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 interrupted

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

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 $\omega$-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        }
6        if (gotlock!=0)
7            unlock();
8        gotlock--;
9    }
10   ...
...
Properties

1. P1: do not acquire a lock twice
2. P2: do not call unlock without holding the lock
Transition System of P

Variables: line, gotlock

Diagram showing transitions between states based on line and gotlock values.
Recall LTL

Boolean Operators: \( \neg, \land, \lor, \rightarrow, \ldots \)

Temporal Operators:

1. **next**: \( \bigcirc \varphi \) ... in the next step \( \varphi \) holds

2. **until**: \( \varphi_1 \mathbin{\mathsf{U}} \varphi_2 \) ... at some point in the future \( \varphi_2 \) holds and until then \( \varphi_1 \) holds

Useful abbreviations:

1. **eventually**: \( \Diamond \varphi = \text{true} \mathbin{\mathsf{U}} \varphi \)

2. **always**: \( \Box \varphi = \neg \Diamond \neg \varphi \)

3. **weakuntil**: \( \varphi_1 \mathbin{\mathsf{W}} \varphi_2 = (\varphi_1 \mathbin{\mathsf{U}} \varphi_2) \lor \Box \neg \varphi_1 \)

Note that

\[
\neg(\varphi_1 \mathbin{\mathsf{U}} \varphi_2) = (\neg \varphi_2 \mathbin{\mathsf{U}} \neg \varphi_1 \land \neg \varphi_2) \lor \Box \neg \varphi_2 = \neg \varphi_2 \mathbin{\mathsf{W}} (\neg \varphi_1 \land \neg \varphi_2).
\]
Our properties in LTL

1. P1: do not acquire 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 acquire a lock twice

   Whenever we have called lock, we are not allowed to call it again before calling unlock. \( \Box((l = 3) \rightarrow \bigcirc(\neg(l = 3) W(l = 6))) \)
Our properties in LTL

1. P1: do not acquire a lock twice
   Whenever we have called lock, we are not allowed to call it again before calling unlock. $\Box ( (l = 3) \rightarrow \Diamond (\neg (l = 3) W (l = 6) ) )$

2. P2: do not call unlock without holding the lock
Our properties in LTL

1. P1: do not acquire a lock twice
   Whenever we have called lock, we are not allowed to call it again before calling unlock. $\square((l = 3) \rightarrow \bigcirc(\neg(l = 3) \ W(l = 6)))$

2. P2: do not call unlock without holding the lock
   $\neg(l = 6) \ W(l = 3)) \land (l = 6 \rightarrow \bigcirc(\neg(l = 6) \ W(l = 3)))$
From LTL to Automata: Expansion rules

- □ϕ = ϕ ∧ □□ϕ
- ◇ϕ = ϕ ∨ □◇ϕ
- ϕ₁ U ϕ₂ = ϕ₂ ∨ (ϕ₁ ∧ □ϕ₁ U ϕ₂)
- ϕ₁ W ϕ₂ = ϕ₂ ∨ (ϕ₁ ∧ □ϕ₁ W ϕ₂)

Example: □((l = 3) → □(¬(l = 3) W (l = 6)))

Shortcuts: l₃ for (l = 3) and l₆ for (l = 6)

ϕ = □(¬l₃ ∨ (l₃ ∧ □(¬l₃ W l₆)))

Expand: (¬l₃ ∨ (l₃ ∧ □(¬l₃ W l₆))) ∧ □ϕ

DNF: s₀ ∨ s₁ with s₀ = ¬l₃ ∧ □ϕ and s₁ = l₃ ∧ □(¬l₃ W l₆ ∧ ϕ)
Example

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

$\forall s \in \{0, 2\}:
\begin{align*}
\neg l_3 \land \bigcirc (\neg l_3 \mathsf{W} l_6 \land \varphi) & : s_1 \\
l_6 \land \neg l_3 \land \bigcirc \varphi & : s_2 \\
\neg l_3 \land \bigcirc (\neg l_3 \mathsf{W} l_6 \land \varphi) & : s_3 \\
\neg l_3 \land \cdots \land l_3 \cdots & = \text{false}
\end{align*}$
Model Checking

\[ L(\text{Program}) \subseteq L(P1) \]

\[ L(\text{Program}) \cap L(\neg P1) = \emptyset \]
Automaton for $\neg P1$

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

$\neg P1 = \Diamond (l_3 \land \bigcirc (\neg l_6 \mathcal{U} l_3))$

Simplified version:
Product of Program and Property

\[ l=1, gl=-1, s_1 \]
\[ l=2, gl=-1, s_1 \]
\[ l=3, gl=-1, s_1 \]
\[ l=4, gl=-1, s_2 \]

\[ l=1, gl=-1, s_0 \]
\[ l=2, gl=-1, s_0 \]
\[ l=3, gl=-1, s_0 \]

\[ l=1, gl=0, s_0 \]
\[ l=2, gl=0, s_0 \]
\[ l=3, gl=0, s_0 \]

\[ l=4, gl=0, s_1 \]
\[ l=4, gl=0, s_1 \]

\[ l=1, gl=0, s_1 \]
\[ l=5, gl=0, s_1 \]
\[ l=5, gl=0, s_0 \]

\[ l=6, gl=1, s_1 \]

\[ l=7, gl=0, s_1 \]
\[ l=7, gl=0, s_0 \]
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)

```c
... 
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   }
9   ...
10  ...
11  ...
12  ...
13  if (gotlock!=0)
14    unlock();
15    gotlock=?;
16  }
17  ...
18  ...
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.
Recall,

\[
\varphi = \Box (\neg l_3 \lor (l_3 \land \bigcirc (\neg l_3 \mathsf{W} l_6)))
\]

Note: this is a safety automaton.
Add Automaton to Game on P

Variables: line, gotlock

\[-l_3 \lor -l_6\]

\[-l_3, l_6\]

\[-l_3, l_6\]

\[-l_3, l_6\]
A Winning Strategy

Variables: line, gotlock

\[
\begin{align*}
&l=1, gl=0 \\
&l=2, gl=0 \\
&l=3, gl=0 \\
&l=4, gl=0 \\
&l=5, gl=0 \\
&l=6, gl=0 \\
&l=7, gl=0 \\
&l=8, gl=0 \\
&l=5, gl=1 \\
&l=6, gl=1 \\
&l=7, gl=1
\end{align*}
\]
A Correct Program

1 while(...) {
2     if (...) {
3         lock();
4         gotlock=1;
5         
6     }
7     ... 
8     ...
9     ...
10    if (gotlock!=0)
11         unlock();
12         gotlock=0;
13     }
14 }