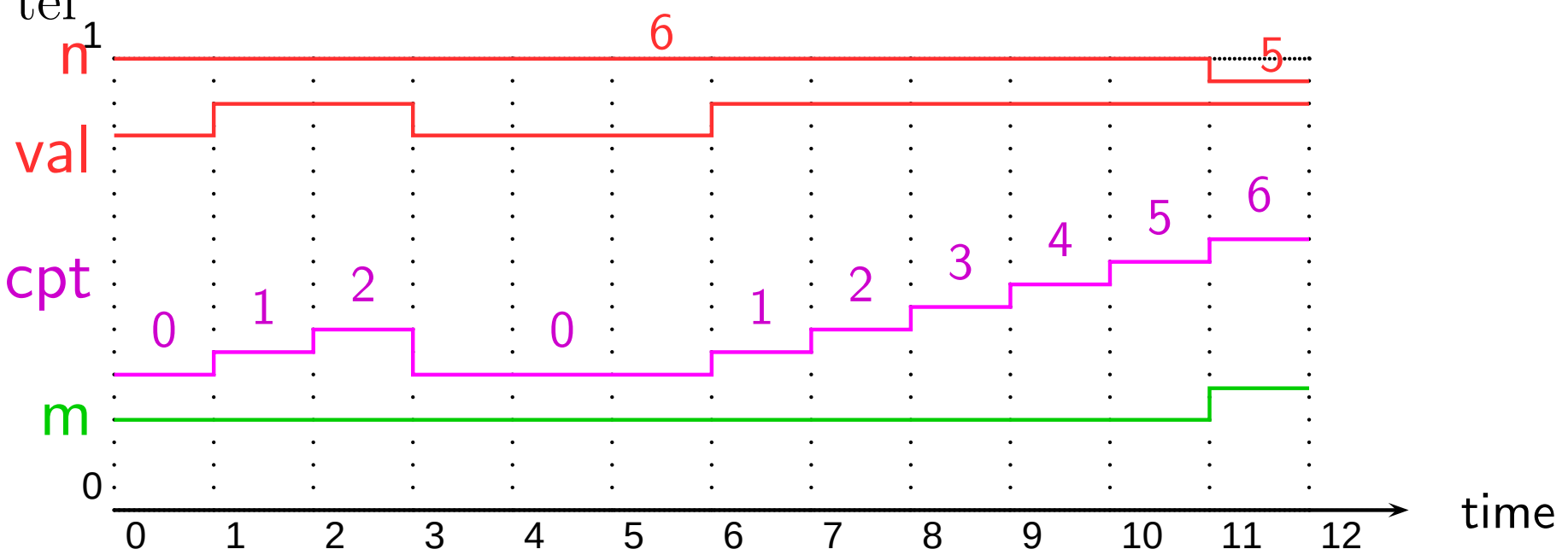
Our running example

```
node maintain(n:int; val:bool) returns (m:bool);
var cpt:int;
let cpt = if val then (0 -> pre(cpt)) + 1 else 0;
    m = cpt >= n;
    assert(n >= 5);
tel
```



Our running example

```
node maintain(n:int; val:bool) returns (m:bool);
var cpt:int;
let cpt = if val then (0 -> pre(cpt)) + 1 else 0;
    m = cpt >= n;
    assert(n >= 5);
tel
```



Our goal: a new debugging feature

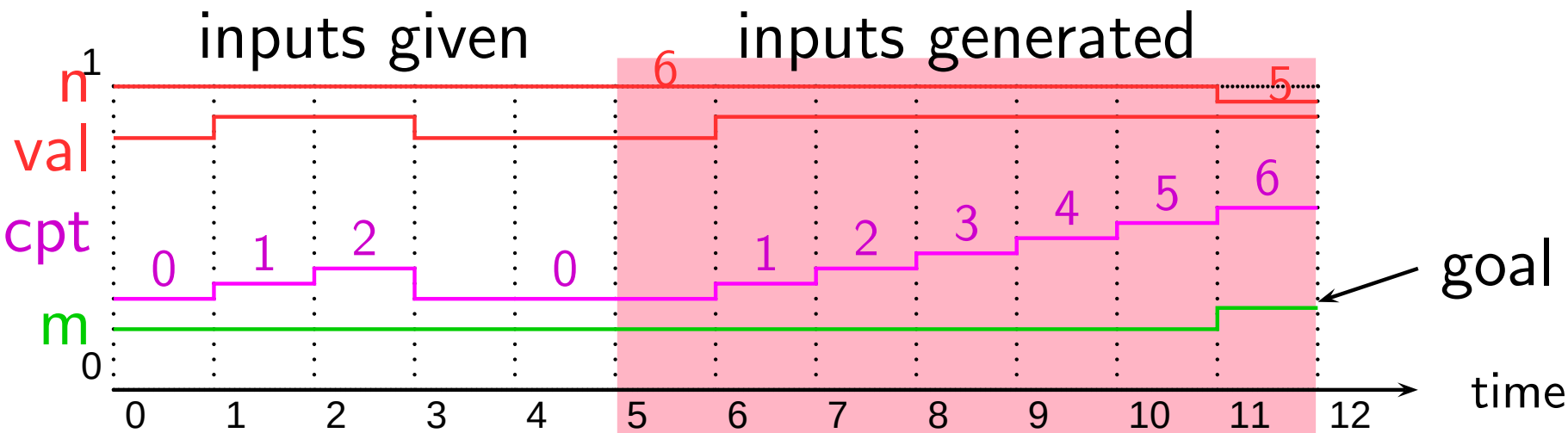
Useful features in a debugger

- Simulation: choose inputs and play the program until some instant n. Can be very tedious !

Our goal: a new debugging feature

Useful features in a debugger

- Simulation: choose inputs and play the program until some instant n. Can be very tedious !
- But also "goalpoint": is it possible to reach some interesting state (here, satisfying m true) ? If it is, give me a sequence of inputs (here, n,val) that, from the current state, leads to such a goal state.



Formalization

- We have a program, whose semantics can be described by $(s', o) = f(s, i)$
  s, s' are states, i the input, o the output
  An execution is: $s_0 \overset{i_0/o_0}{\longrightarrow} s_1 \overset{i_1/o_1}{\longrightarrow} s_2 \ldots$
  with $(s_{k+1}, o_k) = f(s_k, i_k)$
- We are in some state $\mathbf{Init} = \{s\}$
- We define the set of goal states $\mathbf{Final}$ by the mean of an observer

Question: can I reach a state in $\mathbf{Final}$ from a state in $\mathbf{Init}$, and how ?

Formalization

- We have a program, whose semantics can be described by $(s', o) = f(s, i)$
  s, s' are states, i the input, o the output
  An execution is: $s_0 \overset{i_0/o_0}{\longrightarrow} s_1 \overset{i_1/o_1}{\longrightarrow} s_2 \ldots$
  with $(s_{k+1}, o_k) = f(s_k, i_k)$
- We are in some state $\mathbf{Init} = \{s\}$
- We define the set of goal states $\mathbf{Final}$ by the mean of an observer

Question: can I reach a state in $\mathbf{Final}$ from a state in $\mathbf{Init}$, and how ?
Remark 1: if an observer depends on inputs and/or outputs, we can transform it into a state observer

Formalization

- We have a program, whose semantics can be described by $(s', o) = f(s, i)$
  s, s' are states, i the input, o the output
  An execution is: $s_0 \overset{i_0/o_0}{\longrightarrow} s_1 \overset{i_1/o_1}{\longrightarrow} s_2 \ldots$
  with $(s_{k+1}, o_k) = f(s_k, i_k)$
- We are in some state $\mathbf{Init} = \{s\}$
- We define the set of goal states $\mathbf{Final}$ by the mean of an observer

Question: can I reach a state in $\mathbf{Final}$ from a state in $\mathbf{Init}$, and how ?
Remark 2: equivalent view of the problem: search for a counter-example to the invariance property $\mathbf{not\ Final}$

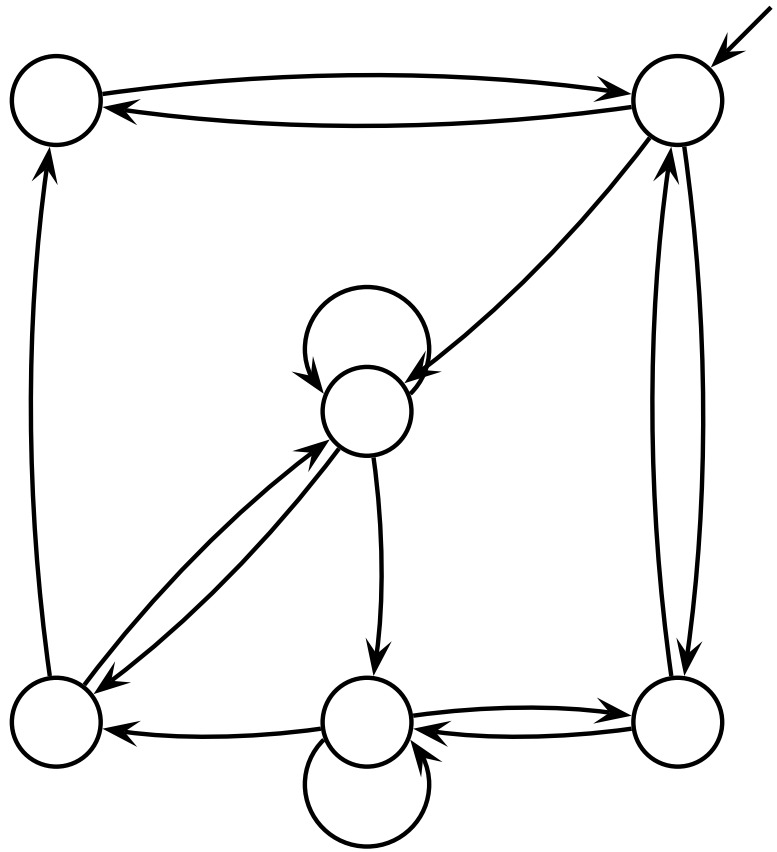
Simple case: Boolean programs

All variables are Booleans, the problem is decidable

Simple case: Boolean programs

All variables are Booleans, the problem is decidable

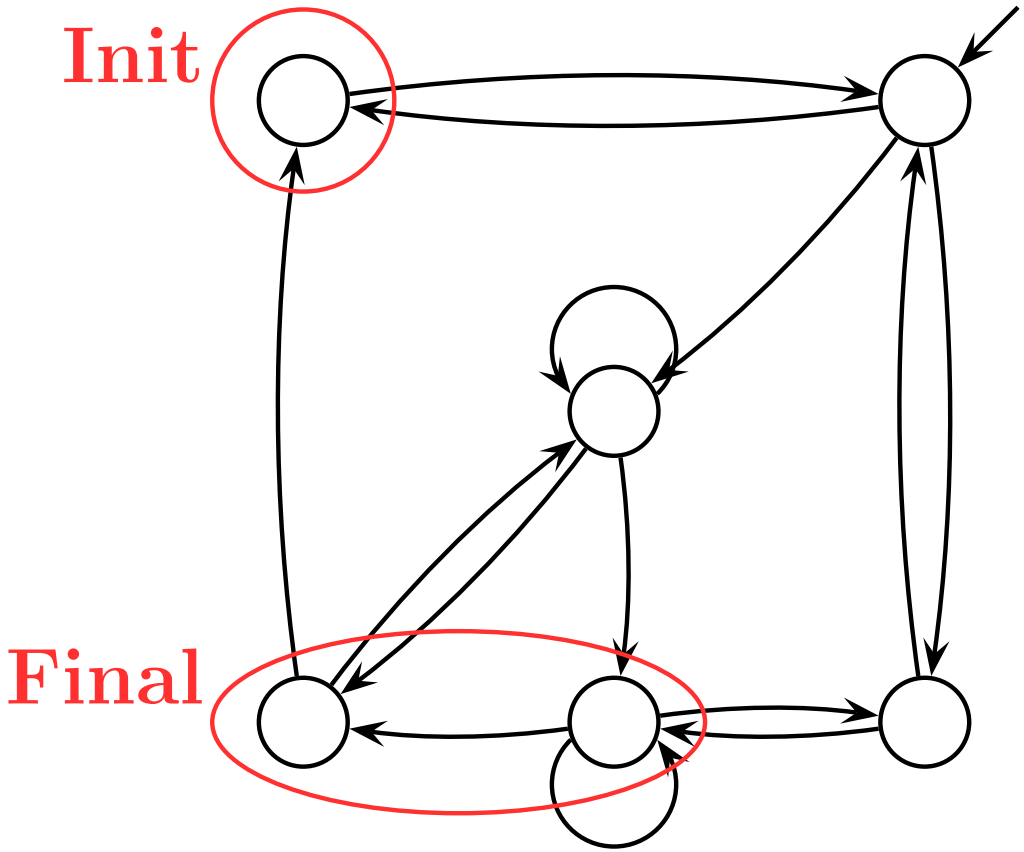
If the number of variables is small: build the Mealy automaton and use graph algorithms to find a path



Simple case: Boolean programs

All variables are Booleans, the problem is decidable

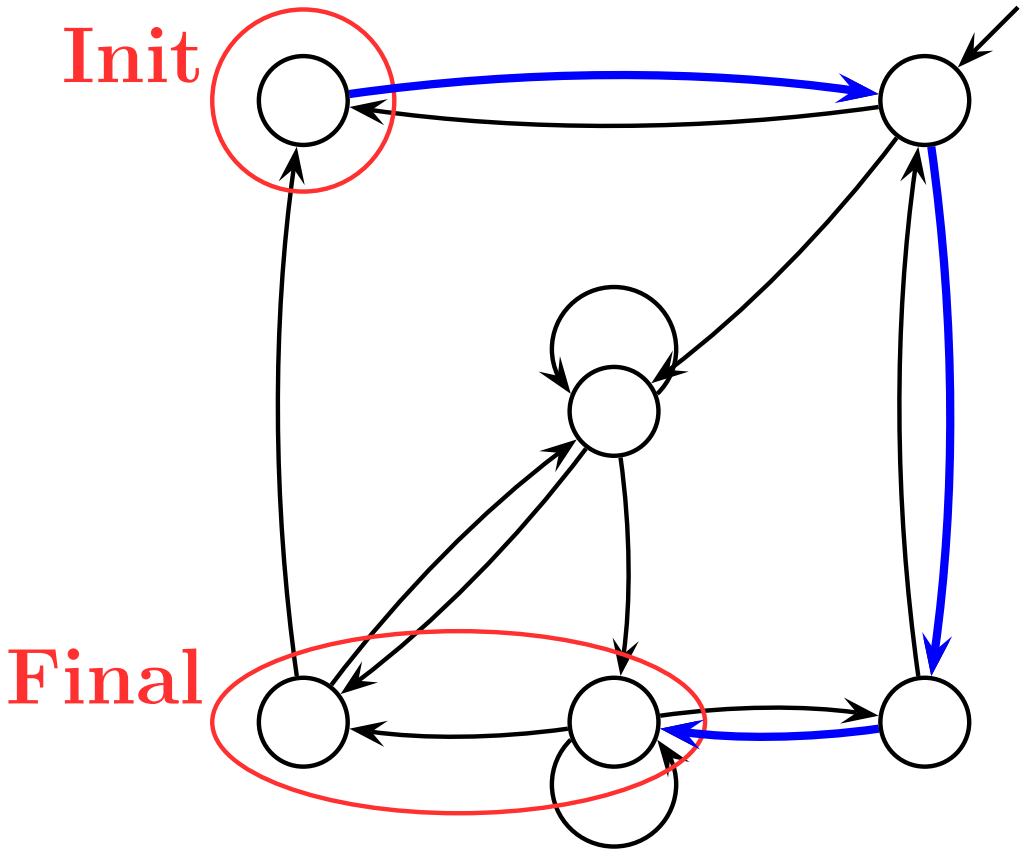
If the number of variables is small: build the Mealy automaton and use graph algorithms to find a path



Simple case: Boolean programs

All variables are Booleans, the problem is decidable

If the number of variables is small: build the Mealy automaton and use graph algorithms to find a path



Simple case: Boolean programs

All variables are Booleans, the problem is decidable

If the number of variables is bigger: uses symbolic techniques (BDDs, implemented in model-checkers)

Simple case: Boolean programs

All variables are Booleans, the problem is decidable

If the number of variables is bigger: uses symbolic techniques (BDDs, implemented in model-checkers)
Principle: forward fixpoint computation from **Init**, until **Final** is reached, and extraction of an execution

Simple case: Boolean programs

All variables are Booleans, the problem is decidable

If the number of variables is bigger: uses symbolic techniques (BDDs, implemented in model-checkers)

Principle: forward fixpoint computation from $\mathbf{Init}$, until $\mathbf{Final}$ is reached, and extraction of an execution

One compute $\boldsymbol{X_0 = \mathbf{Init}}$

$$\boldsymbol{X_{n+1} = post(X_n) \setminus \cup_{k \leq n} X_k}$$

Simple case: Boolean programs

All variables are Booleans, the problem is decidable

If the number of variables is bigger: uses symbolic techniques (BDDs, implemented in model-checkers)

Principle: forward fixpoint computation from $\mathbf{Init}$, until $\mathbf{Final}$ is reached, and extraction of an execution

One compute $\boldsymbol{X_0 = \mathbf{Init}}$

$$\boldsymbol{X_{n+1} = post(X_n) \setminus \cup_{k \leq n} X_k}$$

If convergence to $\emptyset$ with $\forall n : X_n \cap \mathbf{Final} = \emptyset$:
there is no execution from $\mathbf{Init}$ to $\mathbf{Final}$

Simple case: Boolean programs

All variables are Booleans, the problem is decidable

If the number of variables is bigger: uses symbolic techniques (BDDs, implemented in model-checkers)

Principle: forward fixpoint computation from $\mathbf{Init}$, until $\mathbf{Final}$ is reached, and extraction of an execution

One compute $\boldsymbol{X_0 = \mathbf{Init}}$

$$\boldsymbol{X_{n+1} = post(X_n) \setminus \cup_{k \leq n} X_k}$$

If convergence to $\emptyset$ with $\forall n : X_n \cap \mathbf{Final} = \emptyset$:
there is no execution from $\mathbf{Init}$ to $\mathbf{Final}$

Otherwise, one takes $s_n \in X_n \cap \mathbf{Final}$,

$s_{n-1} \in pre(\{s_n\}) \cap X_{n-1}$ and $i_{n-1} \mid s_{n-1} \stackrel{i_{n-1}/?}{\longrightarrow} s_n$,
and so on until $s_O \in \mathbf{Init}$, $\Rightarrow$ shortest execution

Less simple case: programs with numerical variables

Infinite state-space and non-decidability of the problem

$\Rightarrow$ Abstraction/Approximation needed

Less simple case: programs with numerical variables

Infinite state-space and non-decidability of the problem

$\Rightarrow$ Abstraction/Approximation needed

Idea of our method:

With a verification tool we compute first an over-approximation of the set of states S' which are both

Less simple case: programs with numerical variables

Infinite state-space and non-decidability of the problem

$\Rightarrow$ Abstraction/Approximation needed

Idea of our method:

With a verification tool we compute first an over-approximation of the set of states S' which are both

- reachable from $\mathbf{Init}$ (like before)

Less simple case: programs with numerical variables

Infinite state-space and non-decidability of the problem

$\Rightarrow$ Abstraction/Approximation needed

Idea of our method:

With a verification tool we compute first an over-approximation of the set of states S' which are both

- reachable from $\mathbf{Init}$ (like before)
- and also coreachable from $\mathbf{Final}$

Less simple case: programs with numerical variables

Infinite state-space and non-decidability of the problem

$\Rightarrow$ Abstraction/Approximation needed

Idea of our method:

With a verification tool we compute first an over-approximation of the set of states S' which are both

- reachable from $\mathbf{Init}$ (like before)
- and also coreachable from $\mathbf{Final}$

With a test generation tool we try to find a concrete execution from $\mathbf{Init}$ to $\mathbf{Final}$, restricting the search space to S'

Less simple case: programs with numerical variables

Infinite state-space and non-decidability of the problem

$\Rightarrow$ Abstraction/Approximation needed

Idea of our method:

With a verification tool we compute first an over-approximation of the set of states S' which are both

- reachable from $\mathbf{Init}$ (like before)
- and also coreachable from $\mathbf{Final}$

With a test generation tool we try to find a concrete execution from $\mathbf{Init}$ to $\mathbf{Final}$, restricting the search space to S'

Method already experimented by Halbwachs, Pace & Raymond 01, here refinement of it

Connecting tools

Debugger (Ludic)

$\boldsymbol{X} \longrightarrow \boldsymbol{Y}$?

Connecting tools

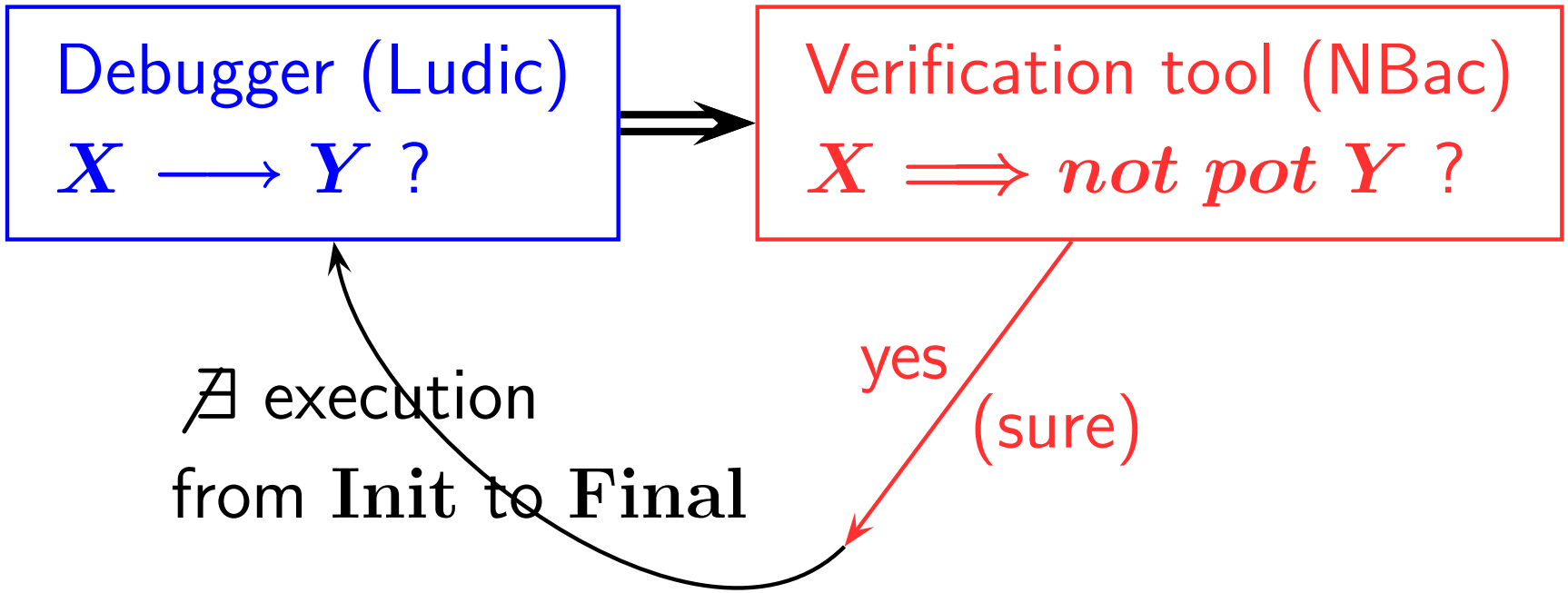
Debugger (Ludic)

$X \longrightarrow Y$?

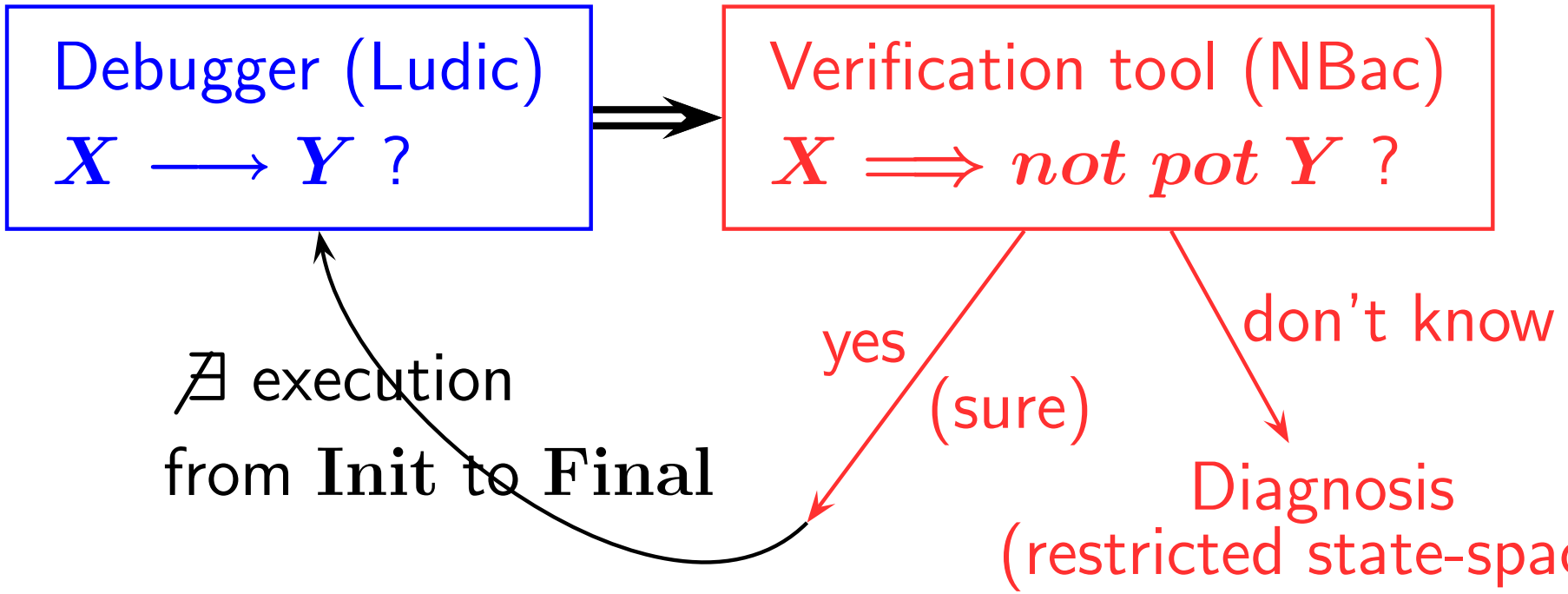
Verification tool (NBac)

$X \Longrightarrow not\ pot\ Y$?

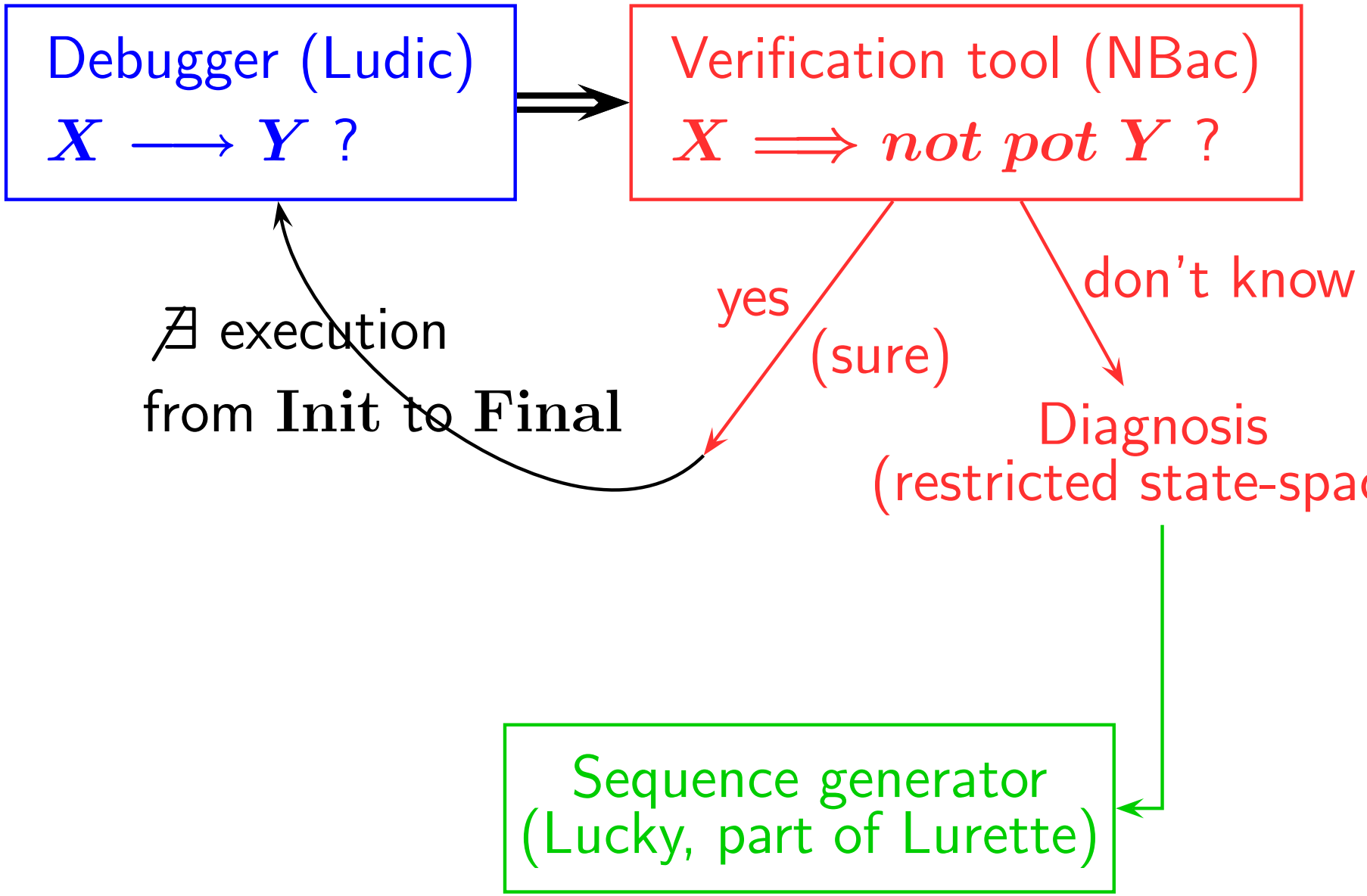
Connecting tools



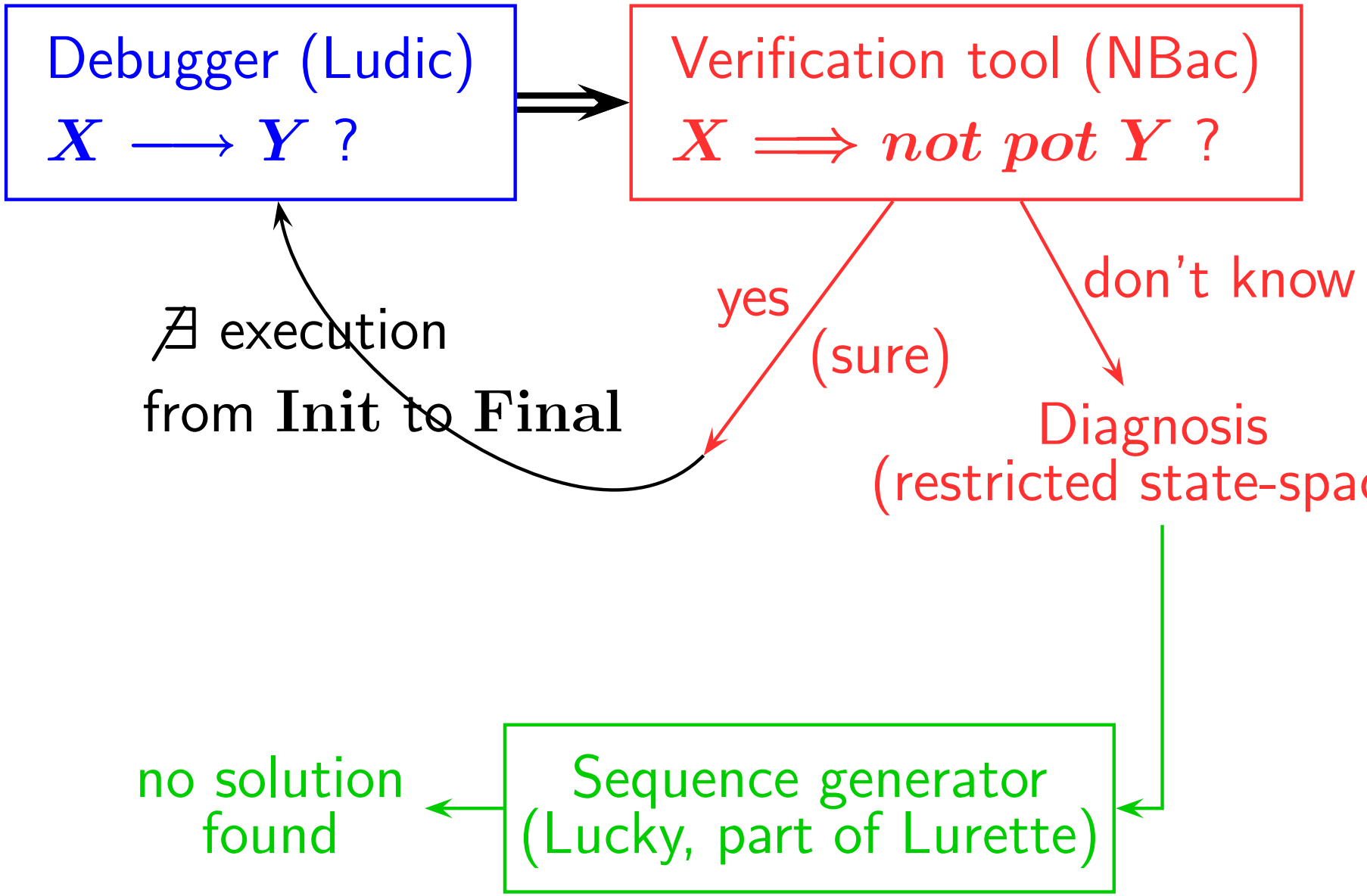
Connecting tools



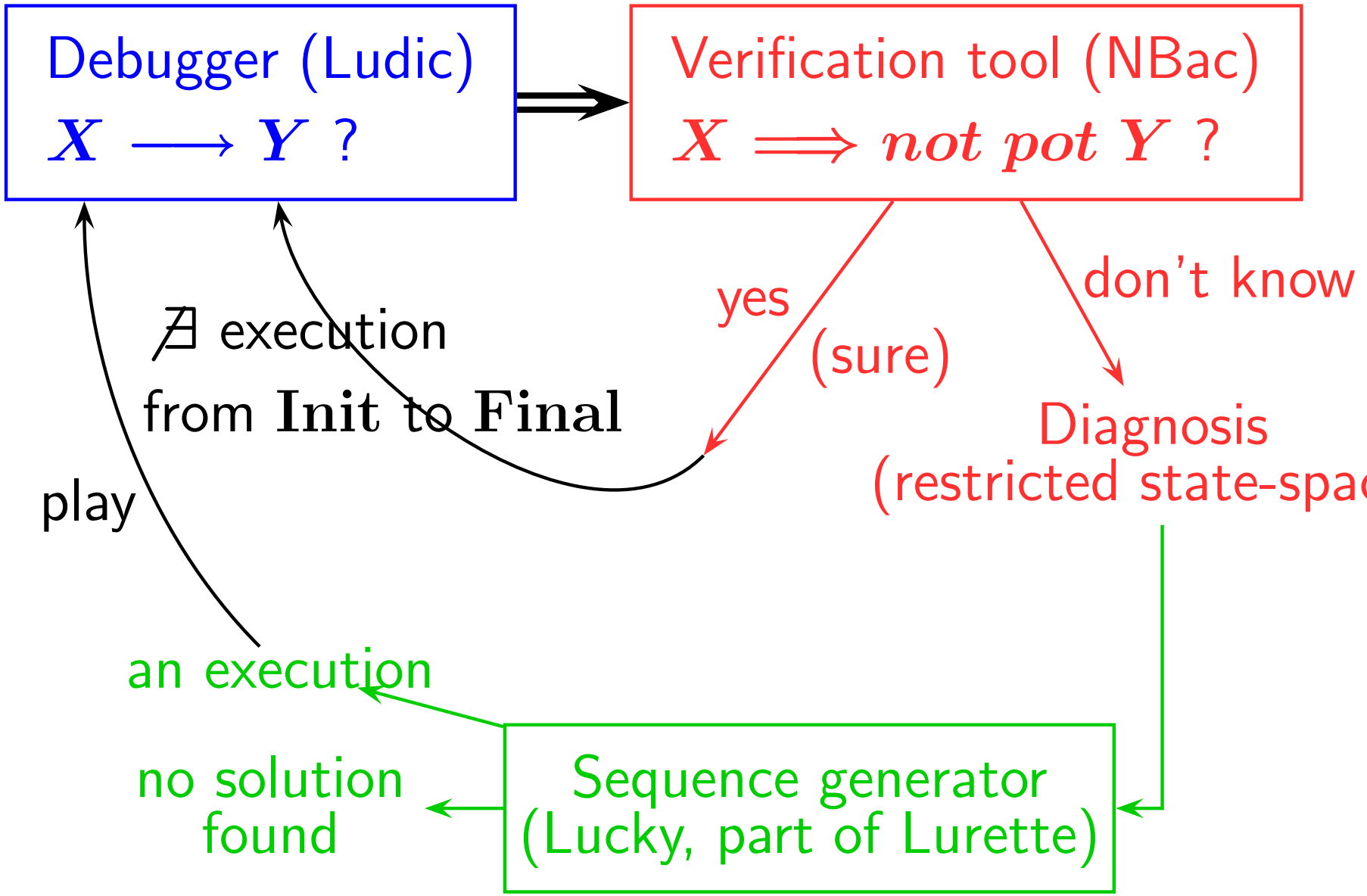
Connecting tools



Connecting tools

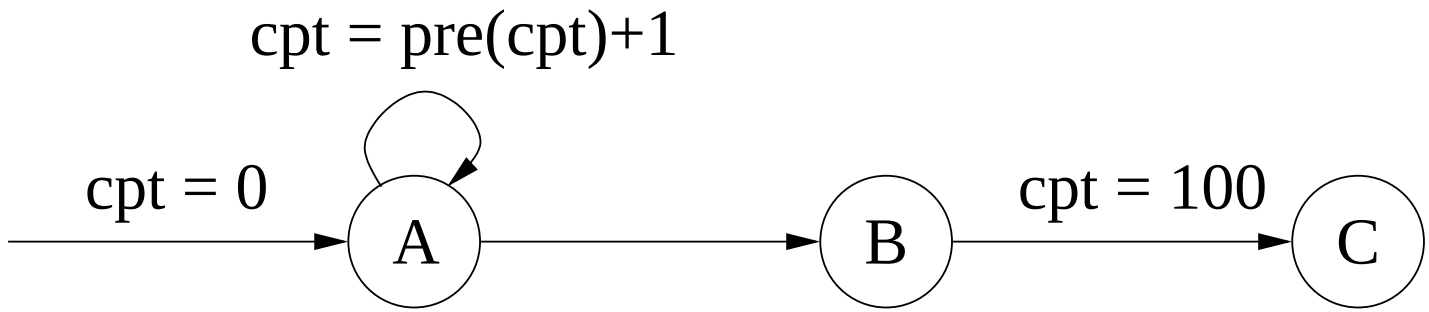


Connecting tools



Why restricting the search space is important

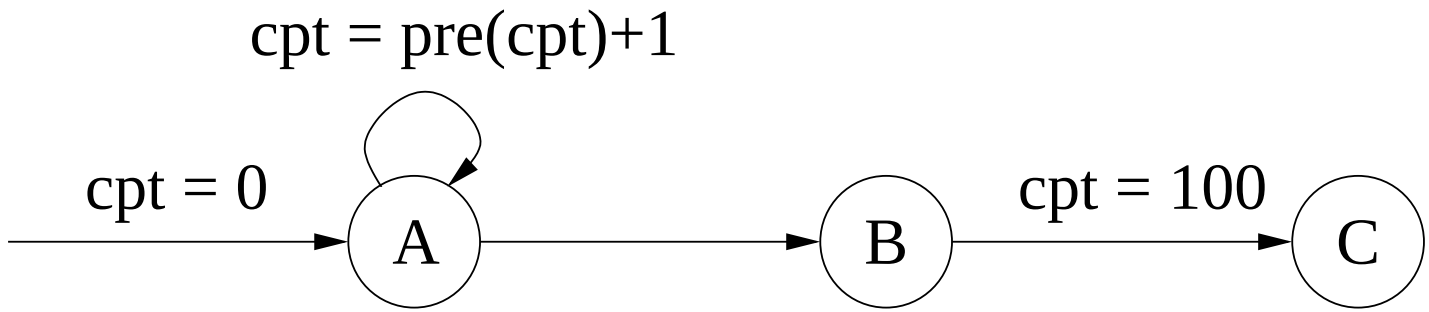
A problematic case:



Here, we are unlikely to produce a successfull sequence !

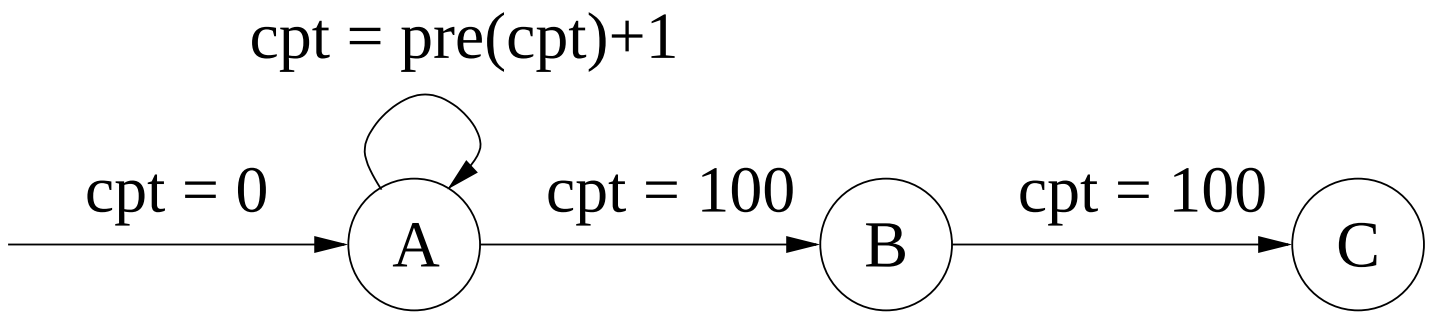
Why restricting the search space is important

A problematic case:



Here, we are unlikely to produce a successfull sequence !

However, a backward analysis on numerical variables gives:



The debugger Ludic

Debugger for Lustre programs

Offer among other features:

- step-by-step execution, both forward and backward
- inspection of program's state
- slicing algorithms to analyse computations within a step

Role of Ludic in the present context: given a program, a current execution state and a goal property observer,

- performs slicing of the program to select its relevant part of w.r.t. the goal property
- generates the verification problem for NBac
- at the end, in case of success, plays the computed execution

From Ludic to NBac

Given the program

```
node maintain(n:int; val:bool) returns (m:bool);
var cpt:int;
let cpt = if val then (0 -> pre(cpt)) + 1 else 0;
    m = cpt >= n;
    assert(n >= 5);
tel
```

and the observer

```
node goal(m:bool) returns (ok:bool);
let ok = false -> pre(m); tel
```

From Ludic to NBac

```
state
   init, pre_m : bool;
   pre_cpt : int;
input
   val : bool;
   n : int;
local
   goal, start : bool;
   m : bool;
   cpt : int;
definition
   start = (not init) and (pre_cpt = 0);
   goal = if start then false else pre_m;
   cpt = if val then pre_cpt + 1 else 0;
   m = cpt >= n;
transition
   pre_cpt' = cpt;
   pre_m' = m;
   init' = false;
assertion
   n>=5;
initial start;
final goal;
```

NBac

Verification tool for synchronous (dataflow) programs with Boolean and numerical variables

Functionalities: approximate reachability, coreachability analysis, and their combination (intersection)

NBac

Verification tool for synchronous (dataflow) programs with Boolean and numerical variables

Functionalities: approximate reachability, coreachability analysis, and their combination (intersection)

How it works ? it is based on abstract interpretation techniques

- "Basic" abstract values: (BDD, polyèdre convexe)
  (Conjunction of a Boolean invariant and a numerical invariant)

NBac

Verification tool for synchronous (dataflow) programs with Boolean and numerical variables

Functionalities: approximate reachability, coreachability analysis, and their combination (intersection)

How it works ? it is based on abstract interpretation techniques

- "Basic" abstract values: (BDD, polyèdre convexe)
  (Conjunction of a Boolean invariant and a numerical invariant)
- To improve the precision, we partition the state-space to replace single abstract values by bounded union of abstract values

Actually, partition $\Leftrightarrow$ explicit control structure

NBac

Verification tool for synchronous (dataflow) programs with Boolean and numerical variables

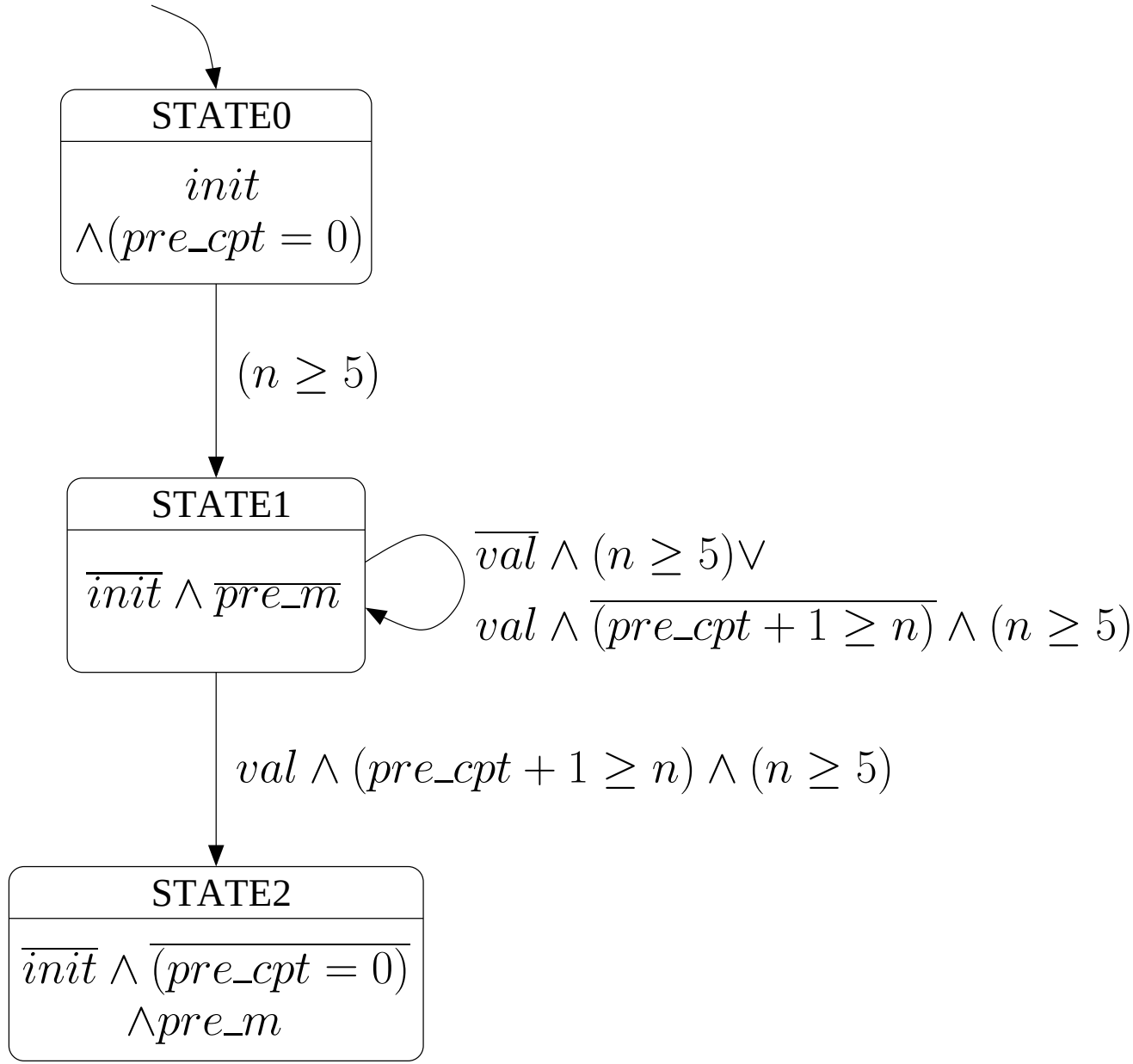
Functionalities: approximate reachability, coreachability analysis, and their combination (intersection)

How it works ? it is based on abstract interpretation techniques

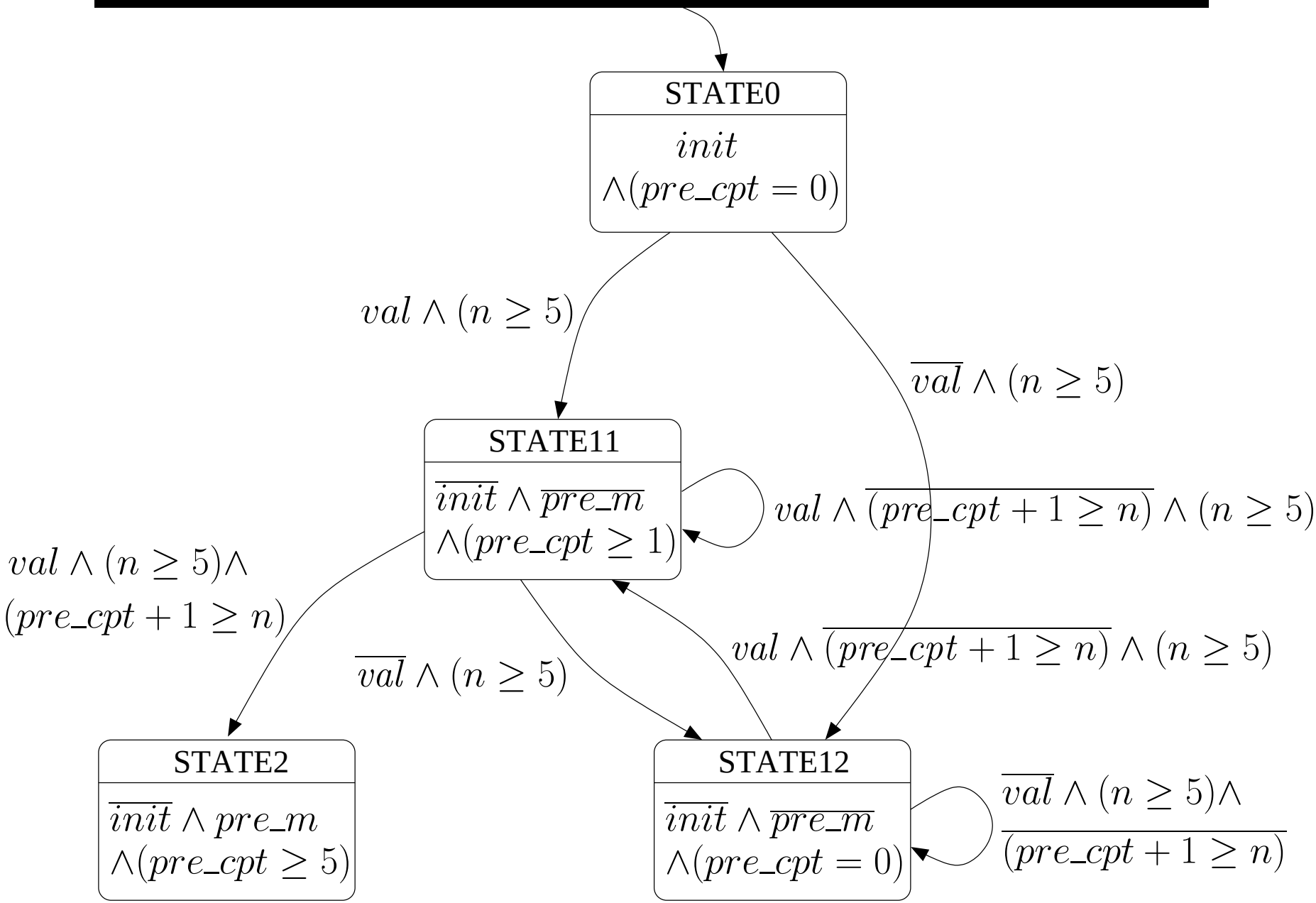
- "Basic" abstract values: (BDD, polyèdre convexe)
  (Conjunction of a Boolean invariant and a numerical invariant)
- To improve the precision, we partition the state-space to replace single abstract values by bounded union of abstract values
- Automatic partition refinement heuristics

Actually, partition $\Leftrightarrow$ explicit control structure

NBac on the example: initial control structure



NBac on the example: output control structure

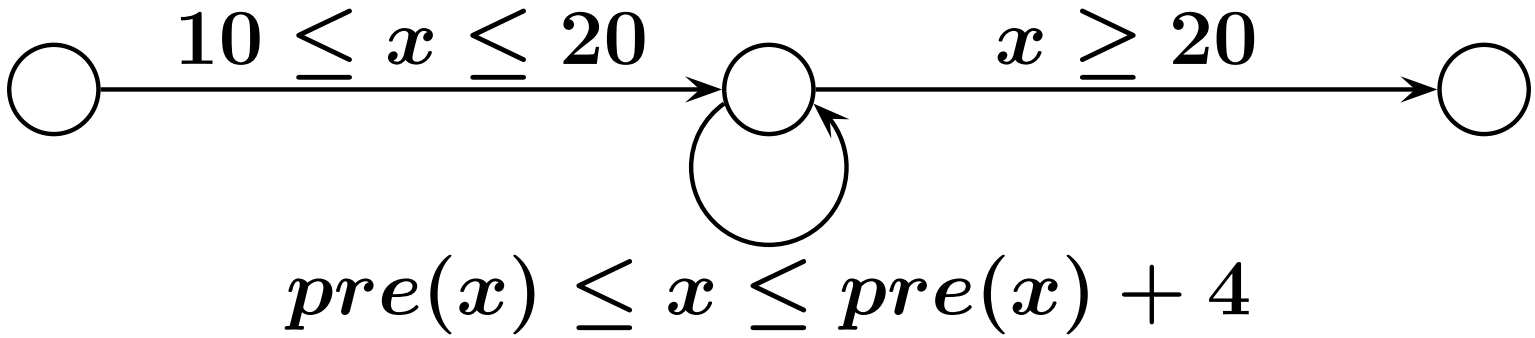


The sequence generator Lucky

Generates sequences from a non-deterministic automaton

Main use: modeling realistic environments

Example of a Lucky automaton:

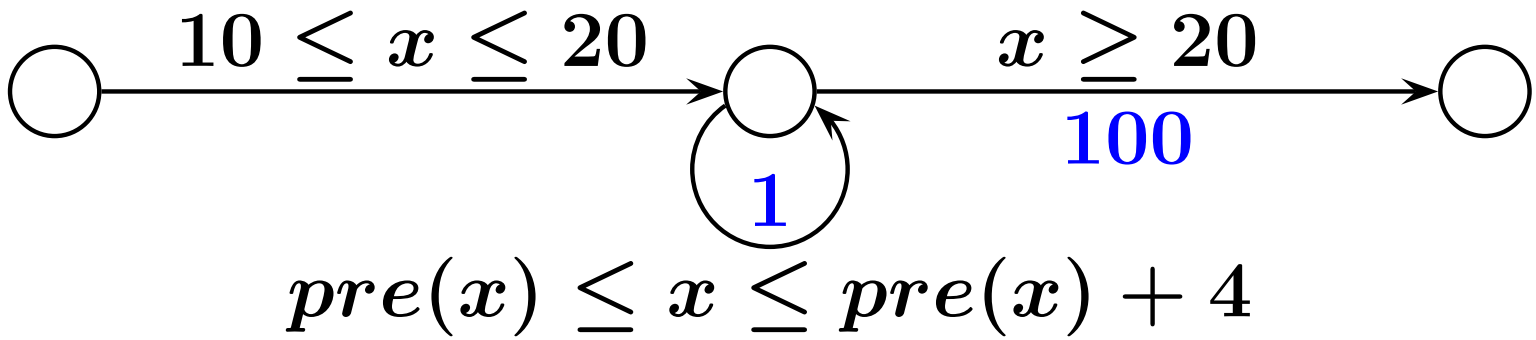


The sequence generator Lucky

Generates sequences from a non-deterministic automaton

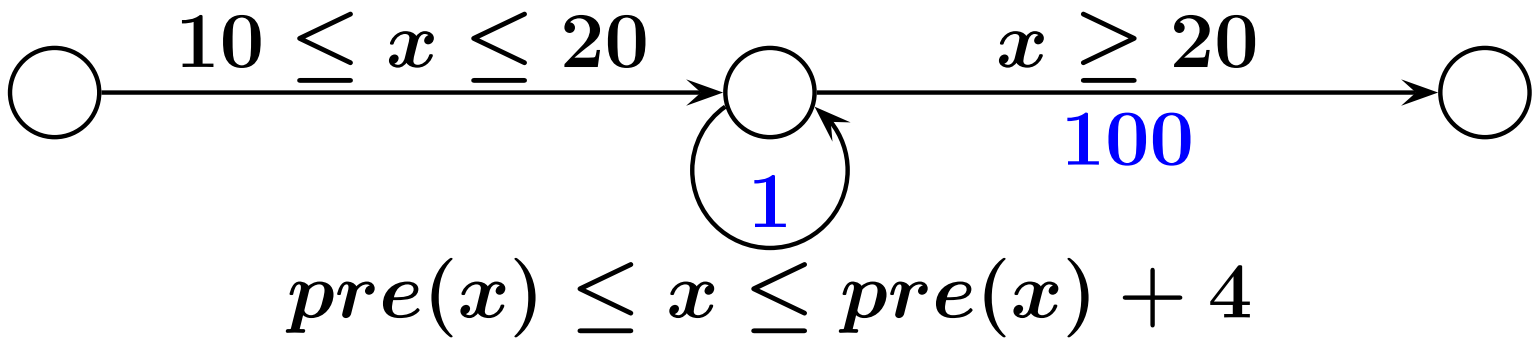
Main use: modeling realistic environments

Example of a Lucky automaton:



The sequence generator Lucky

Generates sequences from a non-deterministic automaton
Main use: modeling realistic environments
Example of a Lucky automaton:

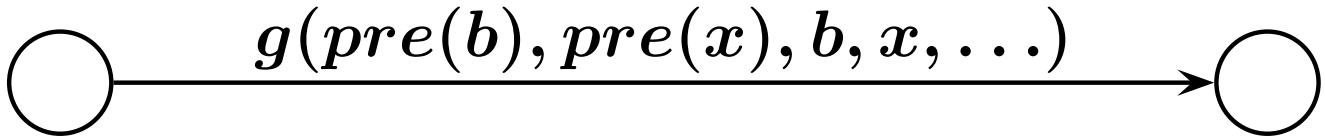


Use in our context:

- automaton generating all the sequences (states & inputs) staying in the restricted state space
- guiding the exploration of the Lucky tool, so as to obtain a (short) sequence leading to a $\mathbf{Final}$ state

The sequence generator Lucky

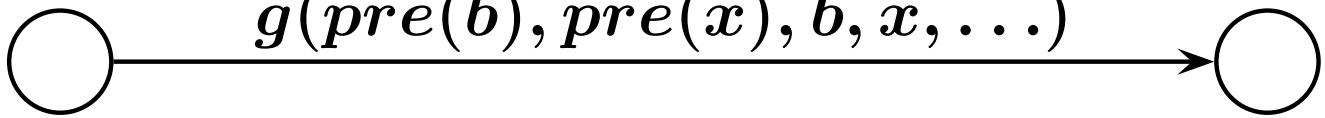
How it works ?



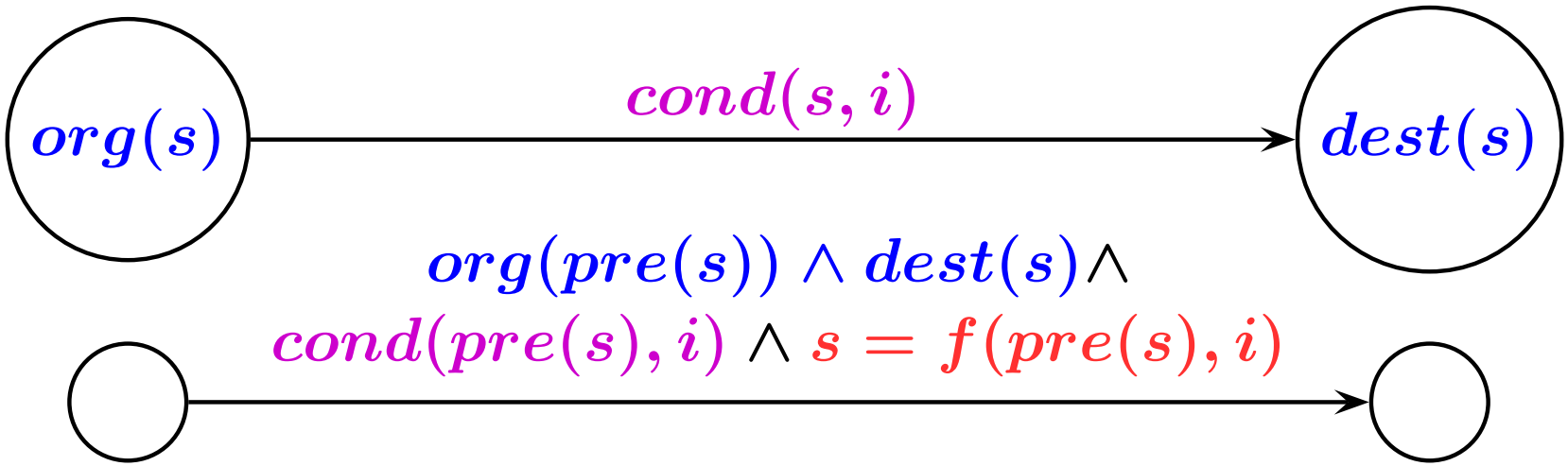
Given a memory state (value of $pre(\cdot)$), builds the set of solutions for $b, x, \ldots$ such that $g = true$
(Using BDDs and convex polyhedra)
Performs uniform random choice to extract one solution

The sequence generator Lucky

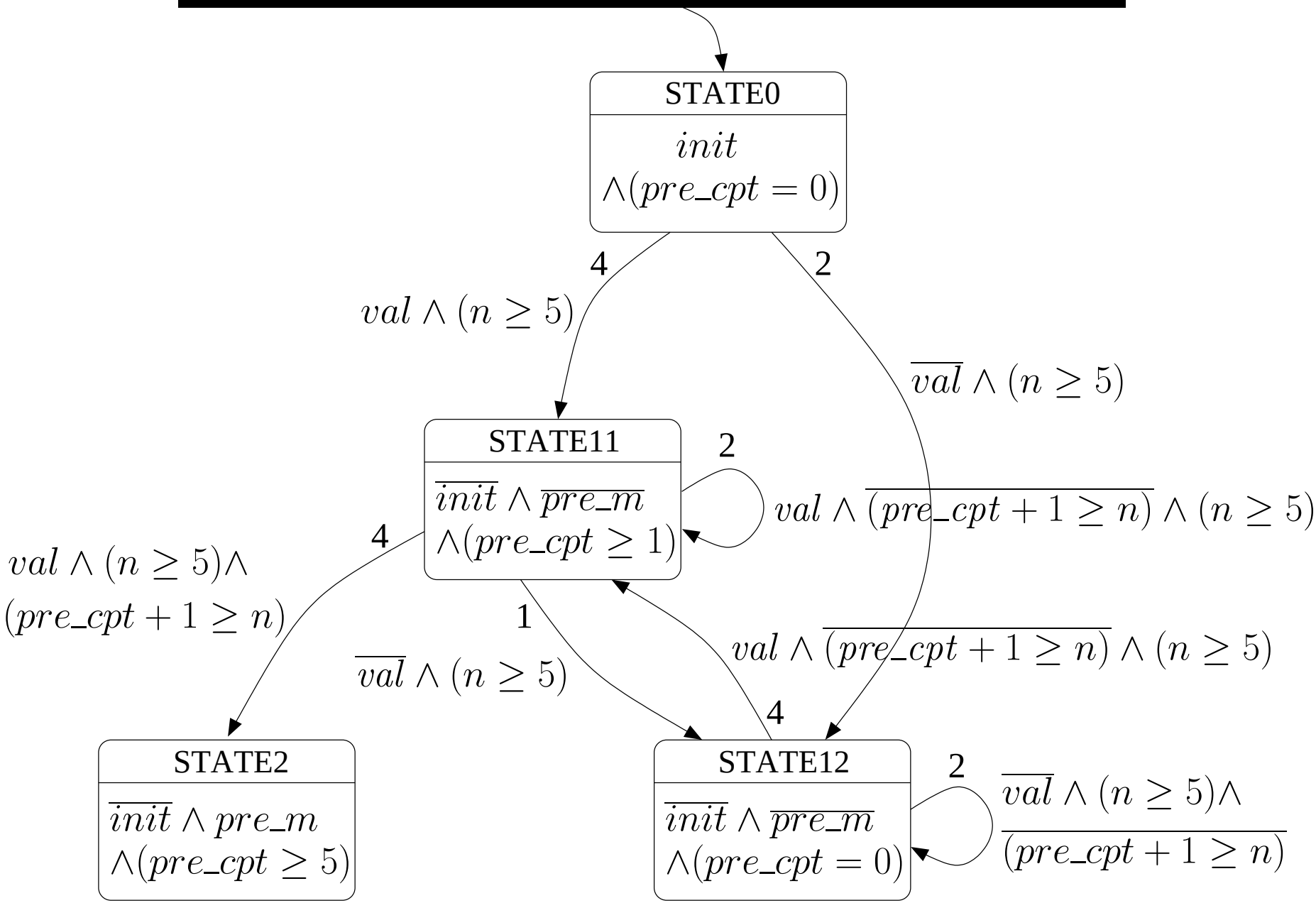
How it works ?



Given a memory state (value of $pre(\cdot)$), builds the set of solutions for $b, x, \ldots$ such that $g = true$
(Using BDDs and convex polyhedra)
Performs uniform random choice to extract one solution
Translation of NBac to Lucky



Decorate NBac automaton with weights



Putting weight in NBac automaton

Main heuristics:

We compute for each location l the minimal distance $\delta(l)$ (in number of edges) leading to a $\mathbf{Final}$ location

Weight of an edge $l \longrightarrow l'$: $p^{\delta(l)-\delta(l')}$

$p > 0$: parameter of the heuristics

Intuition: favorizing edges getting closer to the goal location, and inversally for edges getting more far of it

Putting weight in NBac automaton

Main heuristics:

We compute for each location l the minimal distance $\delta(l)$ (in number of edges) leading to a $\mathbf{Final}$ location

Weight of an edge $l \longrightarrow l'$: $p^{\delta(l)-\delta(l')}$

$p > 0$: parameter of the heuristics

Intuition: favorizing edges getting closer to the goal location, and inversally for edges getting more far of it

In the example: with $p \simeq 2, 3, 4$, obtained sequences are always of length < 15

Putting weight in NBac automaton

Main heuristics:

We compute for each location l the minimal distance $\delta(l)$ (in number of edges) leading to a $\mathbf{Final}$ location

Weight of an edge $l \longrightarrow l'$: $p^{\delta(l)-\delta(l')}$

$p > 0$: parameter of the heuristics

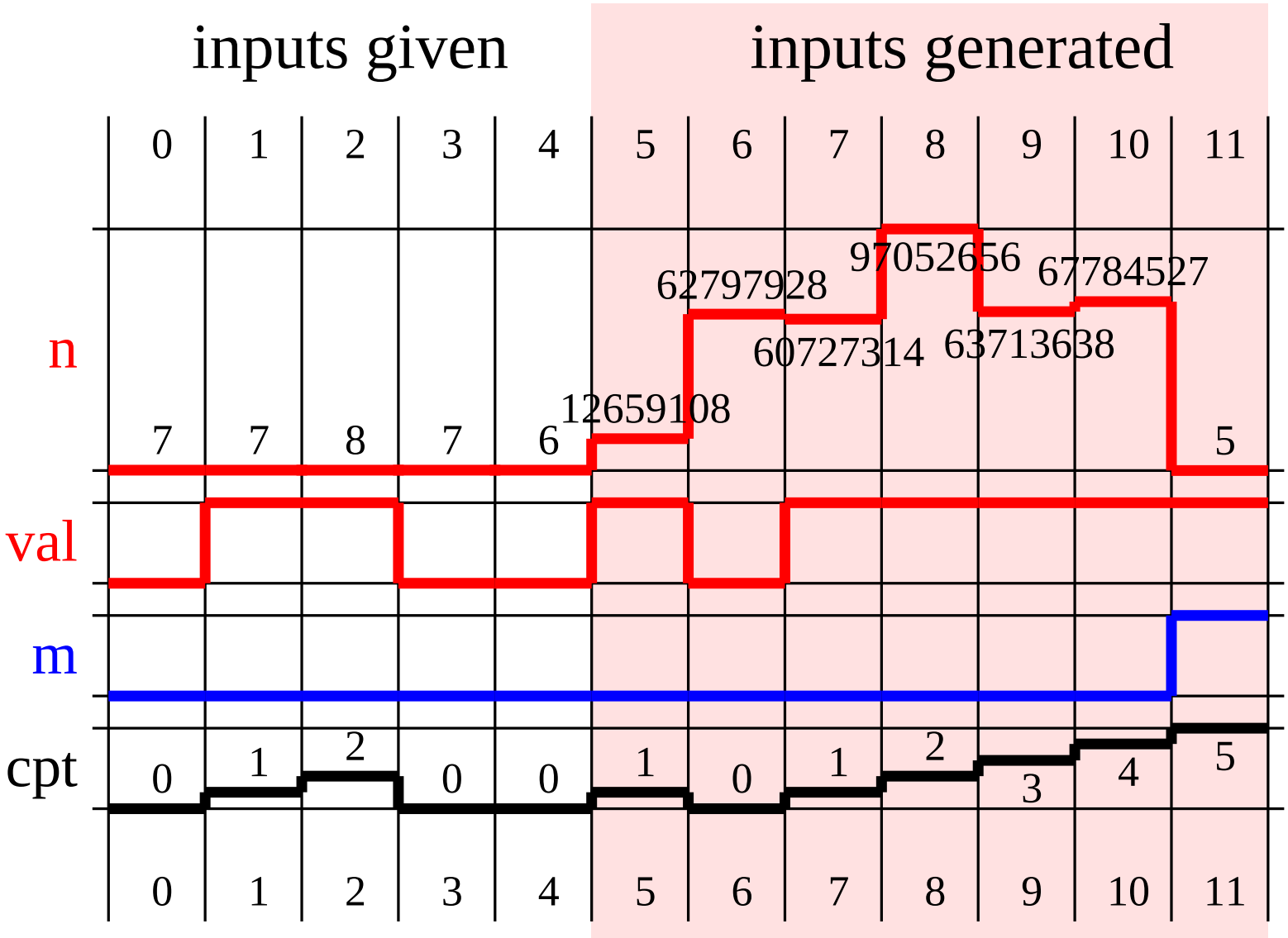
Intuition: favorizing edges getting closer to the goal location, and inversally for edges getting more far of it

In the example: with $p \simeq 2, 3, 4$, obtained sequences are always of length < 15

Other possible heuristics: Using Strongly Connected (Sub-)Components decomposition, and favorizing edges exiting components

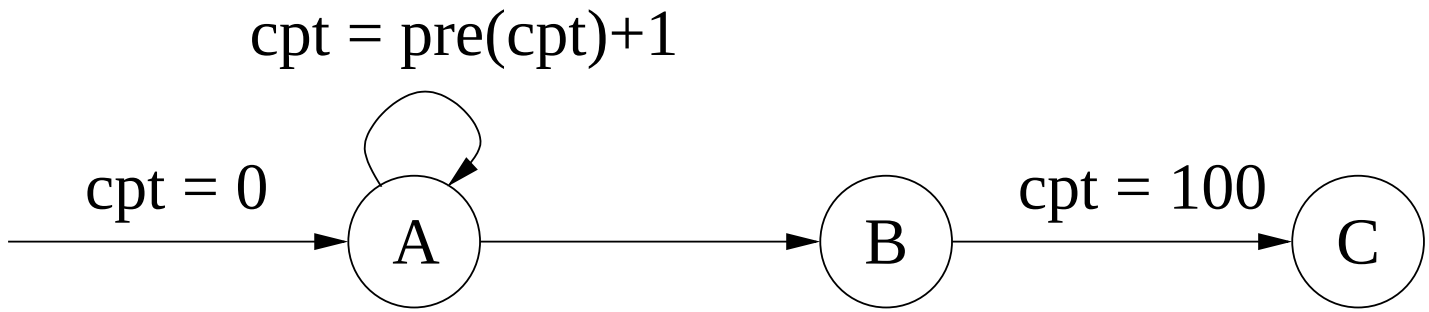
(Could be combined with the previous heuristics)

A sequence generated by Lucky



Why restricting the search space is important

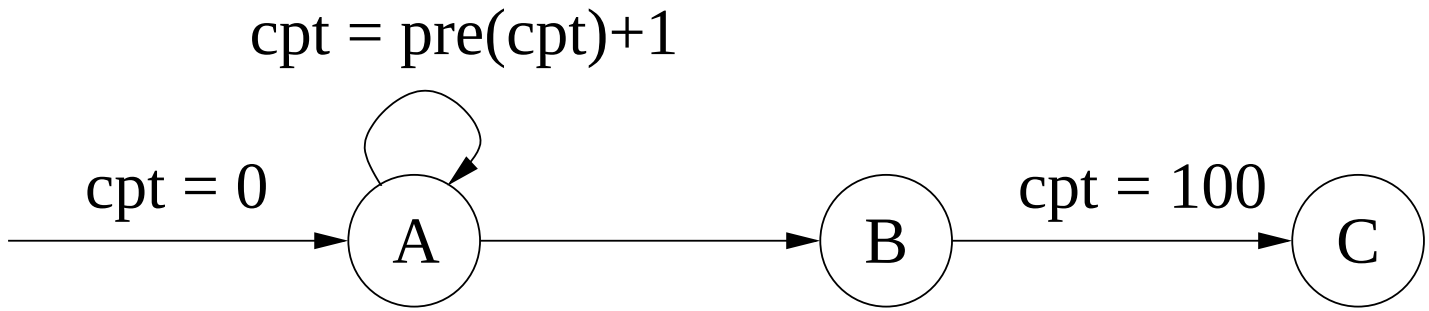
A problematic case:



Here, Lucky is unlikely to produce a successfull sequence !

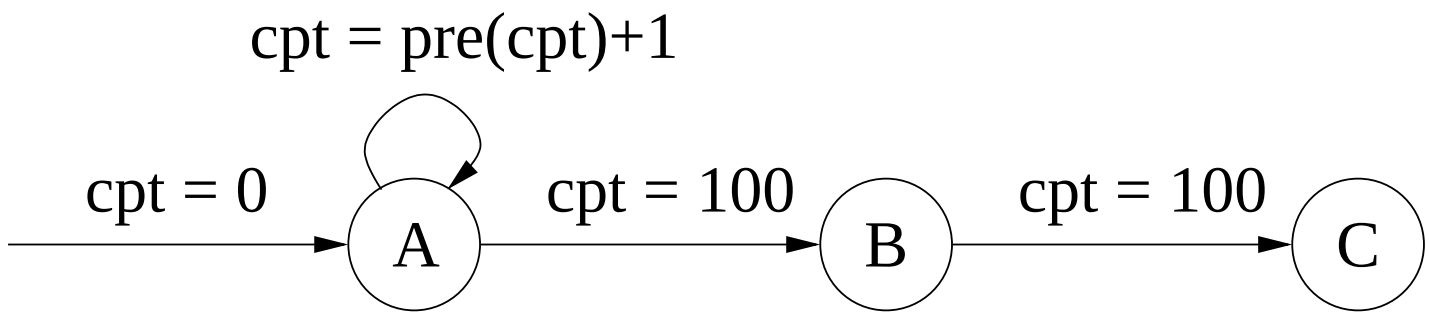
Why restricting the search space is important

A problematic case:



Here, Lucky is unlikely to produce a successfull sequence !

Hopefully, NBac cannot produce such an automaton: the backward analysis gives:



Conclusion

- Lightweight implementation work (connecting tools)
- We have used it for debugging, but application to counter-example generation is straightforward
- More experiments should be conducted
  First experiments are encouraging: slicing is efficient and NBac can often handle the verification problem
- Here, we looked for short sequences
  Other criteria might be interesting (sequences with few input changes between two steps)