Infinite Games: Motivation

Barbara Jobstmann

CNRS/Verimag (Grenoble, France)

Bucharest, May 2010

Build Correct HW/SW Systems

- Use logic to specify correctness properties, e.g.:
  - *every job sent to the printer is eventually printed*
  - *two jobs do not overlap (only one job is printed at a time)*
  - *a job that is canceled will be interupted*

  These are conditions on infinite sequences (system runs), and can be specified by automata and logical formulas.

Build Correct HW/SW Systems

- Use logic to specify correctness properties, e.g.:
  - *every job sent to the printer is eventually printed*
  - *two jobs do not overlap (only one job is printed at a time)*
  - *a job that is canceled will be interupted*

  These are conditions on infinite sequences (system runs), and can be specified by automata and logical formulas.
- Given a logical specification, we can do either:
  - VERIFICATION: prove that a given system satisfies the specification
  - SYNTHESIS: build a system that satisfies the specification

Example: Elevator

- Aim: build controller that moves elevator of 10 floor building
- Environment: Passengers pressing buttons to (1) call elevator and (2) request floor
- System state:
  1. Set of requested floor numbers: $\{0, 1\}^{10}$
  2. Current position of lift: $\{1, \ldots, 10\}$
  3. Indicator whose turn is next (assuming lift and passengers act in alternation) $\{0, 1\}$

Infinite Games

Two players:

1. Controller is Player 0
2. Passengers are Player 1

A play of a game is an infinite sequence of states of elevator transition system, where the two players choose moves alternatively.

How does the transition system look like?

- State space: $\{0,1\}^{10} \times \{1,\ldots,10\} \times \{0,1\}$
- Transitions:
  - Player 0: $(r_1 \ldots r_{10}, j, 0) \rightarrow [r'_1 \ldots r'_{10}, j', 1]$ s.t. $r_j = 0$, $\forall_{i \neq j} r_i = r'_i$
    Actions: open/closes doors and move lift
  - Player 1: $[r_1 \ldots r_{10}, j, 1] \rightarrow (r'_1 \ldots r'_{10}, j', 0)$ s.t. $j = j'$, $\forall i : r_i \leq r'_i$
    Actions: request floors

Desired Properties

- Every requested floor is eventually reached
- Floors along the way are severed if requested
- If no floor is request, elevator goes to ground floor
- ...

These are conditions on infinite sequences!

Player 0 (controller) wins the play if all conditions are satisfied independent of the choices Player 1 makes. This corresponds to finding a winning strategy for Player 0 in an infinite game.

Our Aim

Solution of the Synthesis Problem

1. Decide whether there exists such a winning strategy - Realizability Problem
2. If “yes”, then construct the system - Synthesis Problem

Main result:

The synthesis problem is algorithmically solvable for finite-state systems with respect to specifications given as ω-automata or linear-time temporal logic.

Other Applications of Games

- Program repair or program sketching
- Nicer and more intuitive proofs for logics over trees
- Verification for logics over trees

Model Checking versus Repair

An Example

Lock Example

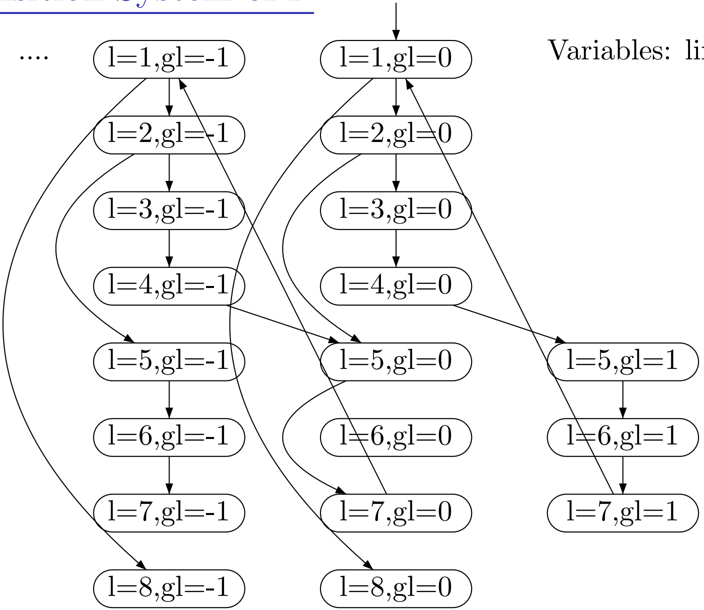
```
     ...
1    while(...) {
2      if (...) {
3        lock();
4        gotlock++;
       }
       ...
       ...
5      if (gotlock!=0)
6        unlock();
7      gotlock--;
     }
8    ...
```

Properties

1. P1: do not aquire a lock twice
2. P2: do not call unlock without holding the lock

Transition System of P

Variables: line, gotlock



Recall LTL

Boolean Operators: $\neg$, $\wedge$, $\vee$, $\rightarrow$,...

Temporal Operators:

- **next**: $\bigcirc\varphi$ ... in the next step φ holds
- **until**: $\varphi_1 \,\mathsf{U}\, \varphi_2$ ... at some point in the future φ_2 holds and until then φ_1 holds

Useful abbreviations:

- **eventually**: $\Diamond\varphi = \mathsf{true}\,\mathsf{U}\,\varphi$
- **always**: $\Box\varphi = \neg\Diamond\neg\varphi$
- **weakuntil**: $\varphi_1 \,\mathsf{W}\, \varphi_2 = (\varphi_1 \,\mathsf{U}\, \varphi_2) \vee \Box\varphi_1$

Note that

$$\neg(\varphi_1 \,\mathsf{U}\, \varphi_2) = (\neg\varphi_2 \,\mathsf{U}\, \neg\varphi_1 \wedge \neg\varphi_2) \vee \Box\neg\varphi_2 = \neg\varphi_2 \,\mathsf{W}(\neg\varphi_1 \wedge \neg\varphi_2).$$

Our properties in LTL

1. P1: do not aquire a lock twice

   Whenever we have called lock, we are not allowed to call it again before calling unlock.

Our properties in LTL

1. P1: do not aquire a lock twice

   Whenever we have called lock, we are not allowed to call it again before calling unlock. $\square((l = 3) \rightarrow \bigcirc(\neg(l = 3) \, \mathsf{W} (l = 6)))$

Our properties in LTL

1. P1: do not aquire a lock twice

   Whenever we have called lock, we are not allowed to call it again before calling unlock. $\Box((l = 3) \rightarrow \bigcirc(\neg(l = 3) \,\mathsf{W}\, (l = 6)))$

2. P2: do not call unlock without holding the lock

Our properties in LTL

1. P1: do not aquire a lock twice

   Whenever we have called lock, we are not allowed to call it again before calling unlock. $\Box((l = 3) \rightarrow \bigcirc(\neg(l = 3)\, \mathsf{W}(l = 6)))$

2. P2: do not call unlock without holding the lock

   $(\neg(l = 6)\, \mathsf{W}(l = 3)) \wedge (l = 6 \rightarrow \bigcirc(\neg(l = 6)\, \mathsf{W}(l = 3)))$

From LTL to Automata: Expansion rules

- $\Box\varphi = \varphi \wedge \bigcirc\Box\varphi$
- $\Diamond\varphi = \varphi \vee \bigcirc\Diamond\varphi$
- $\varphi_1 \,\mathsf{U}\, \varphi_2 = \varphi_2 \vee (\varphi_1 \wedge \bigcirc\varphi_1 \,\mathsf{U}\, \varphi_2)$
- $\varphi_1 \,\mathsf{W}\, \varphi_2 = \varphi_2 \vee (\varphi_1 \wedge \bigcirc\varphi_1 \,\mathsf{W}\, \varphi_2)$

Example: $\Box((l = 3) \rightarrow \bigcirc(\neg(l = 3) \,\mathsf{W}(l = 6)))$

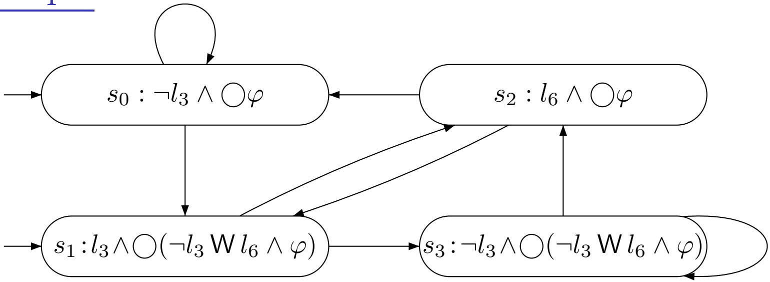
Shortcuts: l_3 for $(l = 3)$ and l_6 for $(l = 6)$

$$\varphi = \Box(\neg l_3 \vee (l_3 \wedge \bigcirc(\neg l_3 \,\mathsf{W}\, l_6)))$$

Expand: $(\neg l_3 \vee (l_3 \wedge \bigcirc(\neg l_3 \,\mathsf{W}\, l_6))) \wedge \bigcirc\varphi$

DNF: $s_0 \vee s_1$ with $s_0 = \neg l_3 \wedge \bigcirc\varphi$ and $s_1 = l_3 \wedge \bigcirc(\neg l_3 \,\mathsf{W}\, l_6 \wedge \varphi)$

Example



Expand: $\neg l_3 \,\mathsf{W}\, l_6 \wedge \varphi$

$(l_6 \vee (\neg l_3 \wedge \bigcirc(\neg l_3 \,\mathsf{W}\, l_6))) \wedge ((\neg l_3 \wedge \bigcirc \varphi) \vee (l_3 \wedge \bigcirc(\neg l_3 \,\mathsf{W}\, l_6 \wedge \varphi)))$

(1) $l_6 \wedge \neg l_3 \wedge \bigcirc \varphi \,:\, s_2$

(2) $l_6 \wedge l_3 \cdots = \mathsf{false}$

(3) $(\neg l_3 \wedge \bigcirc(\neg l_3 \,\mathsf{W}\, l_6)) \wedge (\neg l_3 \wedge \bigcirc \varphi) \,:\, s_3$

(4) $(\neg l_3 \wedge \cdots \wedge l_3 \cdots = \mathsf{false}$

Model Checking

$$L(\text{Program}) \subseteq L(P1)$$

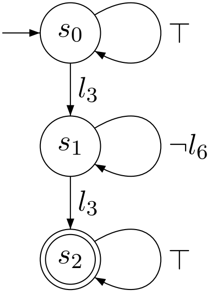
$$L(\text{Program}) \cap L(\neg P1) = \emptyset$$

Automaton for ¬ P1

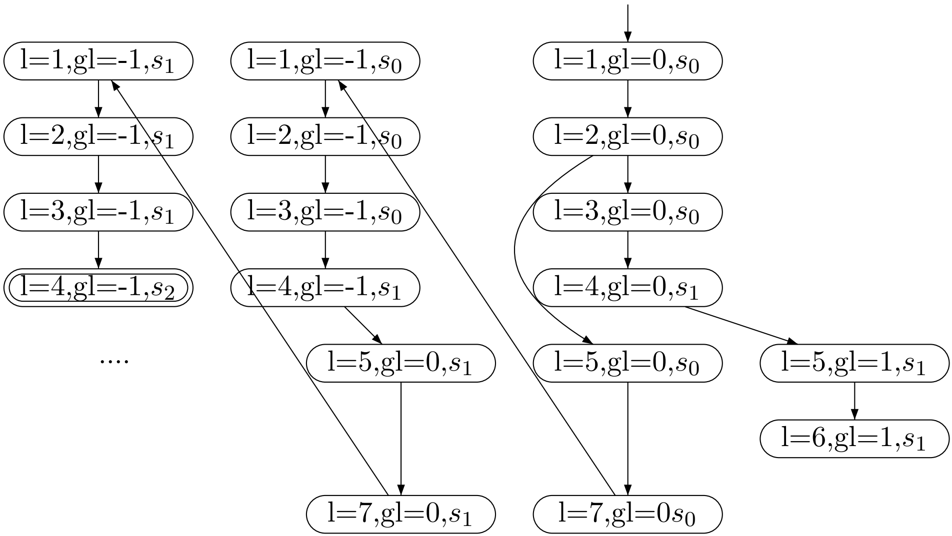
$$\neg P1 = \neg\Box(l_3 \rightarrow \bigcirc(\neg l_3 \,\mathsf{W}\, l_6))$$

$$\neg P1 = \Diamond(l_3 \wedge \bigcirc(\neg l_6 \,\mathsf{U}\, l_3))$$

Simplified version:



Product of Program and Property



Counterexample

1. Line 1: enter while loop
2. Line 2: skip over if
3. ...
4. Line 1: enter while loop
5. Line 2: enter if (call lock)
6. ...
7. Line 1: enter while loop
8. Line 2: enter if (call lock again)

```
     ...
1    while(...) {
2      if (...) {
3        lock();
4        gotlock++;
       }
       ...
       ...
5      if (gotlock!=0)
6        unlock();
7      gotlock--;
     }
8    ...
```

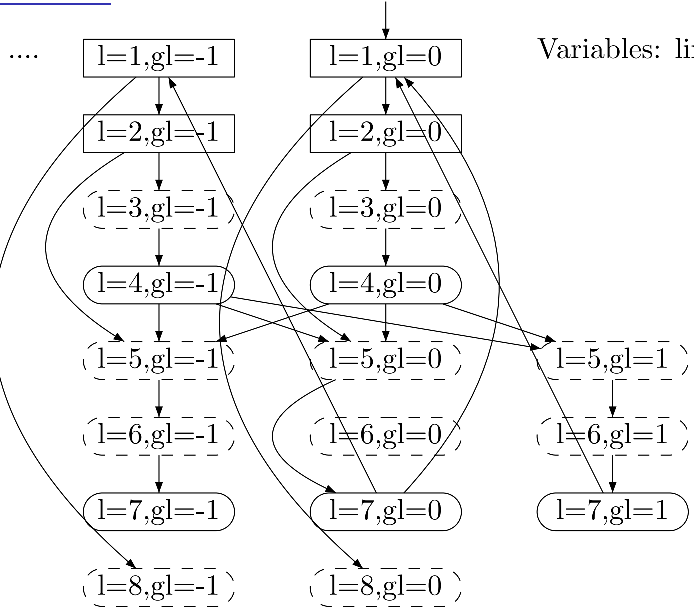
Repair

Repair: Step 1 - Free variables

```
1    while(...) {
2      if (...) {
3        lock();
4        gotlock=?;
       }
       ...
       ...
5      if (gotlock!=0)
6        unlock();
7      gotlock=?;
     }
8    ...
```

Game on P

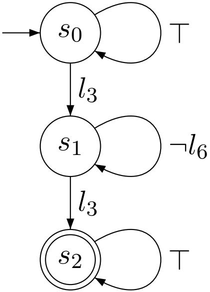
Variables: line, gotlock



Repair: Winning Condition

Note in MC: non-determinism due to input and due to automaton are treated the same way!

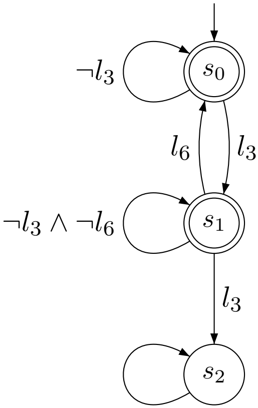
In Game: non-determinism may cause troubles.



Deterministic Automata/Observer

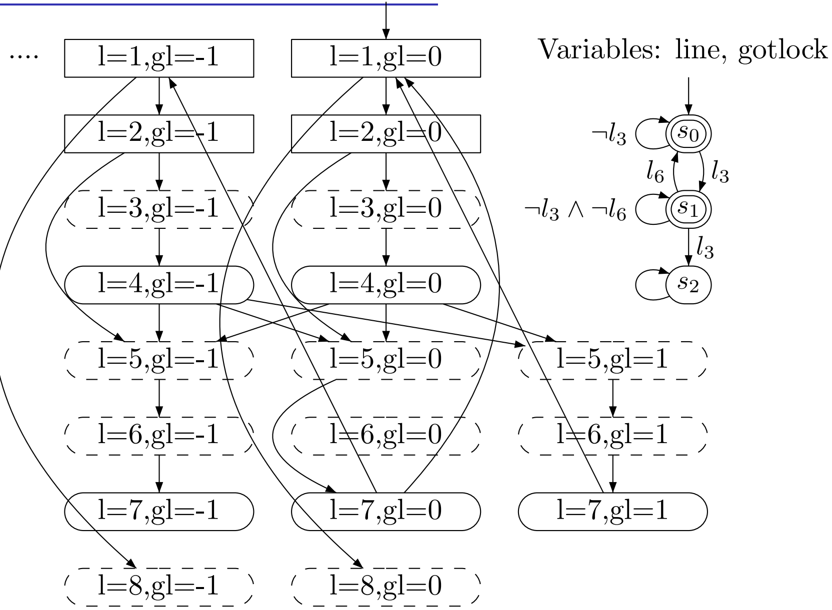
Recall,

$$\varphi = \Box(\neg l_3 \vee (l_3 \wedge \bigcirc(\neg l_3 \,\mathsf{W}\, l_6)))$$



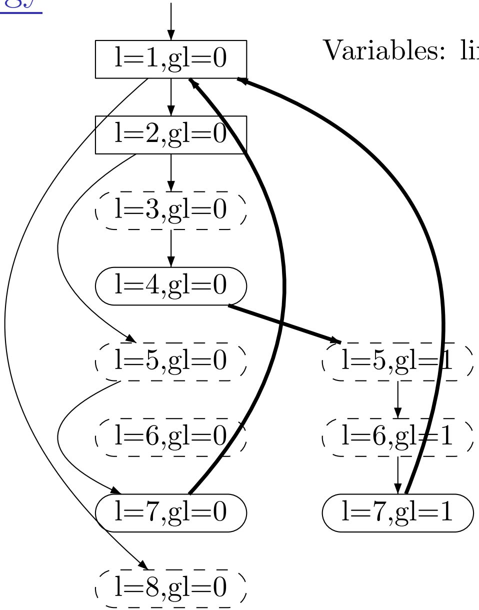
Note: this is a safety automaton.

Add Automaton to Game on P



A Winning Strategy

Variables: line, gotlock



A Correct Program

```
while(...) {
  if (...) {
    lock();
    gotlock=1;
  }
  ...
  ...
  if (gotlock!=0)
    unlock();
  gotlock=0;
}
...
```