Lecture 11 – Partial Programs, Program Repair, and Sketching

PROGRAM ANALYSIS & SYNTHESIS

Eran Yahav

Previously

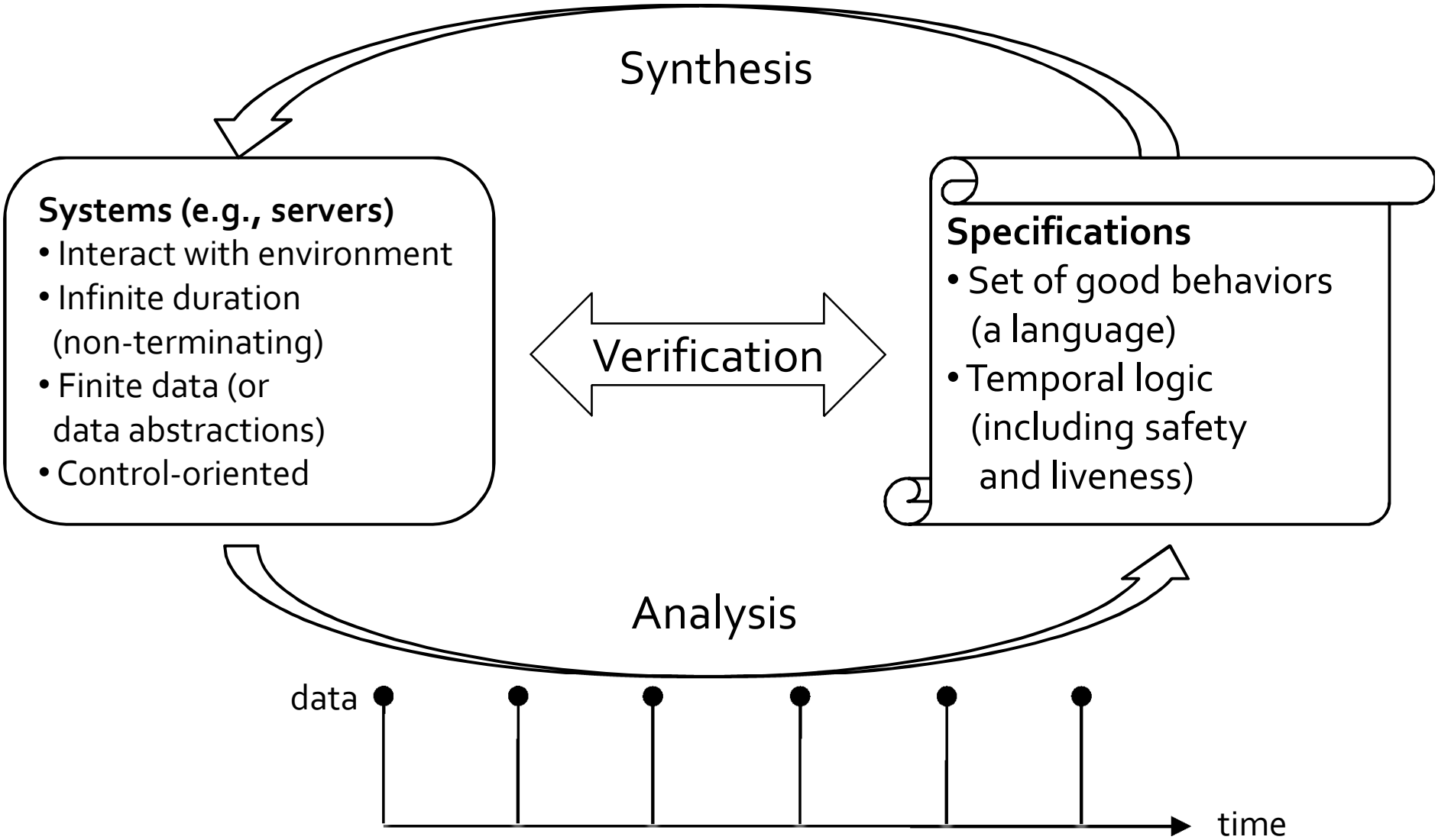
- Synthesis from examples
- SMARTEdit
- String processing in spreadsheet from examples (a little bit)

Today

- Program Repair as a Game
  - angelic non-determinism in the program
- Sketching
  - completion of partial programs using a backing SAT solver

- Acks
  - Program Repair slides from Barbara Jobstmann
  - Sketching slides cannibalized from Armando Solar-Lezama

Reactive Systems



[Z. Manna and A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems., 1992]
[N. Halbwachs. Synchronous Programming of Reactive Systems. 1993]

How to verify a Reactive System?

Core algorithm for linear-time temporal logic:

1. Source code $\rightarrow_{\text{auto/manual}}$ transition system (FSM)
2. Specification $\rightarrow_{\text{auto/manual}}$ monitor violations
3. Check if model has a violating trace
   - product of trans. system and monitor
   - check for exists of a trace in product (emptiness)

$$L(\text{Program}) \subseteq L(\text{Specification})$$

$$L(\text{Program}) \cap L(\neg\text{Specification}) = \emptyset$$

$$L(\text{Program} \times \neg\text{Specification}) = \emptyset$$

Source Code

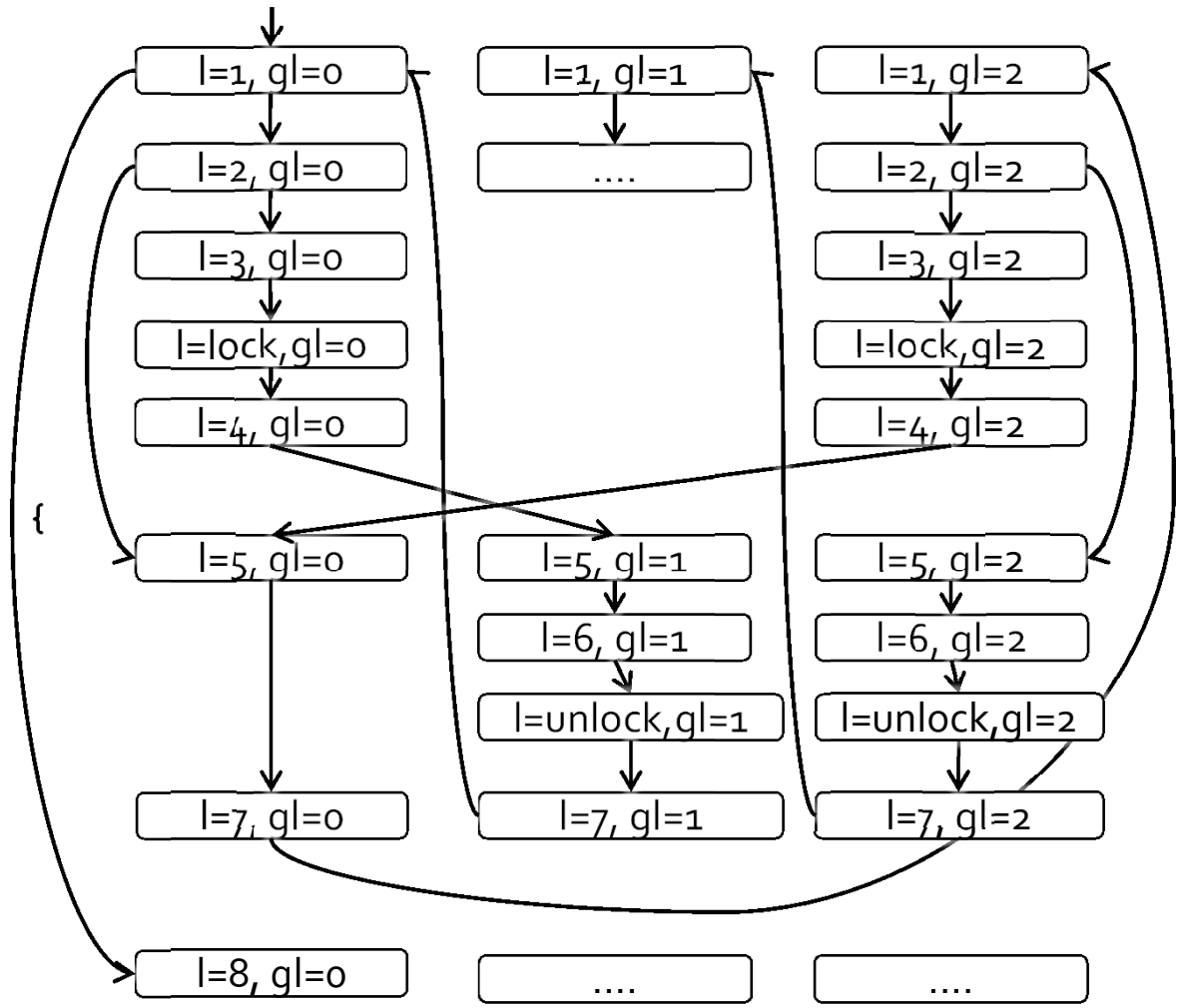
```
   unsigned int got_lock = 0;
   ...
1: while(*) {
     ...
2:   if (*) {
3:     lock();
4:     got_lock++;
     }
     ...
5:   if (got_lock != 0) {
6:     unlock();
     }
7:   got_lock--;
     ...
   }
```

```
lock()
lock:   {LOCK:=1;}
unlock()
unlock: {LOCK:=0;}
```

Step 1: Transition System

```
   int[0,1,2] got_lock = 0;
   ...
1: while(*) {
     ...
2:   if (*) {
3:     lock();
lock:   {LOCK:=1;}
4:     got_lock++;
     }
     ...
5:   if (got_lock != 0) {
6:     unlock();
unlock: {LOCK:=0;}
     }
7:   got_lock--;
     ...
   }
 8:
```

Trans. system variables: line (l), got_lock (gl)



Specification

P1: do not acquire a lock twice
P2: do not call unlock without holding the lock

P1: always(line=lock **implies next**(line!=lock **w-Until** line=unlock))

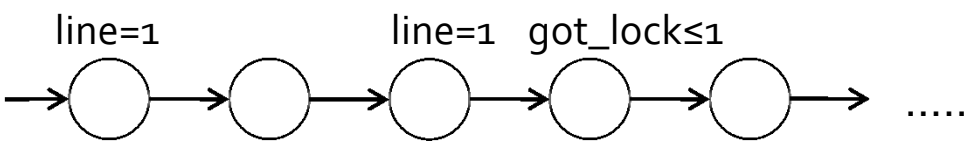
P2: (line!=unlock **w-until** line=lock)) **and**
always(line=unlock **implies**
next(line!=unlock **w-until** line=lock))

Linear-Time Temporal Logic [Pnueli77]

- Syntax:
  - Atomic propositions,
    e.g., line=1, line!=1, got_lock=0
  - Boolean operators: **not**, **and**, **or**, **implies**,…
  - Temporal operators:
    - **next** (ϕ) … ϕ holds in the next step
    - ϕ_1 **until** ϕ_2 … ϕ_1 holds until at some point ϕ_2 holds
- Used in industrial spec languages PSL/SVA
- Can express many interesting properties, e.g., mutual exclusion, deadlock freedom, termination

Linear-Time Temporal Logic

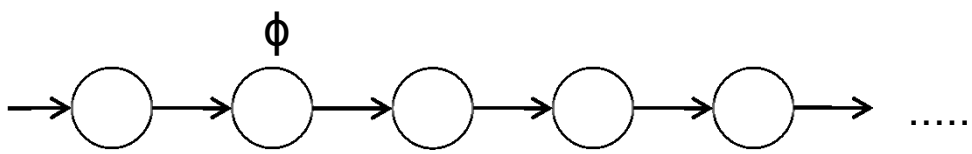
- Semantics
  - defined with respect to infinite traces
  - in each step atomic propositions holds or not
  - E.g., line=1, got_lock≤1



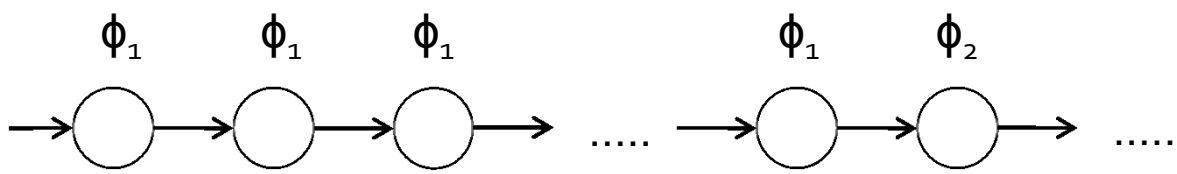
Given a finite set of atomic proposition AP,
a trace (or word) w over AP is an infinite sequence
of truth assignments to AP, i.e., $w \in (2^{AP})^{\omega}$.

Linear-Time Temporal Logic

- Semantics
  - **next** (ϕ)… ϕ holds in the next step



  - ϕ_1 **until** ϕ_2… ϕ_1 holds until at some point ϕ_2 holds



- System S satisfies/models ϕ, if all its behaviors satisfy ϕ

How to verify a Reactive System?

Core algorithm for linear-time temporal logic:

1. Source code $\rightarrow_{\text{auto/manual}}$ transition system (FSM)
2. Specification $\rightarrow_{\text{auto/manual}}$ monitor violations
3. Check if model has a violating trace
   - product of trans. system and monitor
   - check for exists of a trace in product (emptiness)

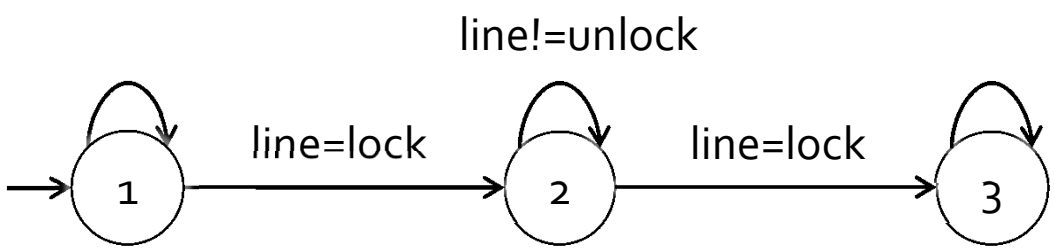
Step 2: Monitor for Violations

P1: always(line=lock **implies**

next(line!=lock **w-until** line=unlock))

= **not eventually**(line=lock **and**

next(line!=unlock **until** line=lock))



s1: non-deterministic choice s2: no edge with line=unlock

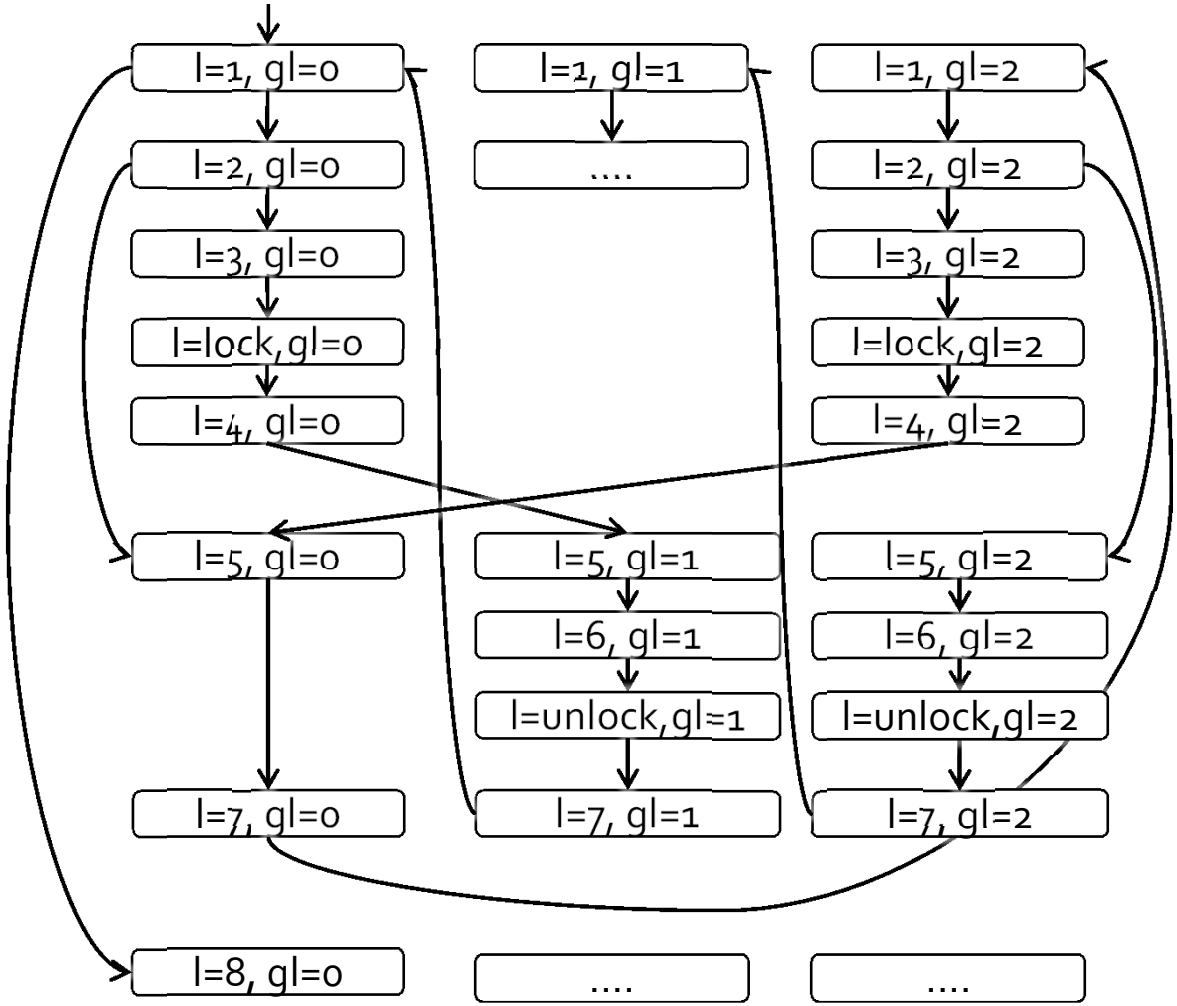
Automaton accepts trace/behavior if a green state is visited infinitely often (Büchi)

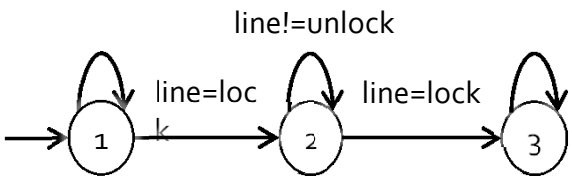
Why do we track bad and not good behaviors?

$L(S) \subseteq L(\phi)$: forall w: $w \in L(S) \rightarrow w \in L(\phi)$

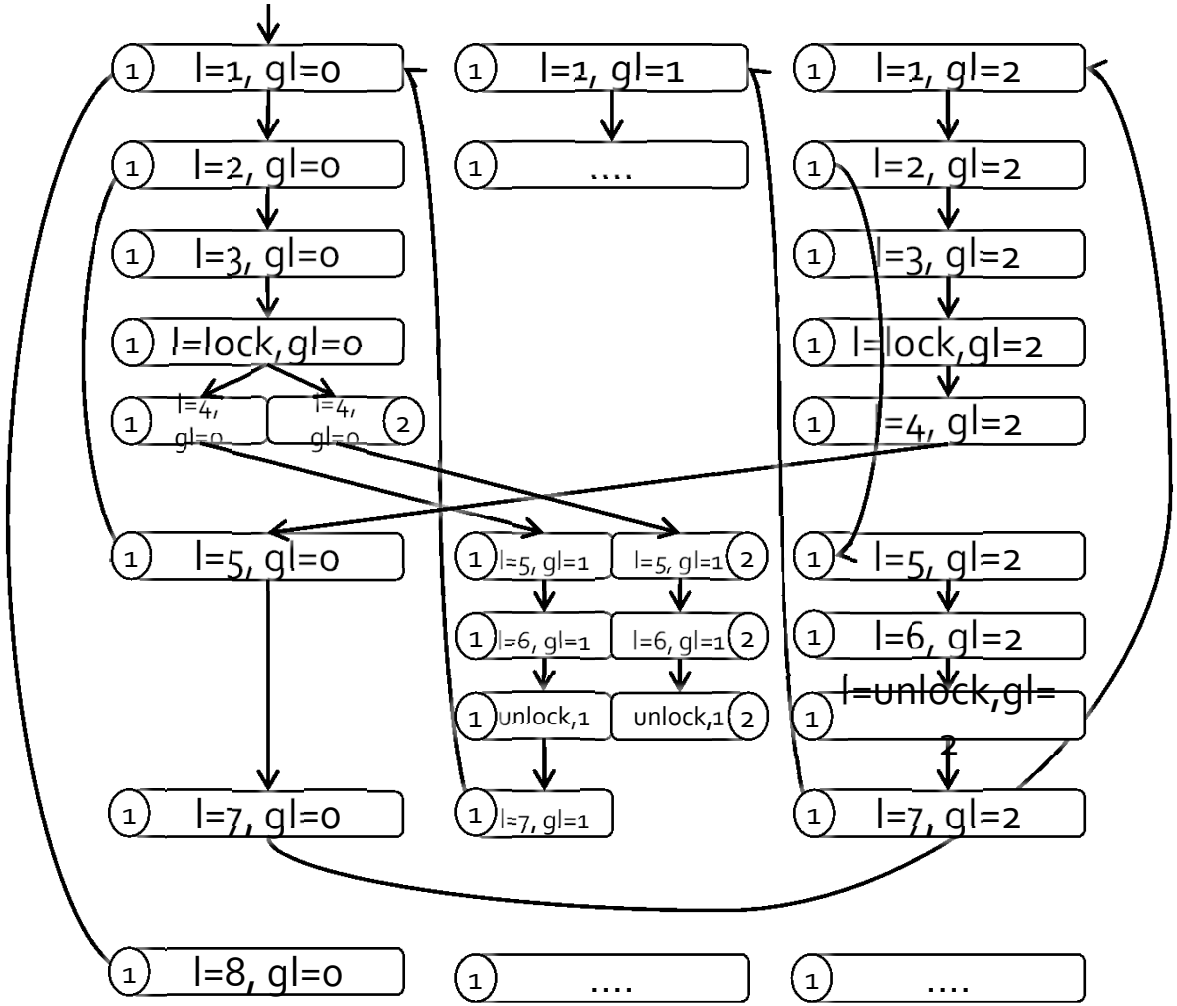
$\neg$ exists w: $w \in L(S) \wedge w \in L(\neg\phi)$

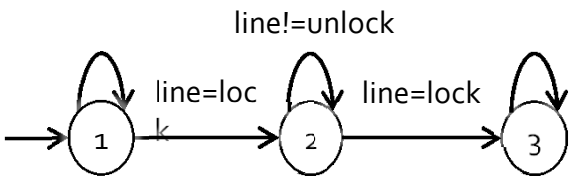
Step 3: Product



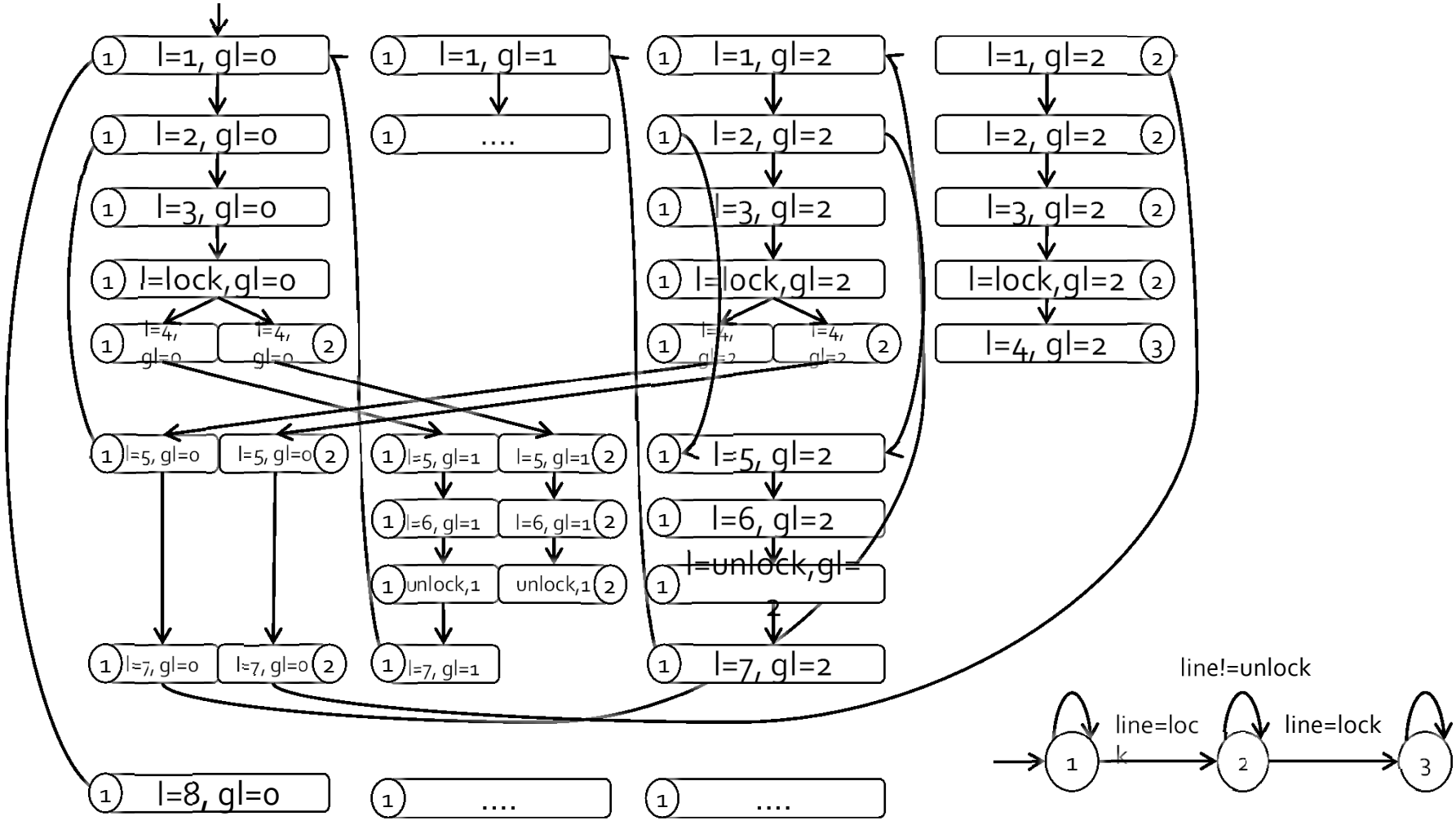


Step 3: Product



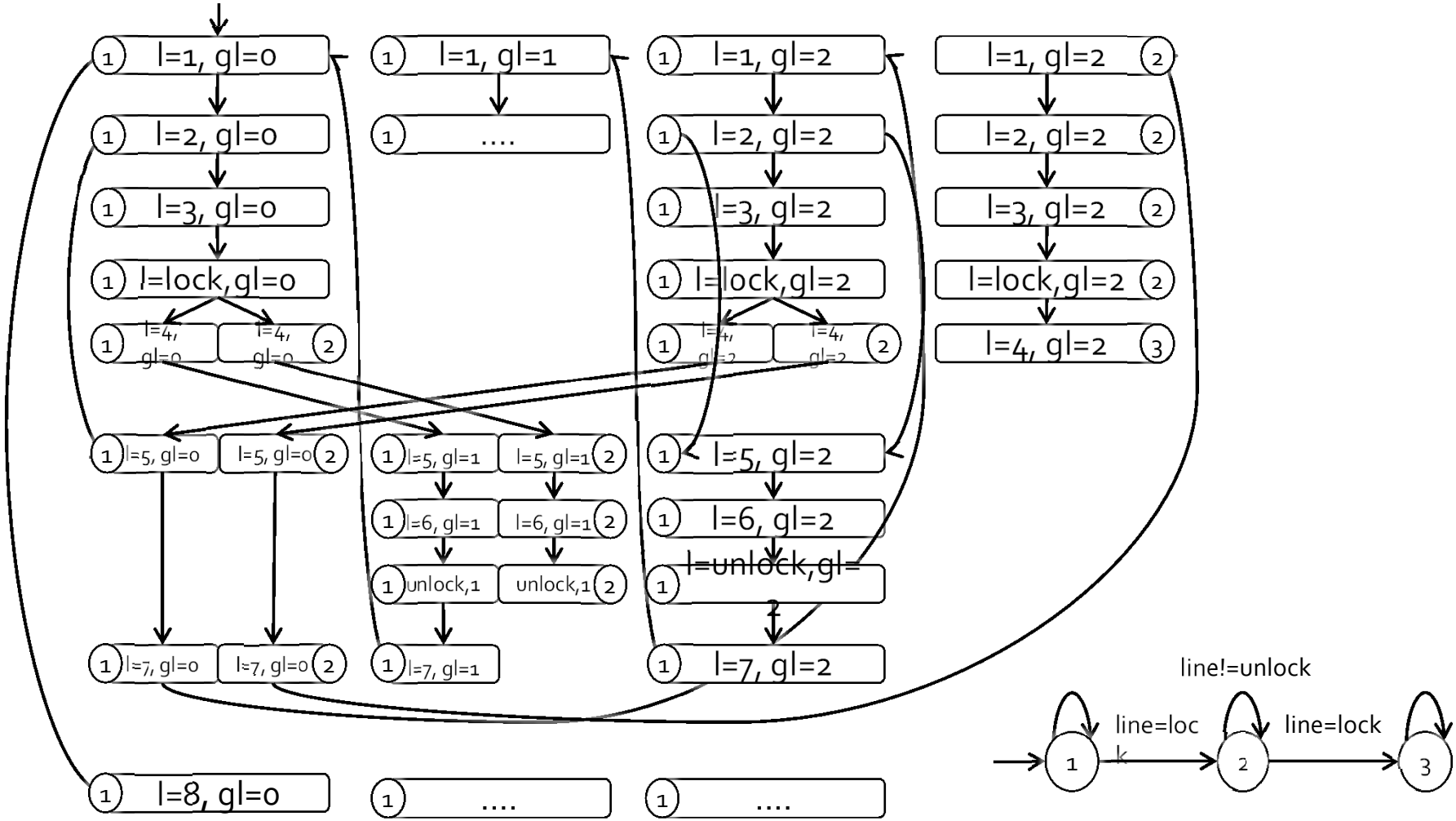


Step 3: Product



Recall, we want to show a violation:

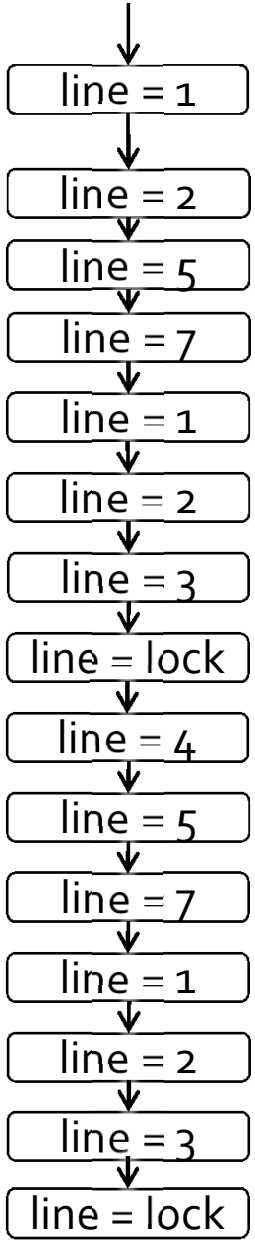
Step 3: Product



Recall, we want to show a violation:
non-determinism in transition system and in monitor pull in the **same direction**
(both can be used to violate property)

Source Code

```
   int[0,1,2] got_lock = 0;
   ...
1: while(*) {
      ...
2:    if (*) {
3:       lock();
lock:    {LOCK:=1;}
4:       got_lock++;
      }
      ...
5:    if (got_lock != 0) {
6:       unlock();
unlock:  {LOCK:=0;}
      }
7:    got_lock--;
      ...
   }
8:
```



How to verify a Reactive System?

Core algorithm for linear-time temporal logic:

1. Source code $\rightarrow_{\text{auto/manual}}$ transition system (FSM)
2. Specification $\rightarrow_{\text{auto/manual}}$ monitor violations
3. Check if model has a violating trace
   - product of trans. system and monitor
   - check for exists of a trace in product (emptiness)

But how to repair it?

How to repair a Reactive System?

1. Add freedom (choice for the system, allowed ways to modify system)
2. Source code $\rightarrow_{a/m}$ transition system (game)
3. Specification $\rightarrow_{a/m}$ monitor acceptance
4. Check if we can find system choices s.t. model is accepted by monitor
   - product of trans. system and monitor
   - search for winning strategy in game

Step 1: Freedom

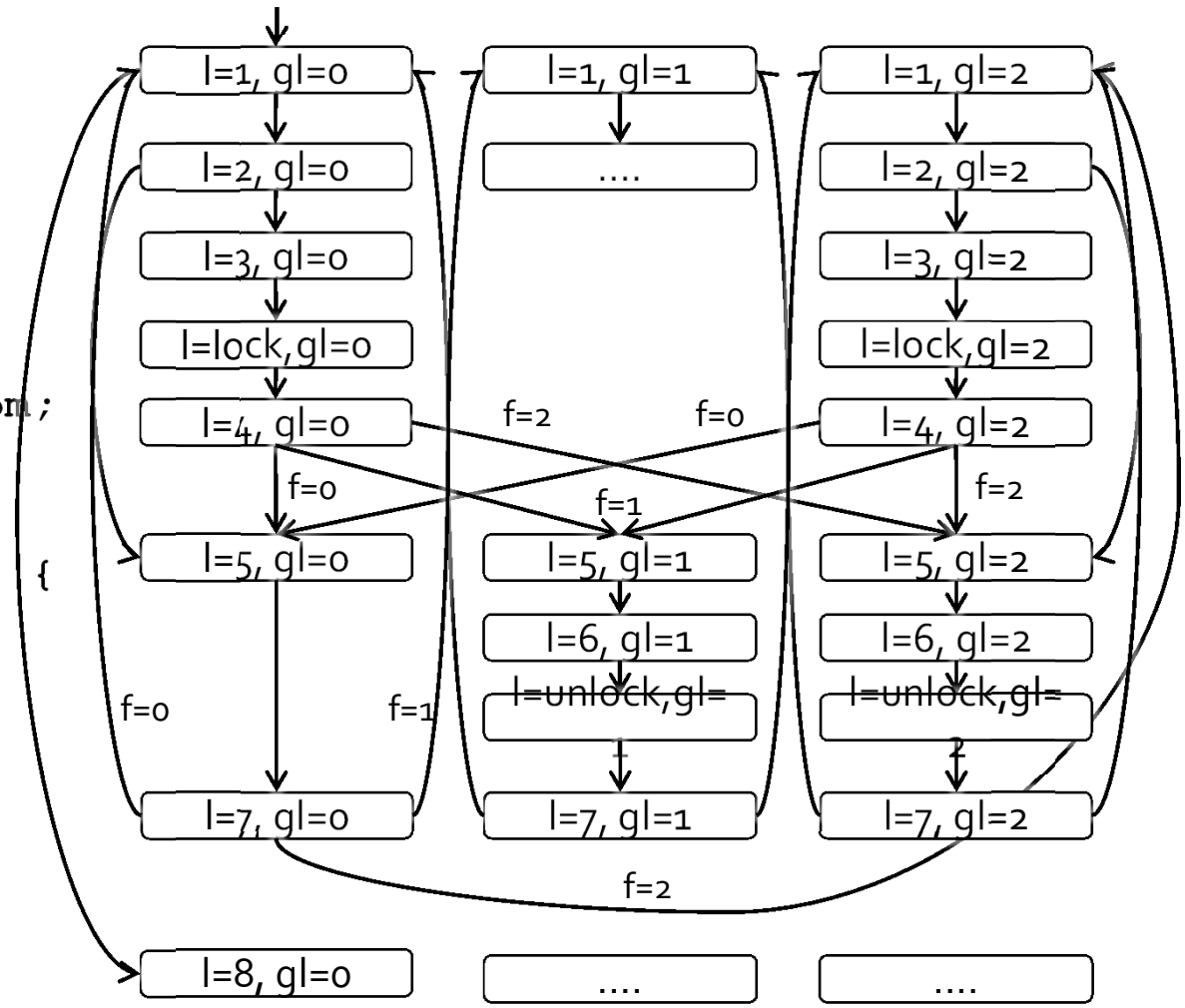
```
   int[0,1,2] got_lock = 0;
   int[0,1,2] freedom;
   ...
1: while(*) {
     ...
2:   if (*) {
3:     lock();
lock:   {LOCK:=1;}
4:     got_lock:=freedom;
     }
     ...
5:   if (got_lock != 0) {
6:     unlock();
unlock: {LOCK:=0;}
     }
7:   got_lock:=freedom;
     ...
   }
 8:
```

(We can also extend to fault localization)

Step 2: Game

```
   int[0,1,2] got_lock = 0;
   int[0,1,2] freedom;
   ...
1: while(*) {
     ...
2:   if (*) {
3:     lock();
lock:   {LOCK:=1;}
4:     got_lock:=freedom;
     }
     ...
5:   if (got_lock != 0) {
6:     unlock();
unlock: {LOCK:=0;}
     }
7:   got_lock:=freedom;
     ...
   }
 8:
```

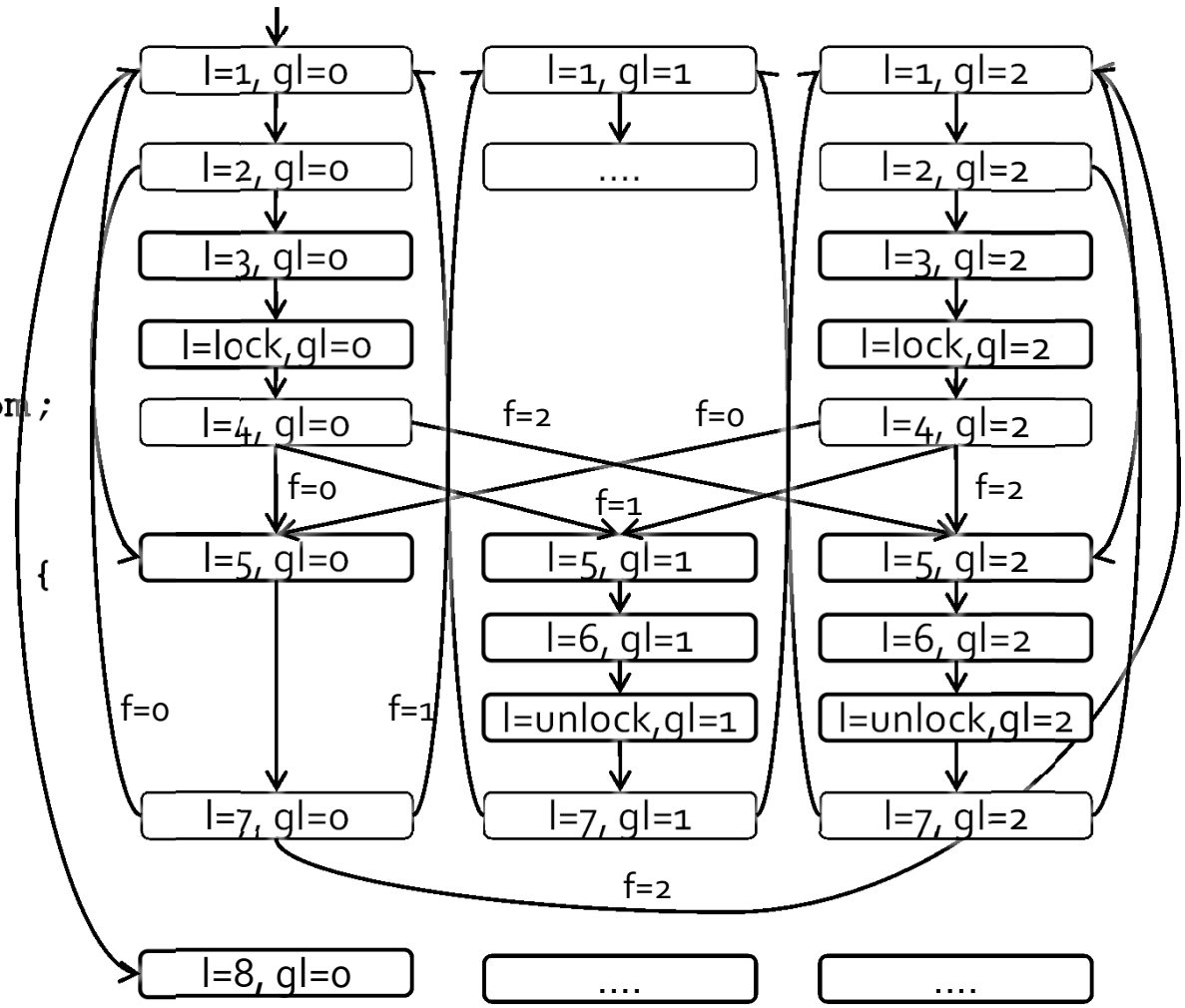
Trans. system variables: line (l), got_lock (gl)



Step 2: Game

```
   int[0,1,2] got_lock = 0;
   int[0,1,2] freedom;
   ...
1: while(*) {
     ...
2:   if (*) {
3:     lock();
lock:   {LOCK:=1;}
4:     got_lock:=freedom;
     }
     ...
5:   if (got_lock != 0) {
6:     unlock();
unlock: {LOCK:=0;}
     }
7:   got_lock:=freedom;
     ...
   }
 8:
```

Trans. system variables: line (l), got_lock (gl)

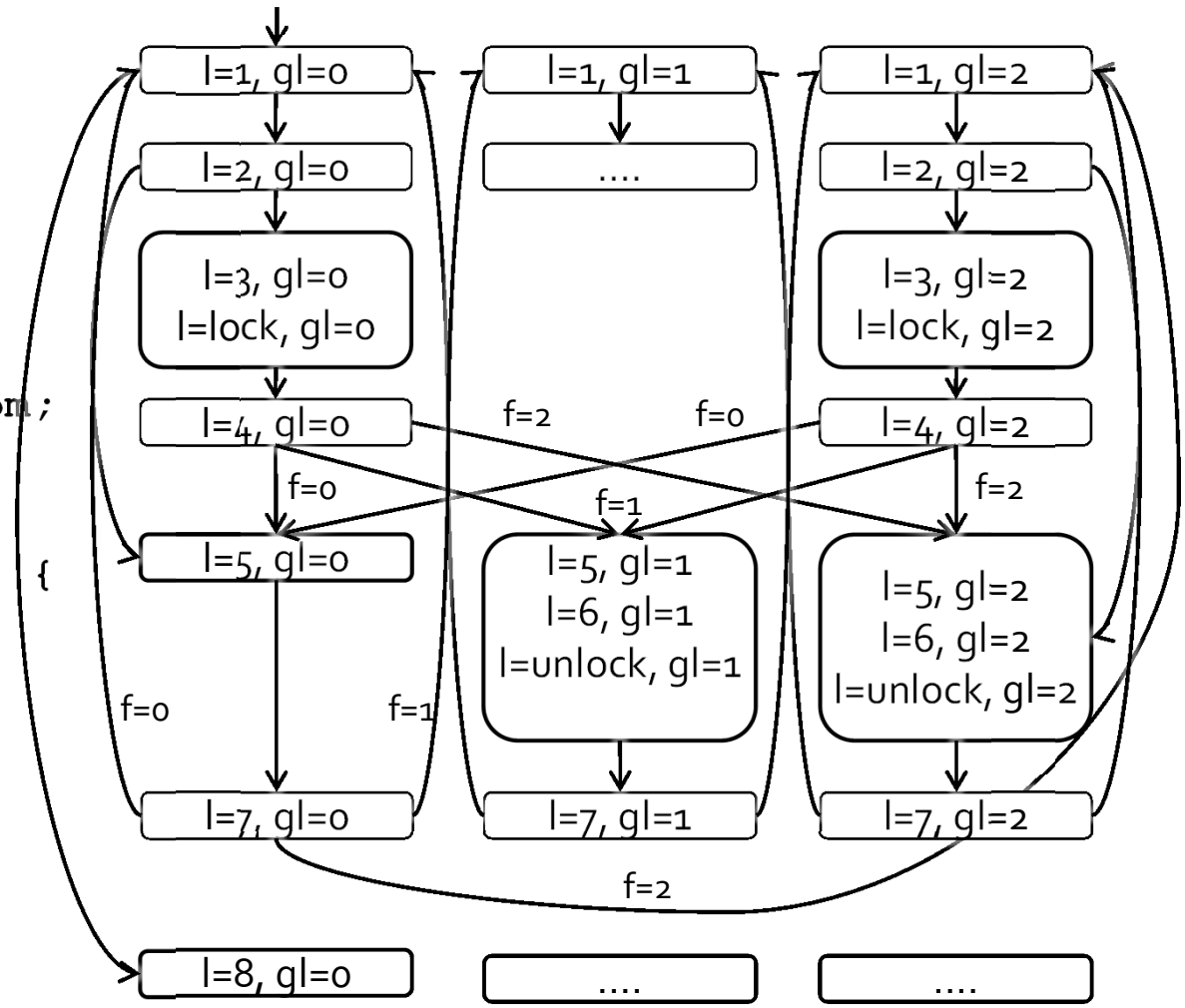


Two types of non-determinism!

Step 2: Game

```
   int[0,1,2] got_lock = 0;
   int[0,1,2] freedom;
   ...
1: while(*) {
     ...
2:   if (*) {
3:     lock();
lock:   {LOCK:=1;}
4:     got_lock:=freedom;
     }
     ...
5:   if (got_lock != 0) {
6:     unlock();
unlock: {LOCK:=0;}
     }
7:   got_lock:=freedom;
     ...
   }
 8:
```

Trans. system variables: line (l), got_lock (gl)

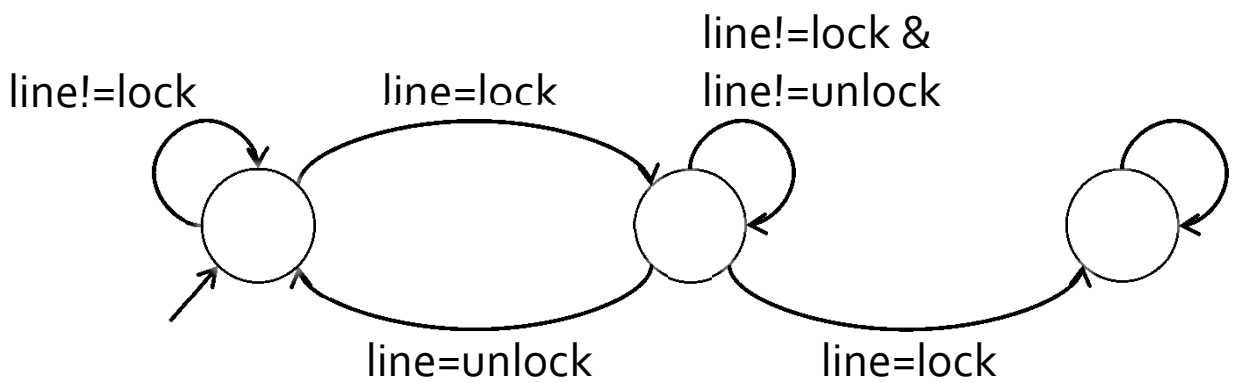


Two types of non-determinism!

Step 3: Monitor for Acceptance

P1: always(line=lock **implies**

next(line!=lock **w-until** line=unlock))

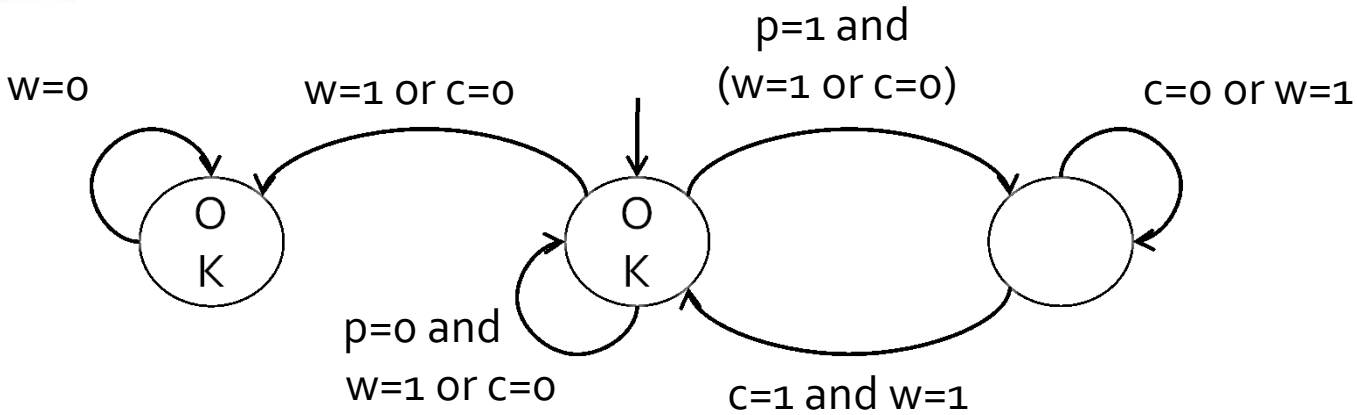


Since game has two types of non-determinism, we need to be careful with non-determinism in monitor.

Problem with Nondeterminism

- Coffee machine is correct if there is no water or if button is pressed machine serves coffee:

eventually always(**not** water) **or**
always(pressed **implies eventually** coffee)
and
always(**not** water **implies not** coffee)

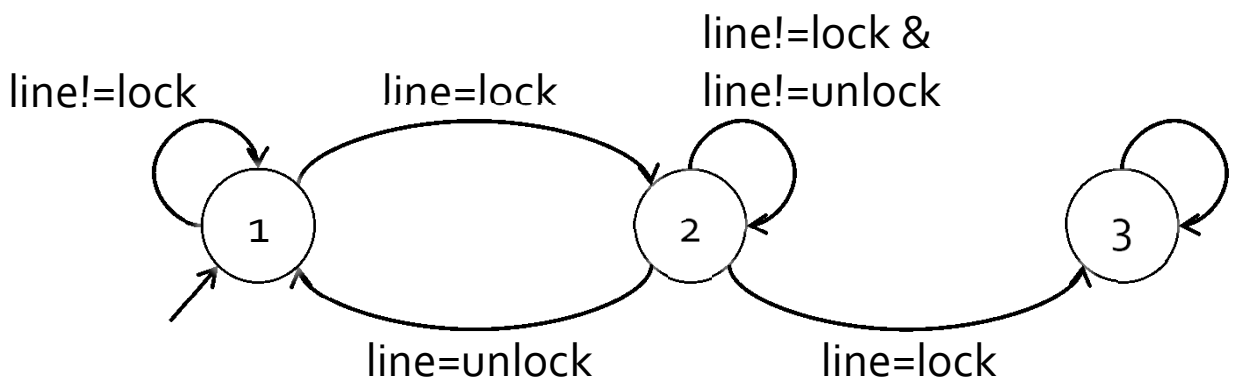


(Coffee machine wins if it visits a green state infinitely often)

Step 3: Det. Monitor for Acceptance

P1: always(line=lock **implies**

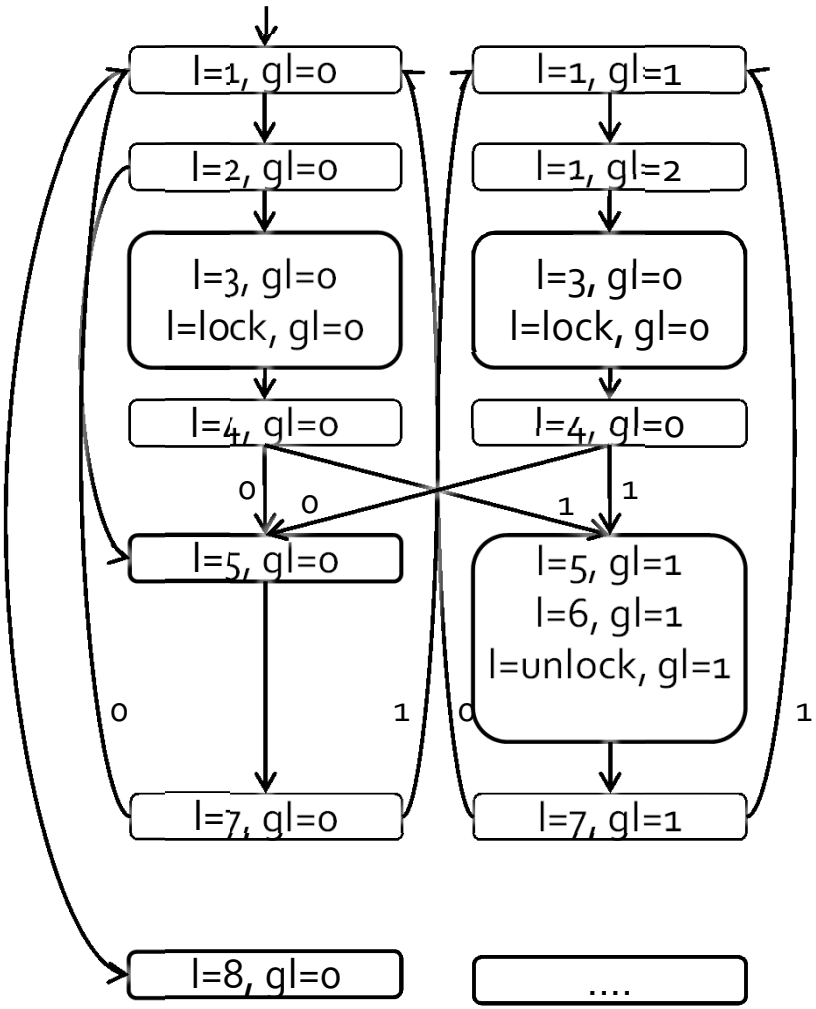
next(line!=lock **w-until** line=unlock))



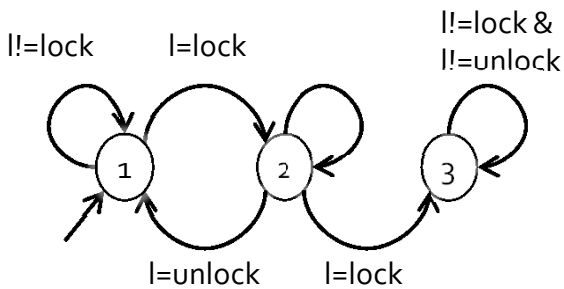
Classical approach: make it deterministic
(more powerful acceptance required)

Step 3: Product

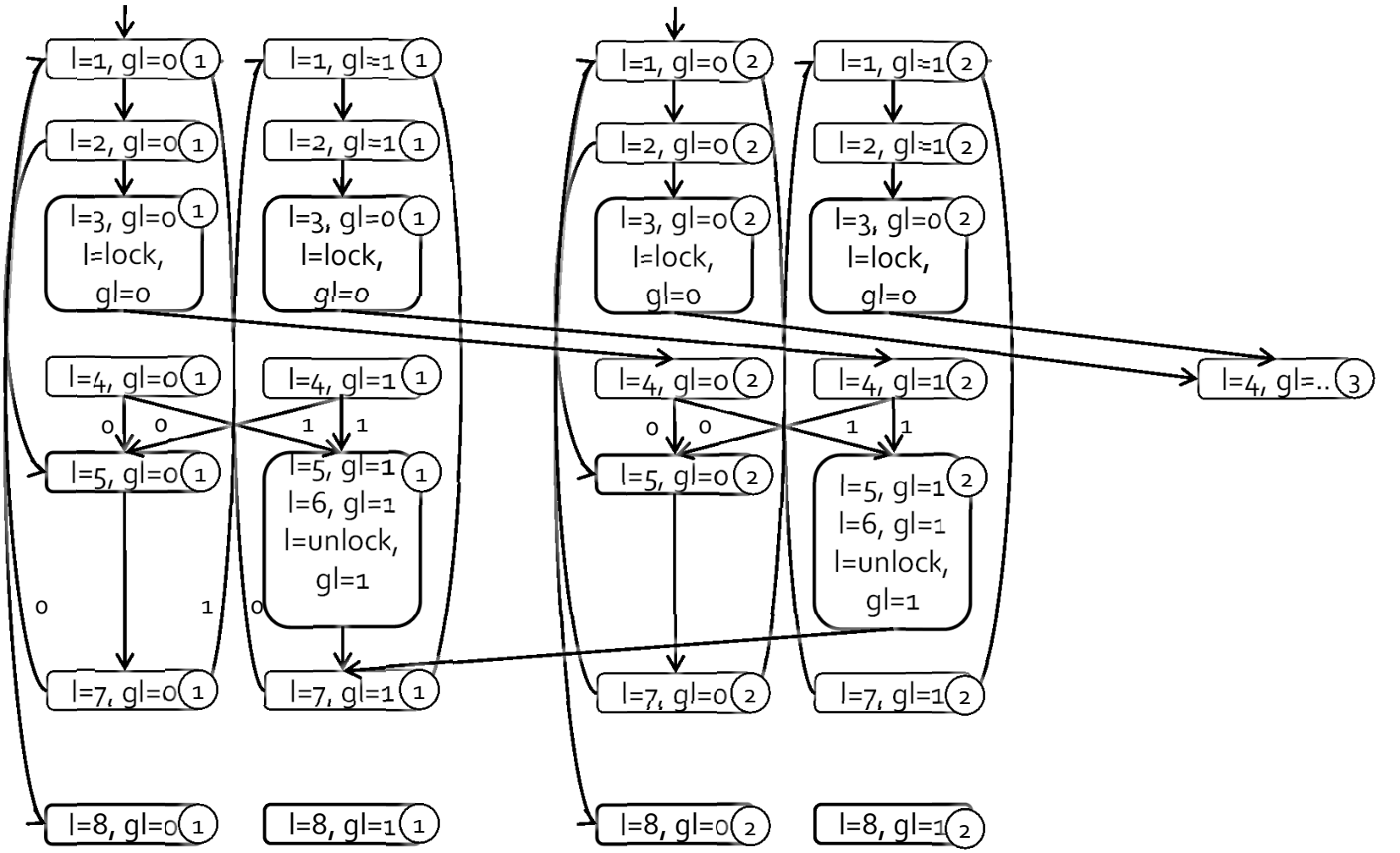
TS for got_lock in {0,1}

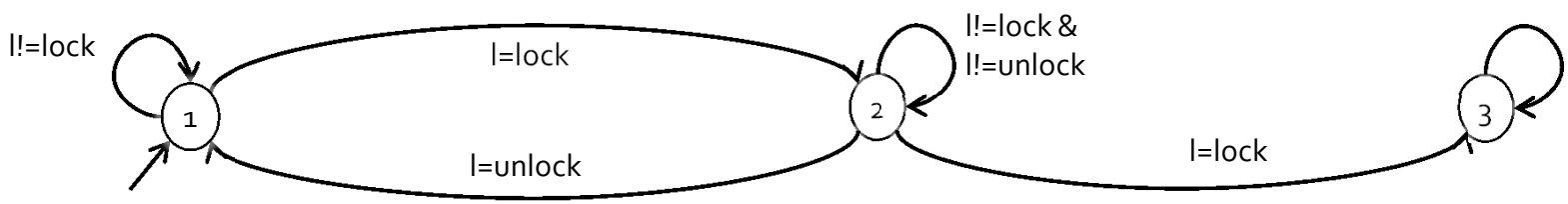


Deterministic automaton

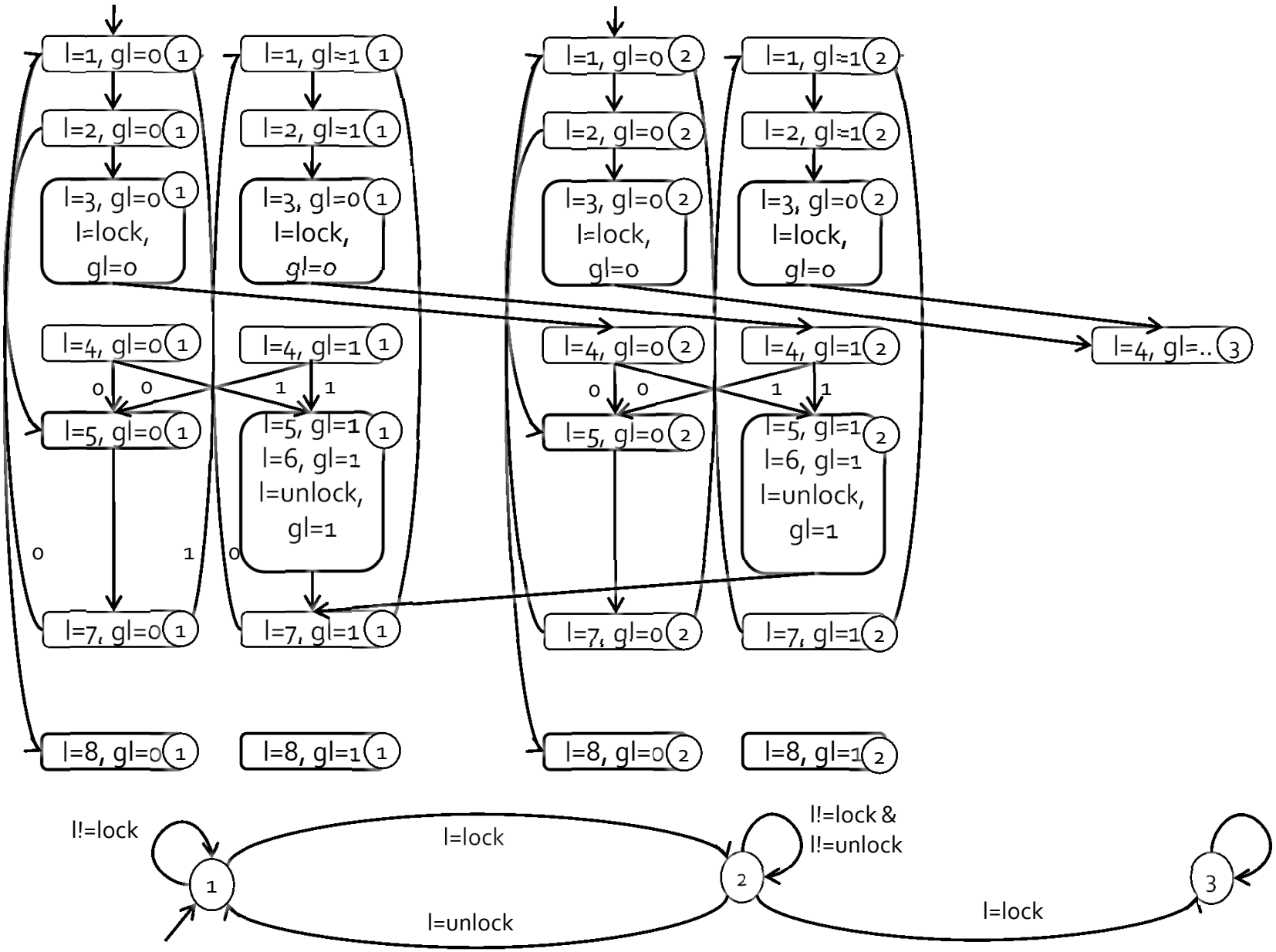


Step 3: Produce

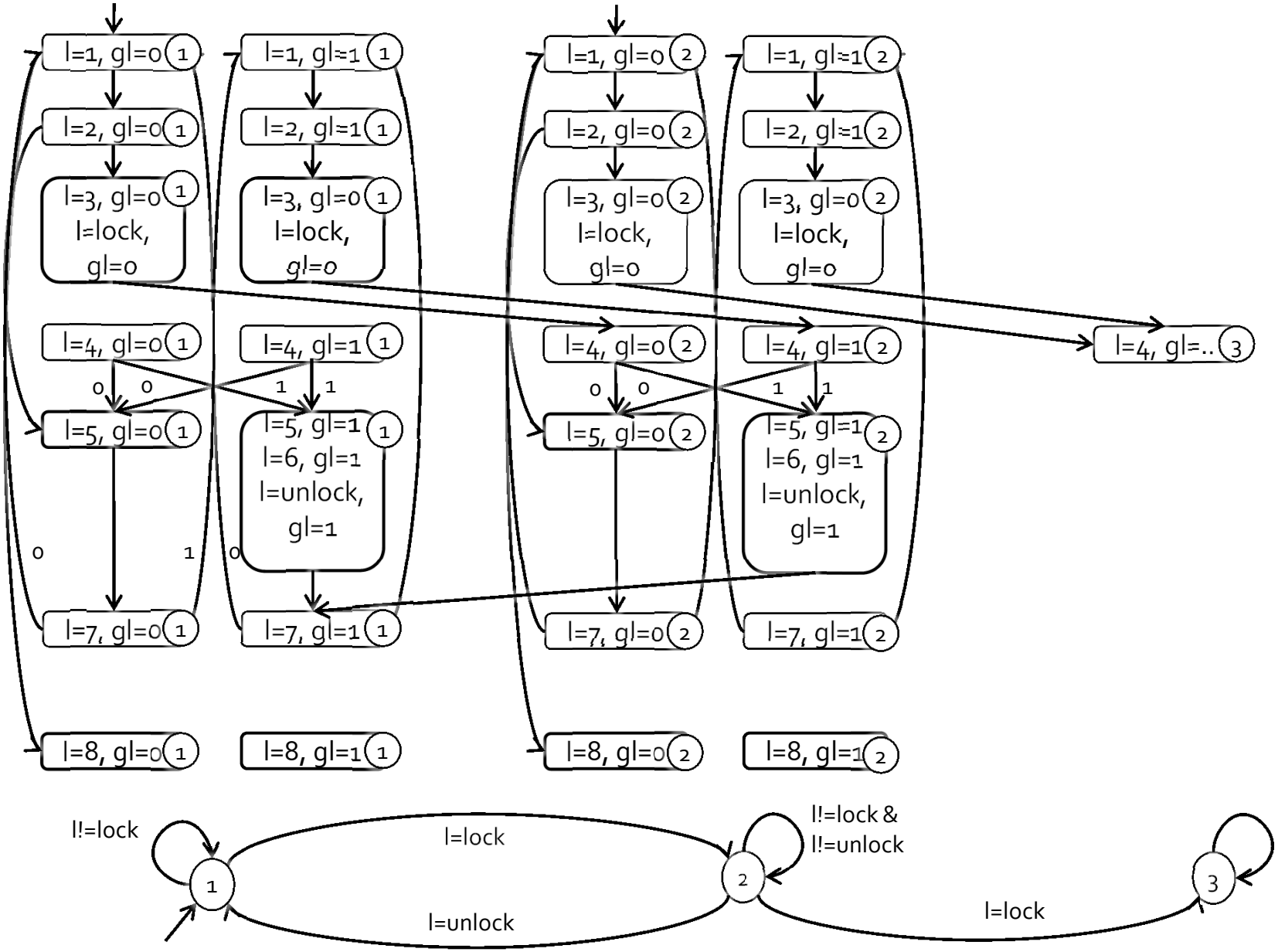




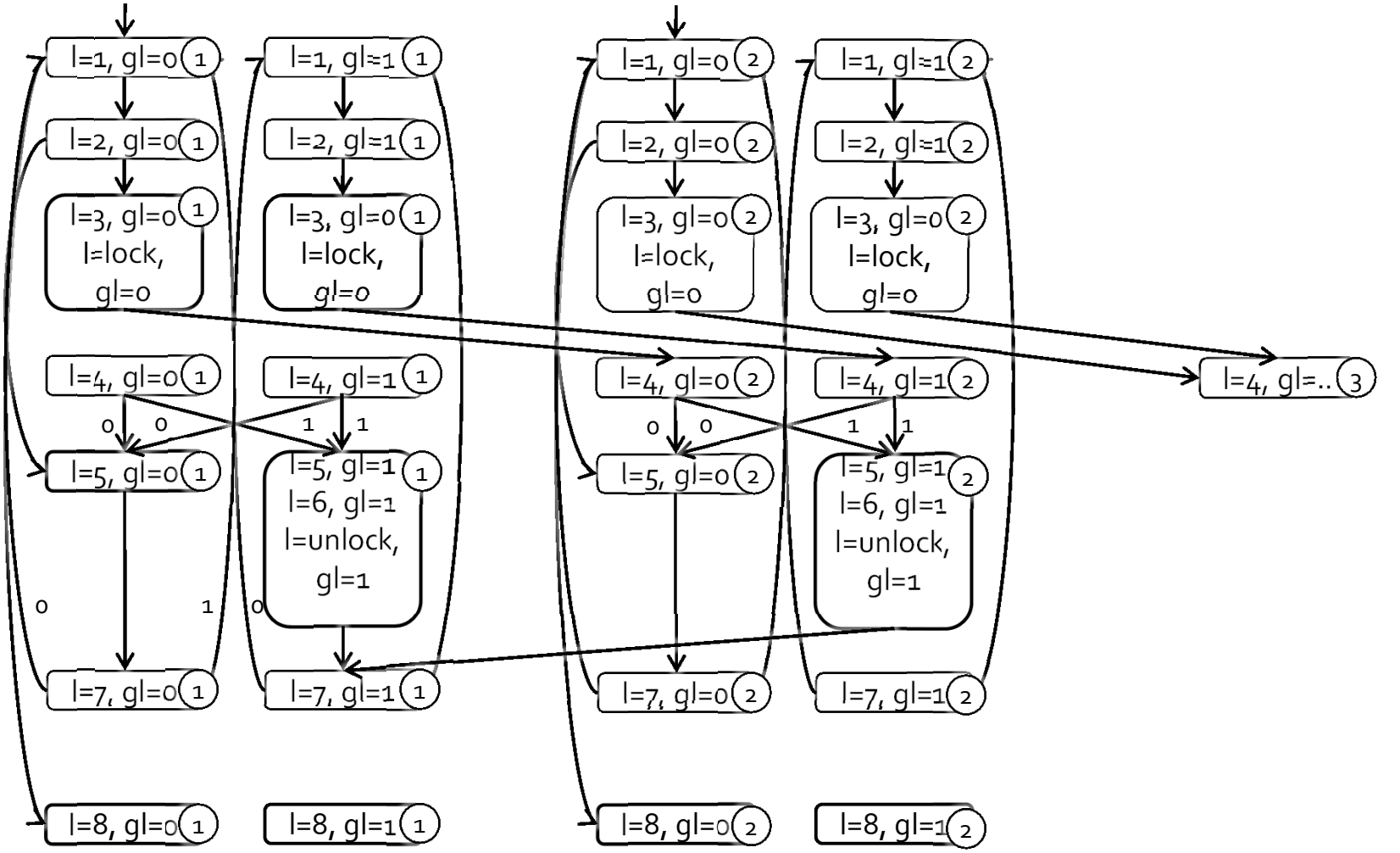
Step 4: Winning States

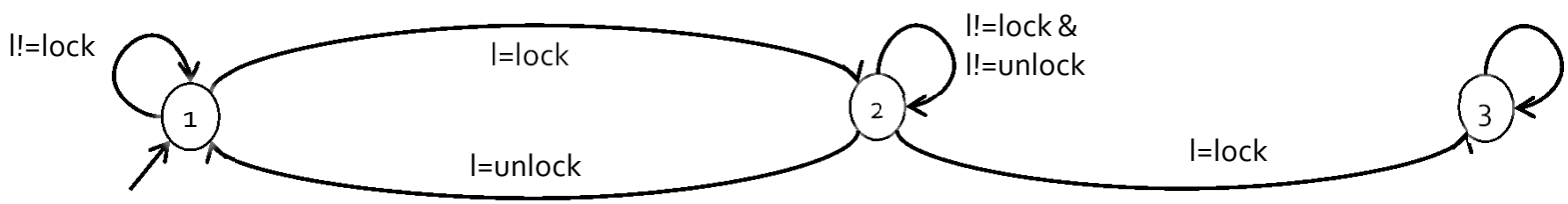


Step 4: Winning States

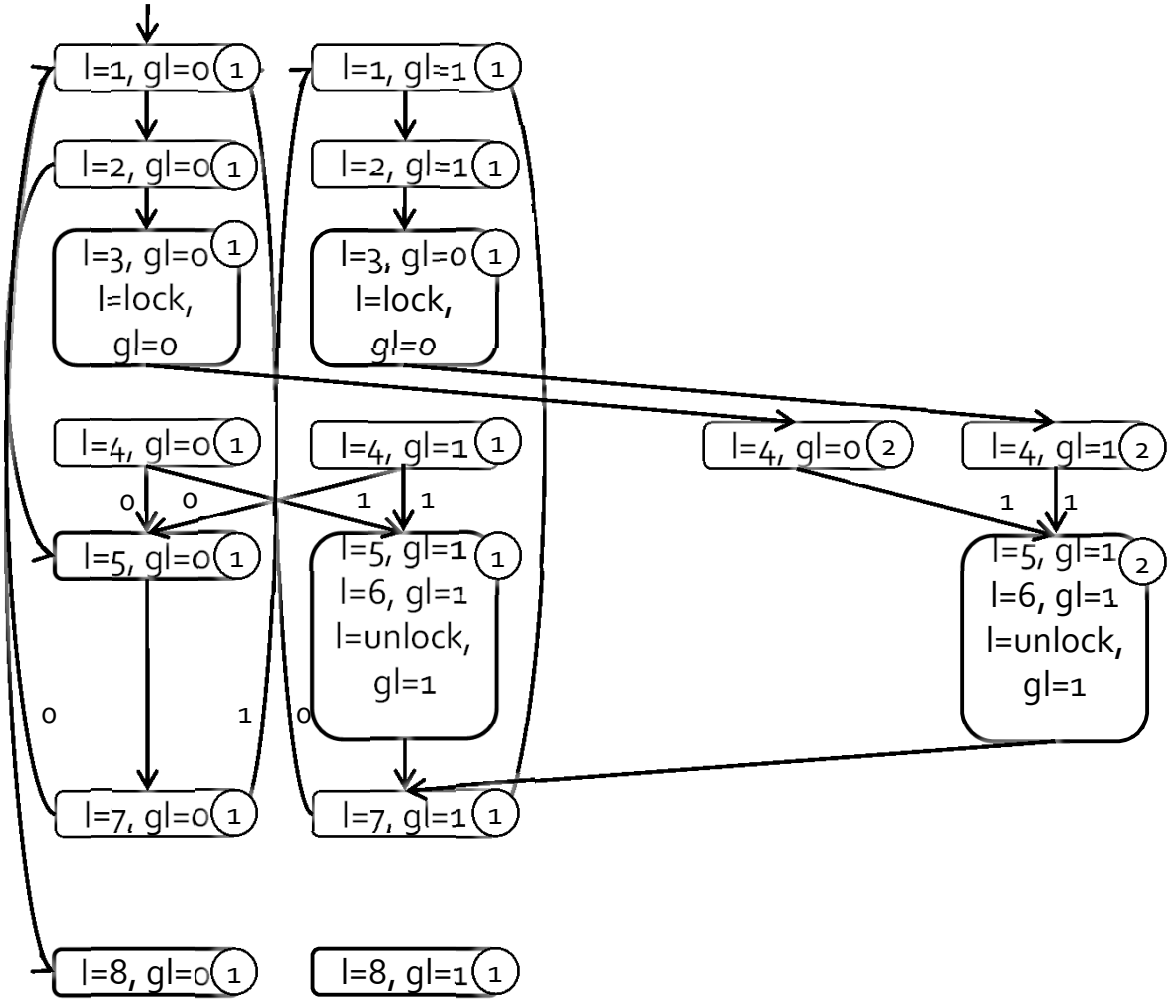


Step 4: Winning States

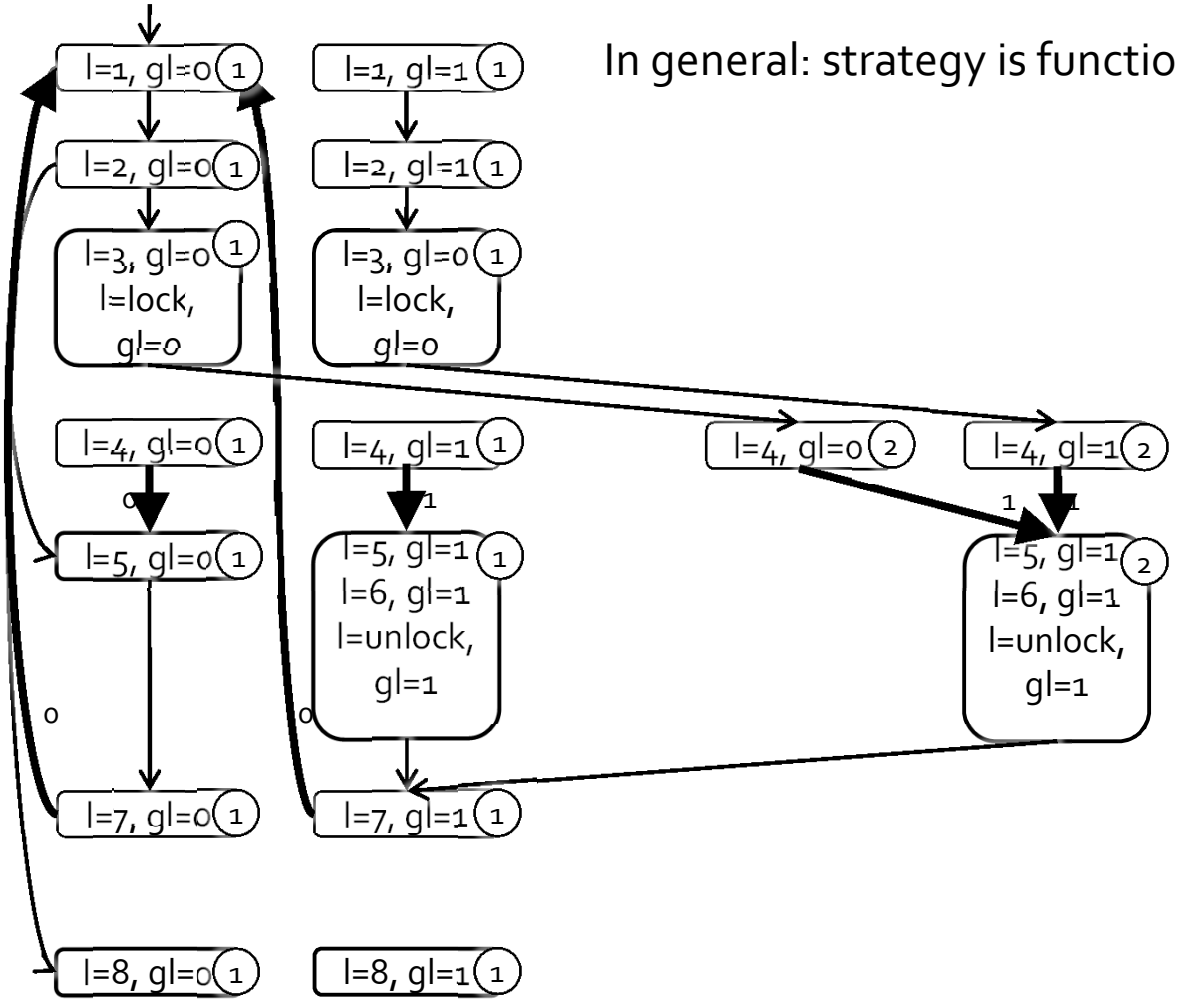




Step 4: Winning States



Step 4: Winning Strategy



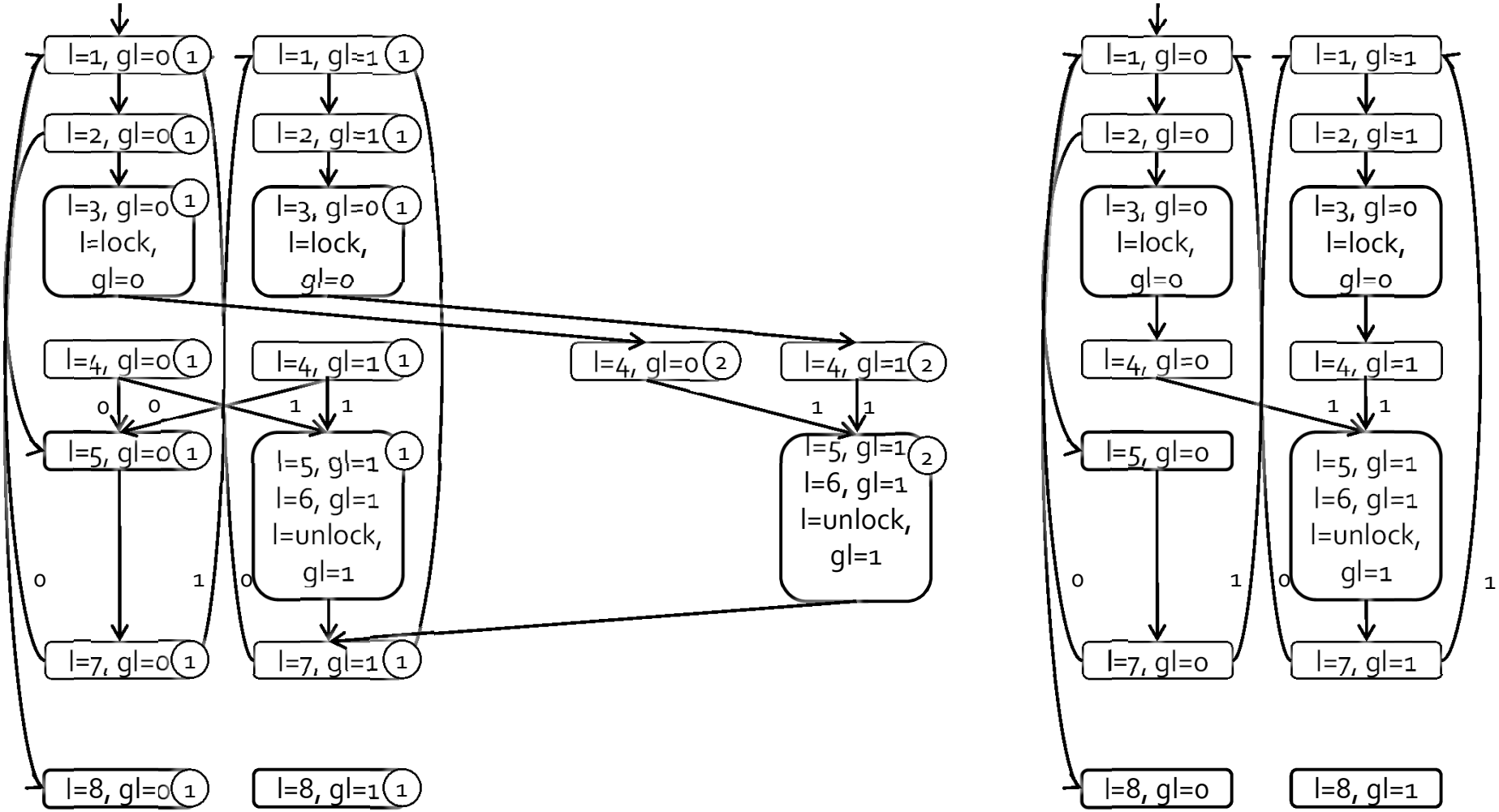
In general: strategy is function of program and monitor state

Strategy to Repair:

if (l=4 & gl=0 & s=1) freedom:=0
if (l=4 & gl=1 & s=1) freedom:=1
if (l=4 & gl=0 & s=0) freedom:=1
if (l=7 & gl=0 & s=1) freedom:=0
if (l=7 & gl=1 & s=1) freedom:=0
..
freedom := f(l,gl,s)
if (line=4)
    freedom := (gl=1) | (s=2)
if (line=7)
    freedom := 0

What we actually do: merge states before picking the strategy

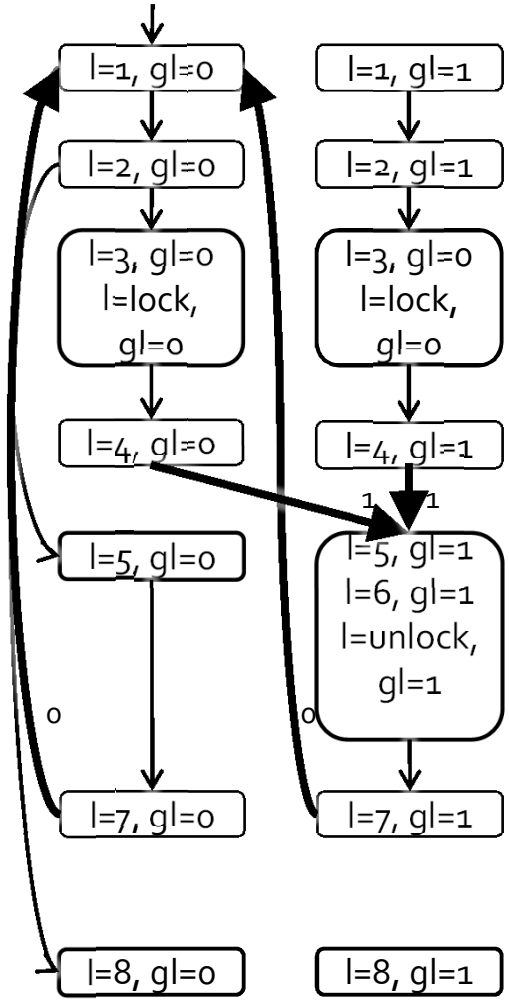
Step 4: Winning Strategy



Step 4: Winning Strategy

(line=4): freedom= 1

(line=7): freedom= 0



Repaired Program

```
   unsigned int got_lock = 0;
   ...
1: while(*) {
     ...
2:   if (*) {
3:     lock();
4:     got_lock = 1;
     }
     ...
5:   if (got_lock != 0){
6:     unlock();
     }
7:   got_lock = 0;
     ...
   }
```

```
lock() {
lock: LOCK:=1;}
unlock(){
unlock: LOCK:=0;}
```

Specification

P1: do not acquire a lock twice

P2: do not call unlock without holding the lock

P1: always(line=lock **implies**
next(line!=lock **w-until** line=unlock))

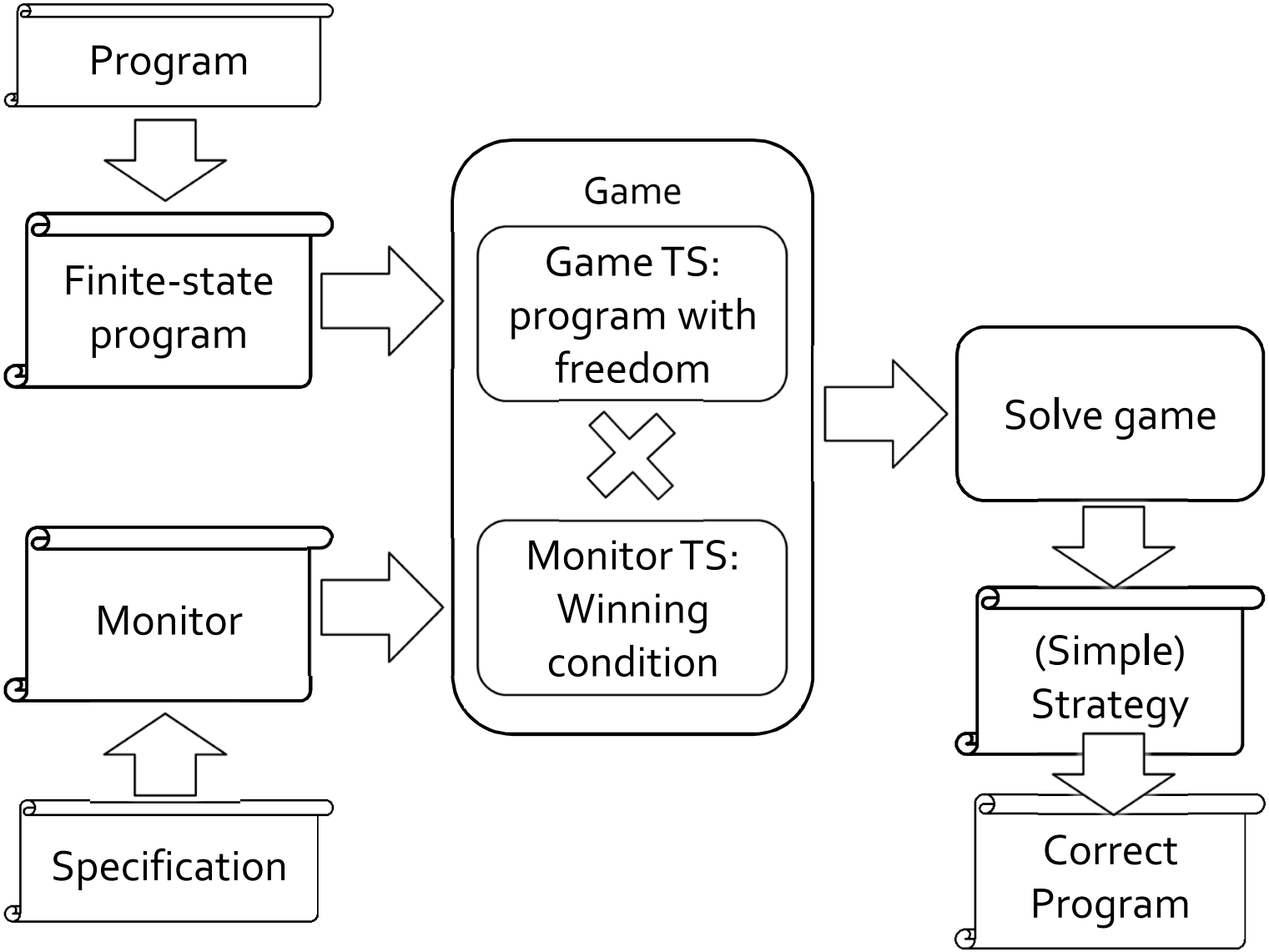
P2: (line!=unlock **w-until** line=lock)) **and**
always(line=unlock **implies**
next(line!=unlock **w-until** line=lock))

(slide adapted with permission from Barbara Jobstmann)

Recap: How to Repair a Reactive System?

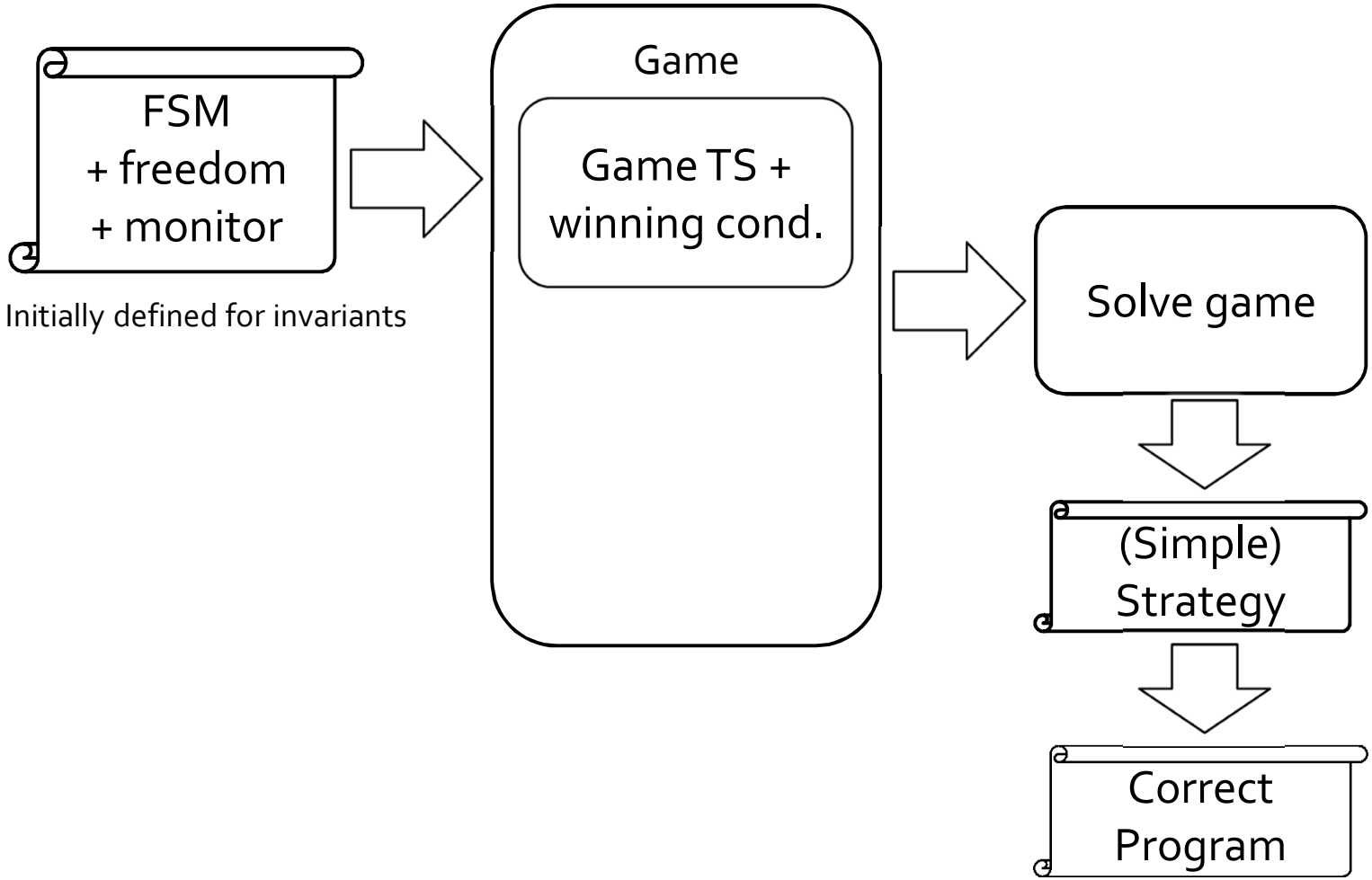
1. Add freedom
   - choice for the system, space of permitted modifications to the system
2. Source code → transition system (**game**)
   - non-determinism in the program (demonic)
   - non-determinism in permitted modification (angelic)
3. Specification → monitor acceptance
4. Check if we can find system choices s.t. model is accepted by monitor
   - product of trans. system and monitor
   - search for winning strategy in game

Program Repair



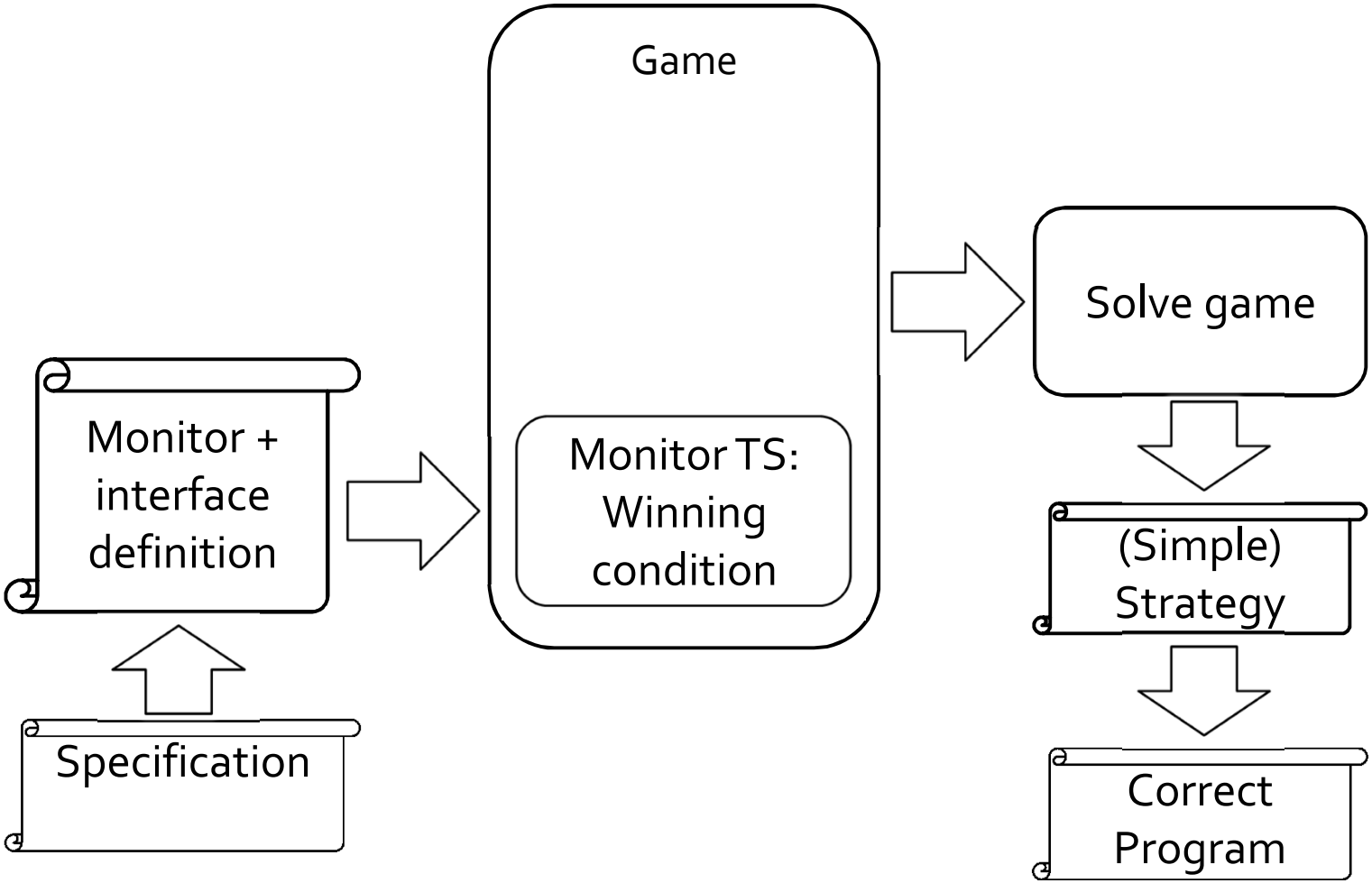
with Bloem, Griesmayer, Staber in CAV 2005, CHARME 2005 (+ext to fault localization)

Classical Controller Synthesis



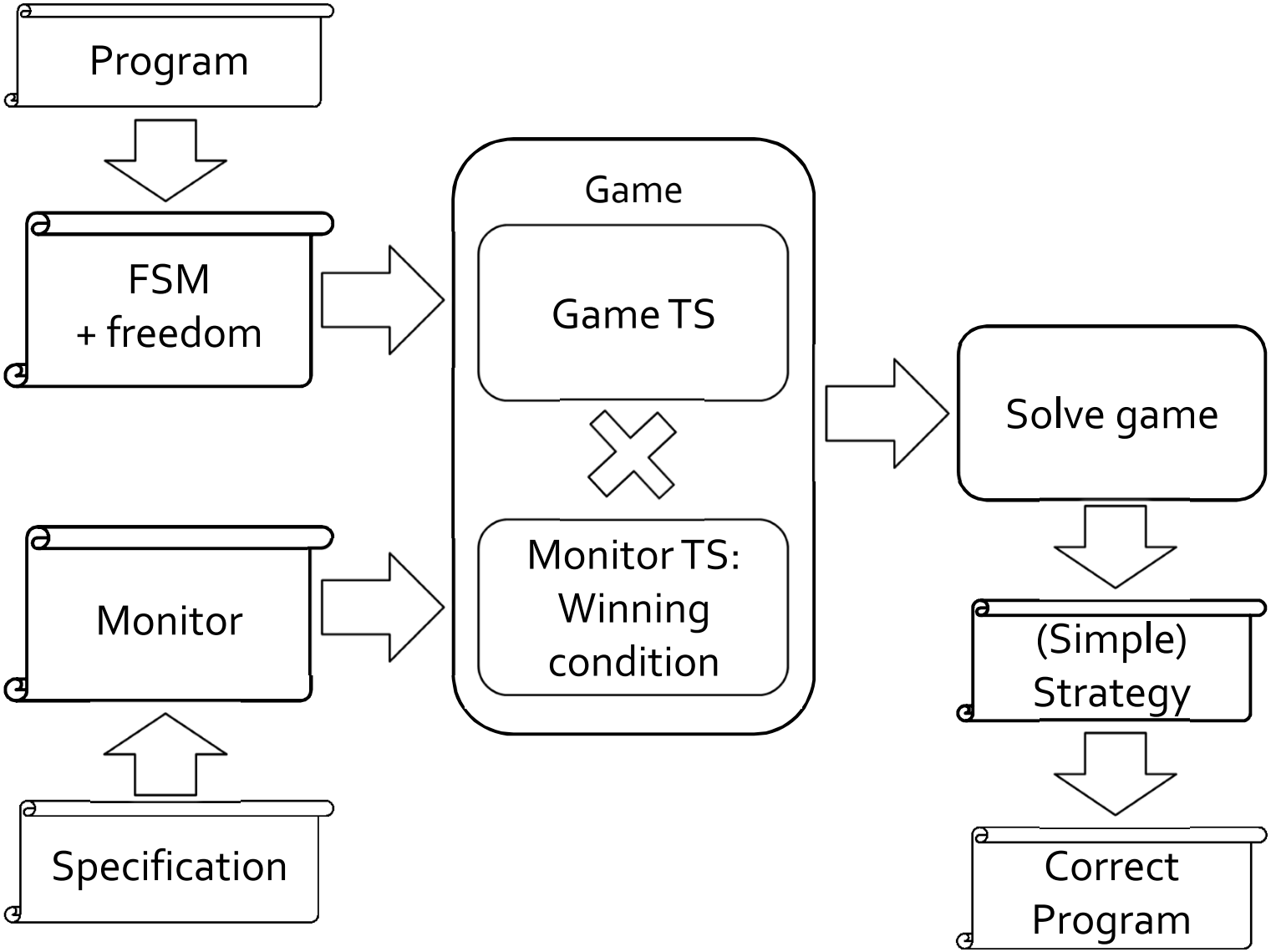
Ramadge, Wonham 87, Book by Cassandras, Lafortune 99/07

Synthesis from Temporal Logics



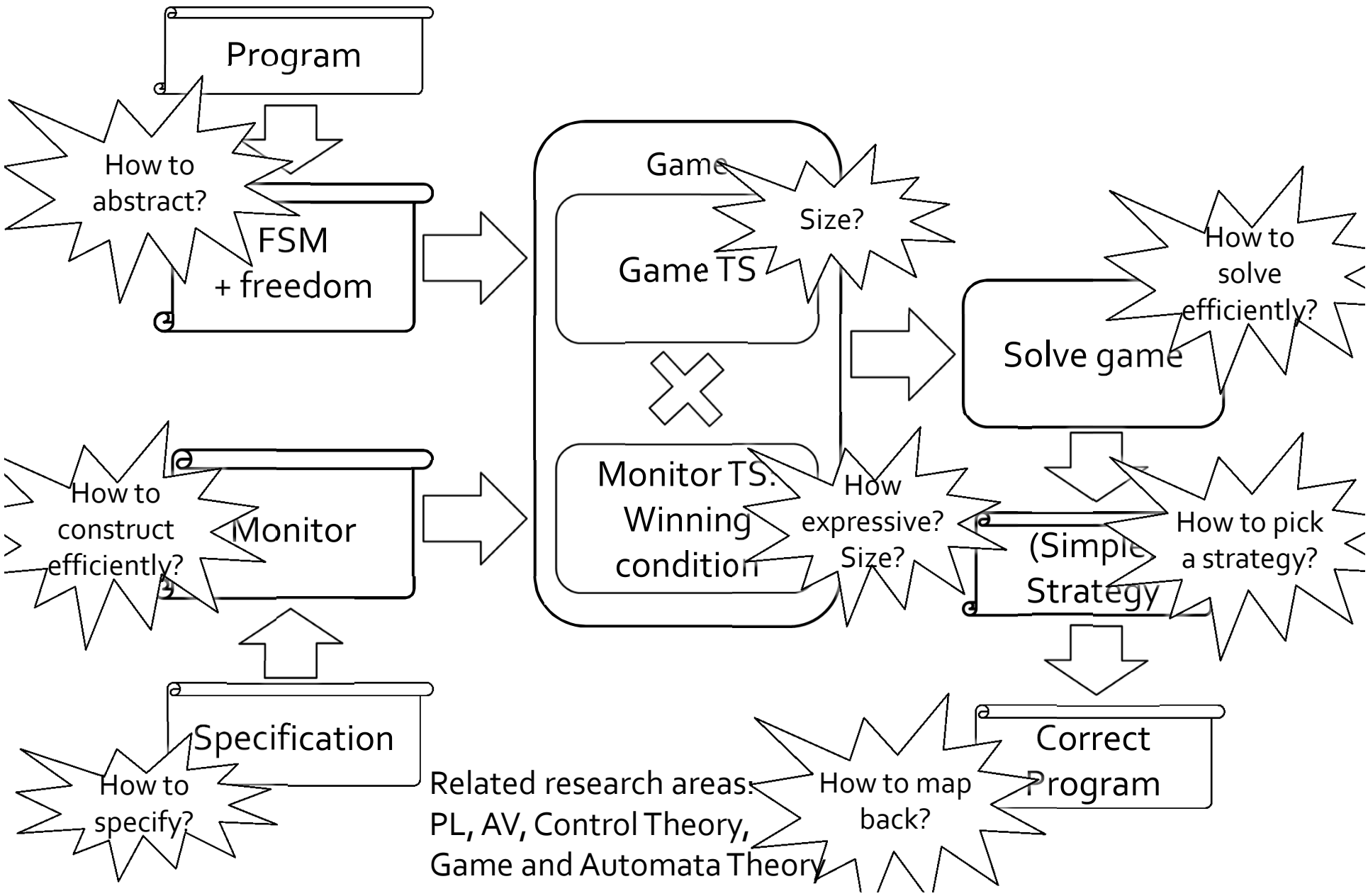
Church (1962), Büchi/Landweber (1969, games), Rabin (1972, trees), Pnueli/Rosner (1989, LTL)

Program Synthesis



Modern Controller Synthesis, see overview papers by Walukiewicz et al., Rutten & Girault, ...

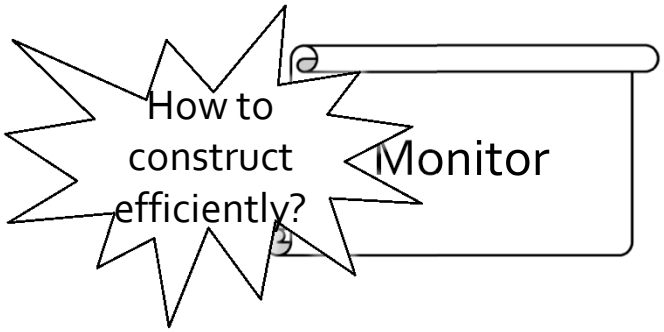
Issues?

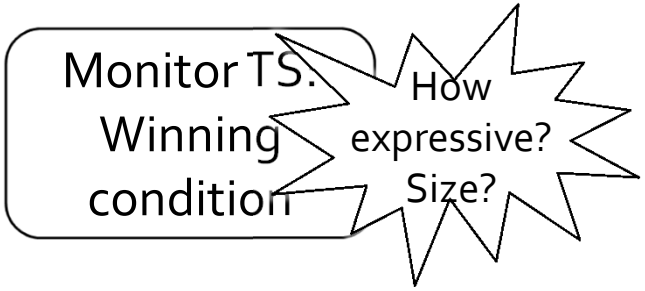


Related research areas:
PL, AV, Control Theory,
Game and Automata Theory

Issues with Monitor for LTL

- Determinization construction (Safra's)
- 2EXP worst case complexity
  - LTL is very succinct





Some Solutions

- Concentrate on subsets (different types of games)
  - Ramadge, Wonham (Proc IEEE'89) — Safety, Reachability Büchi, co-Büchi
  - Asarin, Maler, Pnueli, Sifakis (SSC'98) — Det. generators for several subsets
  - Alur, La Torre (LICS'01)
  - Alur, Madhusudan, Nam (BMC'03, STTT'05) — Safety+ using SAT, QBF, and BDDs
  - Wallmeier, Hütter, Thomas (CIAA'03) — Request-Response
  - Harding, Ryan, Schobbens (TACAS'05) — Work with nondet. automaton Identified syntactic subset
  - Jobstmann, Bloem (CAV'05)
  - Piterman, Pnueli, Sa'ar (VMCAI'06) (base of our work on synthesizing AMBA) — Generalized Reactivity-1 (GR-1)
- Optimize or avoid determinzation construction
  - Althoff, Thomas, Wallmeier (CIAA'05,TCS'06) — Implemented Safra
  - Piterman, Henzinger (CSL'06) — Improved Safra, Good-for-game
  - Kupferman, Vardi (FOCS'05)
  - Kupferman, Piterman, Vardi (CAV'06) — Bounded Synthesis (using co-Büchi)
  - Schewe, Finkbeiner (ATVA'07), Filiot, Jin, Raskin (CAV'09)
- Symbolic representation (e.g., using BDDs)

Next

- More partial programs…
  - this time with Sketching

What is sketching?

- A program synthesis system
  - generates small fragments of code
  - checks their validity against a specification

- A programming aid
  - help you write tricky programs
  - cleverness and insight come from you
    - sorry, no programming robots
  - computer helps with low-level reasoning

The sketching experience

sketch

implementation
(completed sketch)

Sketch language basics

- Sketches are programs with holes
  - write what you know
  - use holes for the rest

- 2 semantic issues
  - specifications
    - How does SKETCH know what program you actually want?
  - holes
    - Constrain the set of solutions the synthesizer may consider

Specifications

- Specifications constrain program behavior
  - assertions

```
assert x > y;
```

  - function equivalence

```
blockedMatMul(Mat a, Mat b) implements matMul
```

Is this enough?

Holes

- Holes are placeholders for the synthesizer
  - synthesizer replaces hole with concrete code fragment
  - fragment must come from a set defined by the user

Integer hole

- Define sets of integer constants

Example: Hello World of Sketching

spec:

```
int foo (int x)
{
    return x + x;
}
```

sketch:

```
int bar (int x) implements foo
{
    return x * ??;
}
```

Integer Hole

Integer Holes → Sets of Expressions

- Example: Find least significant zero bit
  - 0010 0101 → 0000 0010

```
int W = 32;

bit[W] isolateo (bit[W] x) {     // W: word size
        bit[W] ret = 0;
        for (int i = 0; i < W; i++)
                if (!x[i]) { ret[i] = 1; return ret;  }
}
```

- Trick:
  - Adding 1 to a string of ones turns the next zero to a 1
  - i.e. 000111 + 1 = 001000

!(x + ??) & (x + ??) →

!(x + 1) & (x + 0)	!(x + 1) & (x + 0xFFFF)
!(x + 0) & (x + 1)	!(x + 0xFFFF) & (x + 1)

Integer Holes → Sets of Expressions

- Example: Least Significant Zero Bit
  - 0010 0101 → 0000 0010

```
int W = 32;

bit[W] isolateo (bit[W] x) {     // W: word size
        bit[W] ret = 0;
        for (int i = 0; i < W; i++)
                if (!x[i]) { ret[i] = 1; return ret;  }
}

bit[W] isolateSk (bit[W] x) implements isolateo {

        return !(x + ??) & (x + ??) ;
}
```

Integer Holes → Sets of Expressions

- Least Significant One Bit
  - 0010 0100 → 0000 0100

```
int W = 32;

bit[W] isolateo (bit[W] x) {     // W: word size
        bit[W] ret = 0;
        for (int i = 0; i < W; i++)
                if (x[i]) { ret[i] = 1; return ret;  }
}
```

- Will the same trick work?
  - try it out

Integer Holes → Sets of Expressions

- Expressions with `??` == sets of expressions
  - linear expressions `x*?? + y*??`
  - polynomials `x*x*?? + x*?? + ??`
  - sets of variables `?? ? x : y`

- Semantically powerful but syntactically clunky
  - Regular Expressions are a more convenient way of defining sets

Regular Expression Generators

- {| RegExp |}

- RegExp supports choice '|' and optional '?'
  - can be used arbitrarily within an expression
    - to select operands `{| (x | y | z) + 1 |}`
    - to select operators `{| x (+ | -) y |}`
    - to select fields `{| n(.prev | .next)? |}`
    - to select arguments `{| foo( x | y, z) |}`

- Set must respect the type system
  - all expressions in the set must type-check
  - all must be of the same type

Least Significant One revisited

- How did I know the solution would take the form

  `!(x + ??) & (x + ??).`

- What if all you know is that the solution involves x, +, & and !.

```
bit[W] tmp=0;
{| x | tmp |} = {| (!)?((x | tmp) (& | +) (x | tmp | ??)) |};
{| x | tmp |} = {| (!)?((x | tmp) (& | +) (x | tmp | ??)) |};
return tmp;
```

This is now a set of statements
(and a really big one too)

Sets of statements

- Statements with holes = sets of statements

- Higher level constructs for Statements too
  - repeat

```
bit[W] tmp=0;
repeat(3){
   {| x | tmp |} = {| (!)?((x | tmp) (& | +) (x | tmp | ??)) |};
}
return tmp;
```

repeat

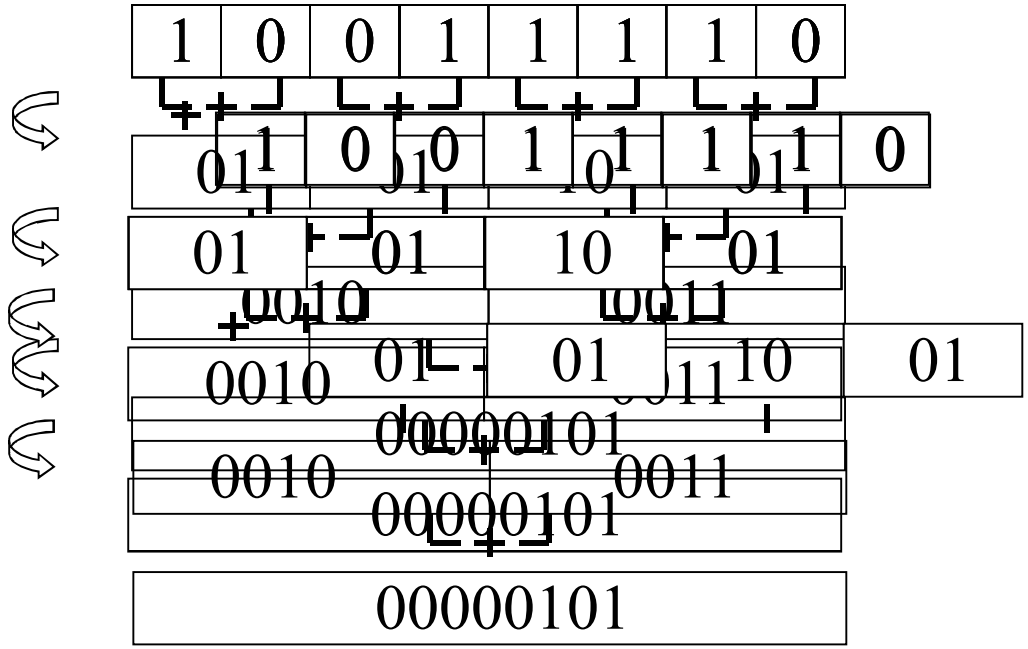
- Avoid copying and pasting
  - `repeat(n){ s}` ➔ $\underbrace{\texttt{s;s;…s;}}_{n}$
  - each of the n copies may resolve to a distinct stmt
  - n can be a hole too.

```
bit[W] tmp=0;
repeat(??){
   {| x | tmp |} = {| (!)?((x | tmp) (& | +) (x | tmp | ??)) |};
}
return tmp;
```

- Keep in mind:
  - the synthesizer won't try to minimize n
  - use --unrollamnt to set the maximum value of n

Example: logcount

```
int pop (bit[W] x)
{
  int count = 0;
  for (int i = 0; i < W; i++) {
    if (x[i]) count++;
  }
  return count;
}
```



Procedures and Sets of Procedures

- 2 types of procedures
  - standard procedures
    - represents a single procedure
    - all call sites resolve to the same procedure
    - identified by the keyword **`static`**
  - generators
    - represents a set of procedures
    - each call site resolves to a different procedure in the set
    - default in the current implementation

Example

```
int rec(int x, int y, int z){
  int t = ??;
  if(t == 0){return x;}

  if(t == 1){return y;}

  if(t == 2){return z;}

  if(t == 3){return rec(x,y,z) * rec(x,y,z);}

  if(t == 4){return rec(x,y,z) + rec(x,y,z);}

  if(t == 5){return rec(x,y,z) - rec(x,y,z);}
}

int sketch( int x, int y, int z ) implements spec{
  return rec(x,y, z);
}
```

```
int spec( int x, int y, int z ){
  return (x + x) * (y - z);
}
```

Step 1: Turn holes into special inputs

- The ?? Operator is modeled as a special input
  - we call them control inputs

```
bit[W] isolSk(bit[W] x)
{
   return ~(x + ??) & (x + ??);
}
```

```
bit[W] isolSk(bit[W] x, bit[W] c1, c2)
{
   return ~(x + c1) & (x + c2);
}
```

- Bounded candidate spaces are important
  - bounded unrolling of **`repeat`** is important
  - bounded inlining of generators is important

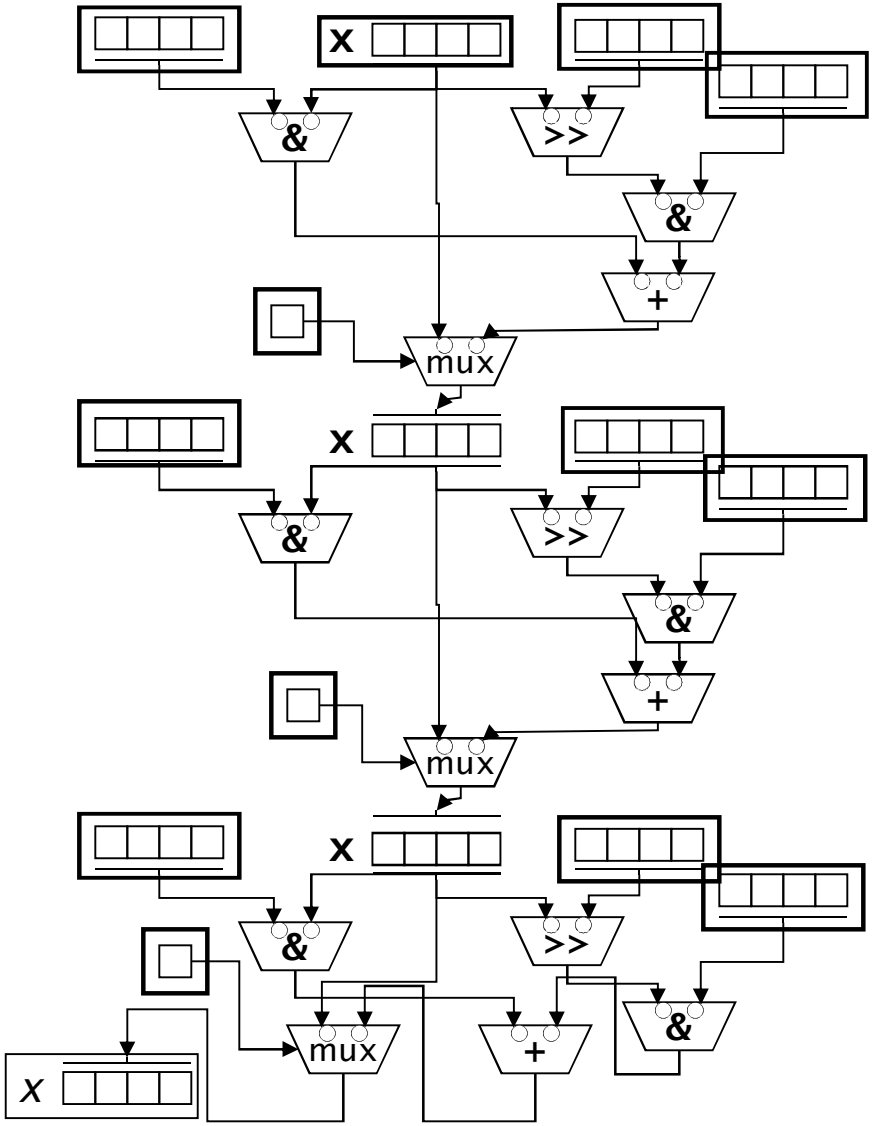
Step 2: Constraining the set of controls

- Correct control
  - causes the spec & sketch to match for all inputs
  - causes all assertions to be satisfied for all inputs

- Constraints are collected into a predicate

$$Q(in, c)$$

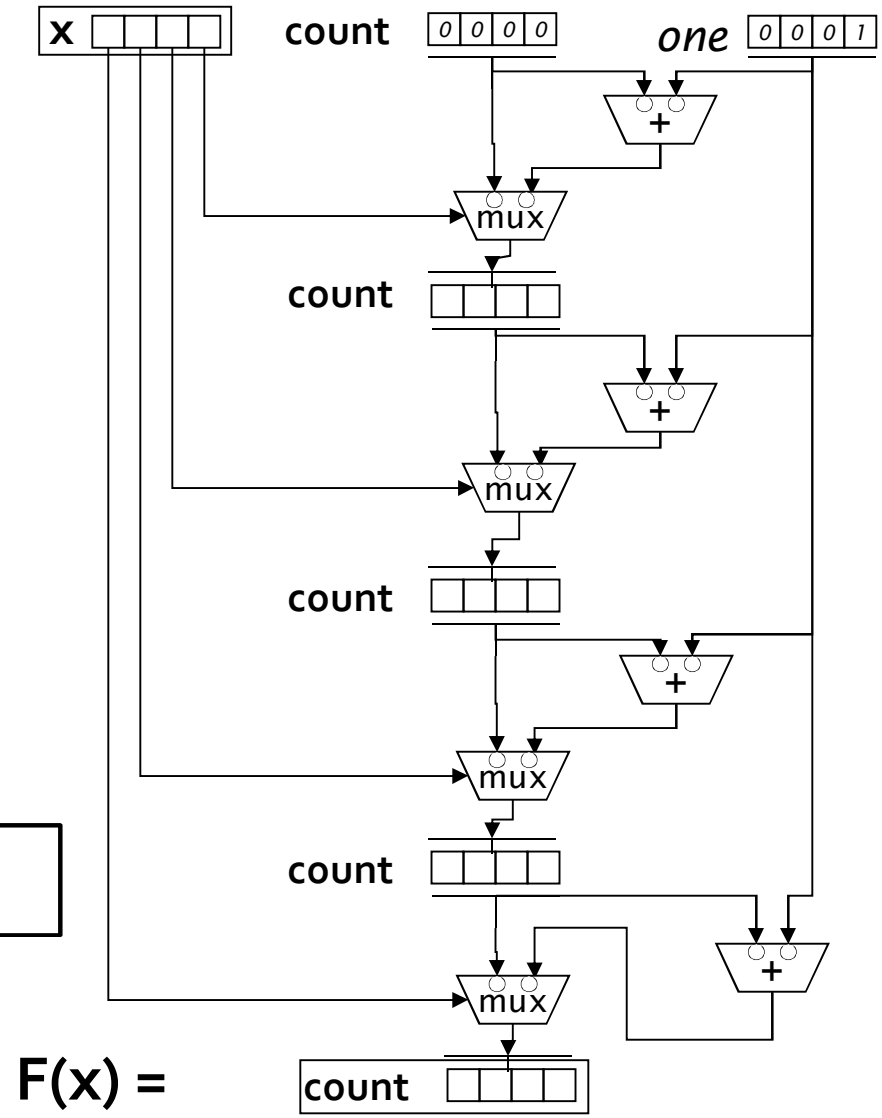
```
int popSketched (bit[W] x)
    implements pop {
    loop (??) {
        x = (x & ??)
        + ((x >> ??) & ??);
    }
    return x;
}
```



S(x,c) =

Ex : Population count.

```
int pop (bit[W] x)
{
   int count = 0;
   for (int i = 0; i < W; i++) {
      if (x[i]) count++;
   }
   return count;
}
```



$$Q(in, c) \triangleq S(x,c) == F(x)$$

A Sketch as a constraint system

Synthesis reduces to constraint satisfaction

$$\exists c \forall x. Q(x,c)$$

Constraints are too hard for standard techniques

- Universal quantification over inputs
- Too many inputs
- Too many constraints
- Too many holes

Insight

Sketches are not arbitrary constraint systems

- They express the high level structure of a program

A small set of inputs can fully constrain the soln

- focus on corner cases

$$\exists c \forall x \text{ in } E.Q(x,c) \quad \text{where } E = \{x_1, x_2, \ldots, x_k\}$$

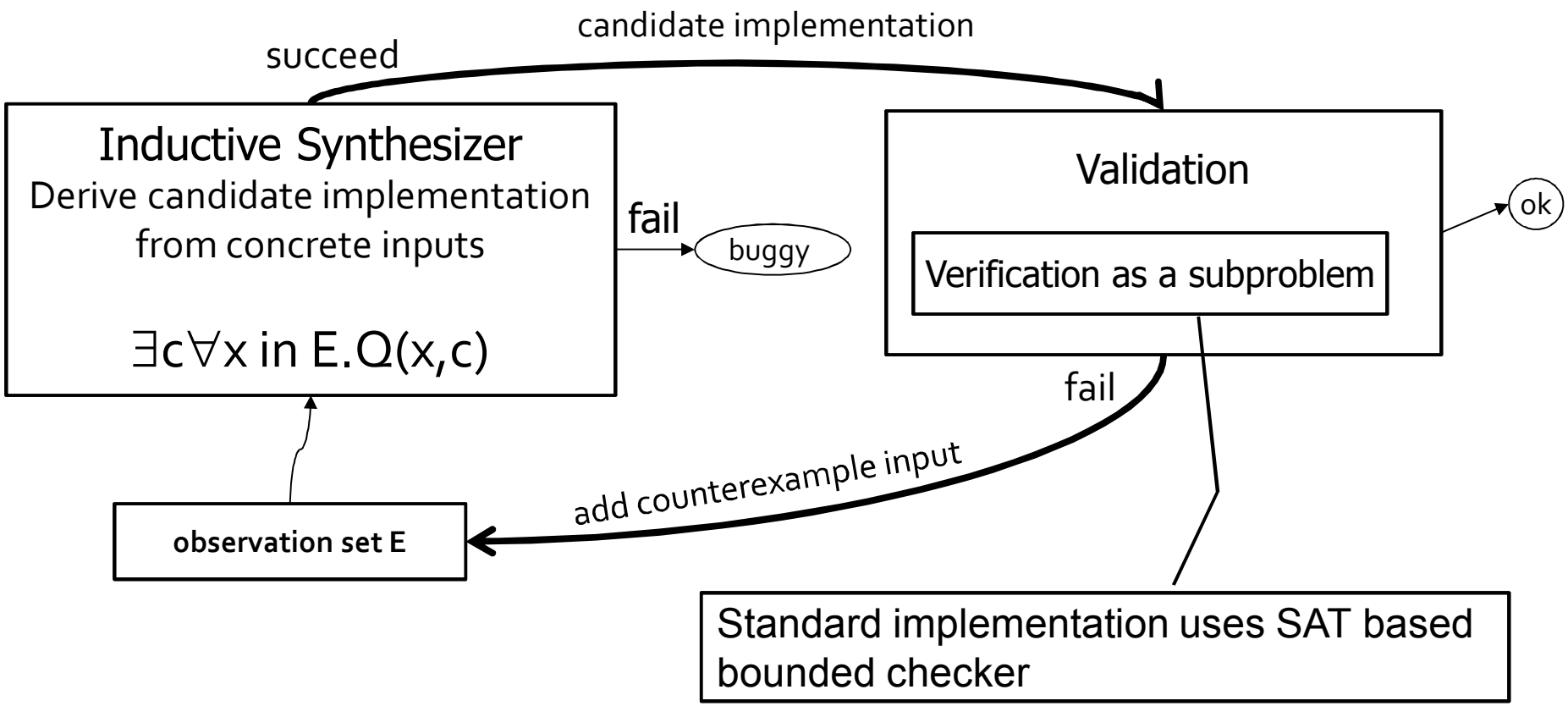
This is an inductive synthesis problem !

- but how do we find the set E?
- and how do we solve the inductive synthesis problem?

Step 3: Counterexample Guided Inductive Synthesis

Idea: Couple inductive synthesizer with a verifier

- Verifier is charged with detecting convergence



Inductive Synthesis

Deriving a candidate from a set of observations

$$\exists c \forall x \text{ in } E.Q(x,c) \quad \text{where } E = \{x_1, x_2, \ldots, x_k\}$$

Encode c as a bit-vector

- natural encoding given the integer holes

Encode $Q(x_i, c)$ as boolean constraints on the bit-vector

Solve constraints using SAT solver

- with lots of preprocessing in between

Summary