Session 1: Games, Automata and Synthesis

Session 1: Games, Automata and Synthesis

Introduction

Traditional program verification

```c
int cut_square(int i)
    return i <= 0 ? 0 : i*i;
```

Introduction

Traditional program verification

```
int cut_square(int i)
    return i <= 0 ? 0 : i*i;
```

Requirements (What needs proofing)

- The output is always `>= 0`
- The output is always larger or equal to the absolute value of the input

Introduction

Traditional program verification

```
int cut_square(int i)
    return i <= 0 ? 0 : i*i;
```

Requirements (What needs proofing)

- The output is always `>= 0`
- The output is always larger or equal to the absolute value of the input

Approach (How to prove)

- Establish invariant
- Use some form of induction
- May rely on external solvers like SAT or SMT

Needs

A clear first input then output relation

Introduction

Model checking

```
int next_move(state* cstate)
  // ...
  return nm;
```

```
void next_state(state* cstate,
                state* nstate, int nm)
  // ...
```

Works on *infinite* behavior, reasons about continuous systems

Introduction

Model checking

```
int next_move(state* cstate)
    // ...
    return nm;
```

```
void next_state(state* cstate,
                state* nstate, int nm)
    // ...
```

Works on *infinite* behavior, reasons about continuous systems

Requirements (What needs proofing)

- Some logical specification The elevator will at some point open its door at every floor to which it is called.

Introduction

Model checking

```
int next_move(state* cstate)
    // ...
    return nm;
```

```
void next_state(state* cstate,
                state* nstate, int nm)
    // ...
```

Works on *infinite* behavior, reasons about continuous systems

Requirements (What needs proofing)

- Some logical specification The elevator will at some point open its door at every floor to which it is called.

Approach (How to prove)

- Translate (the negation of) the specification and the model to some finite automaton
- Check that there exists no run violating the specification

Introduction

Model checking

```
int next_move(state* cstate)
  // ...
  return nm;
```

```
void next_state(state* cstate,
                state* nstate, int nm)
  // ...
```

Works on *infinite* behavior, reasons about continuous systems

Requirements (What needs proofing)

- Some logical specification The elevator will at some point open its door at every floor to which it is called.

Approach (How to prove)

- Translate (the negation of) the specification and the model to some finite automaton
- Check that there exists no run violating the specification

Needs

A continuously executing system

Model checking: A closer look

Correctness

The model may not exhibit a single trace violating the specification

Model checking: A closer look

Correctness

The model may not exhibit a single trace violating the specification

The elevator model

- *Specification*: How the elevator should behave governed by the
- *Controller*: Choosing the next action as a function of
- *State*: the current position of the elevator and the
- *Environment*: behavior, that is which floors are demanded

Model checking: A closer look

Correctness

The model may not exhibit a single trace violating the specification

The elevator model

- *Specification*: How the elevator should behave governed by the
- *Controller*: Choosing the next action as a function of
- *State*: the current position of the elevator and the
- *Environment*: behavior, that is which floors are demanded

The run is a succession of actions of the environment (choices over the environment APs) and the controller (choices over the controlled APs).

Model checking: A closer look

Correctness

The model may not exhibit a single trace violating the specification

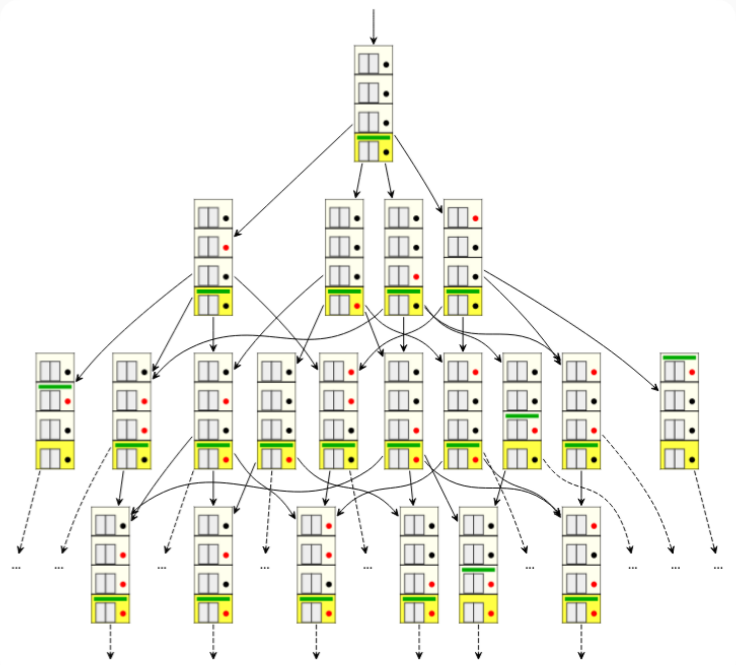
The elevator model

- *Specification*: How the elevator should behave governed by the
- *Controller*: Choosing the next action as a function of
- *State*: the current position of the elevator and the
- *Environment*: behavior, that is which floors are demanded

The run is a succession of actions of the environment (choices over the environment APs) and the controller (choices over the controlled APs).

The model is **correct** if the controller can guarantee the specification for **all** valid moves of the environment.

Model checking: A closer look



Introducing: Synthesis

Synthesis is so to speak the flip-side of verification:

Instead of verifying that a model/controller verifies a specification, why not directly generate it such a manner?

Introducing: Synthesis

Synthesis is so to speak the flip-side of verification:

Instead of verifying that a model/controller verifies a specification, why not directly generate it such a manner?

Intuitively, the idea is to make the *choices* explicit and create a two-player game between the environment (*env*, player 0) and the controller (*player*, player 1).

Introducing: Synthesis

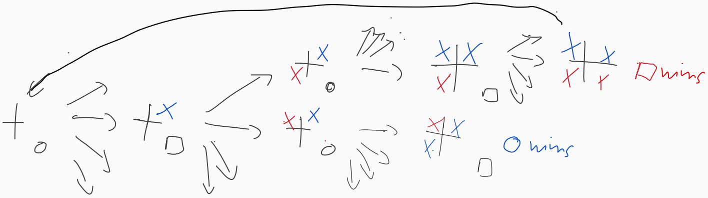
Synthesis is so to speak the flip-side of verification:

Instead of verifying that a model/controller verifies a specification, why not directly generate it such a manner?

Intuitively, the idea is to make the *choices* explicit and create a two-player game between the environment (*env*, player 0) and the controller (*player*, player 1).

If the controller can always *win* the game, he has an answer or strategy for all possible environment behaviors. This strategy then by construction verifies the specification.

Simplified 2 wins: Env wins if it can occupy a diagonal at any point



Definition 1

We are only concerned with 2 player games with perfect information.

The game is played between player 0 (the *Env*) and player 1 (the *Player*). To ease notations, we also define player i ($i \in [0, 1]$) and his opponent player $i - 1$.

Definition 1

We are only concerned with 2 player games with perfect information.

The game is played between player 0 (the *Env*) and player 1 (the *Player*). To ease notations, we also define player i ($i \in [0,1]$) and his opponent player $i-1$.

Arena

An arena $\mathcal{A} = (V, V_0, V_1, E)$ - a finite set of vertices V - the set of vertices V_i owned by player i partitioning V - $E \subseteq V \times V$ the set of directed edges - for every vertex v the set $\{v' | (v, v') \in E\}$ is non-empty

Definition 1

We are only concerned with 2 player games with perfect information.

The game is played between player 0 (the *Env*) and player 1 (the *Player*). To ease notations, we also define player i ($i \in [0, 1]$) and his opponent player $i - 1$.

Arena

An arena $\mathcal{A} = (V, V_0, V_1, E)$ - a finite set of vertices V - the set of vertices V_i owned by player i partitioning V - $E \subseteq V \times V$ the set of directed edges - for every vertex v the set $\{v' | (v, v') \in E\}$ is non-empty

We say arena is **alternating** if for every edge $(v, v') \in E$ we have $v \in V_i$ and $v' \in V_{i-1}$

Definition 2

Sub-Arena

Let $\mathcal{A}$ be an arena and $V' \subseteq V$ a subset of vertices.
The sub-arena of $\mathcal{A}$ induced by V' called $\mathcal{A}_{V'}$ is defined as
$\mathcal{A}_{V'} = (V', V_0 \cap V', V_1 \cap V', E \cap (V' \times V'))$

Definition 2

Sub-Arena

Let $\mathcal{A}$ be an arena and $V' \subseteq V$ a subset of vertices.
The sub-arena of $\mathcal{A}$ induced by V' called $\mathcal{A}_{V'}$ is defined as
$\mathcal{A}_{V'} = (V', V_0 \cap V', V_1 \cap V', E \cap (V' \times V'))$

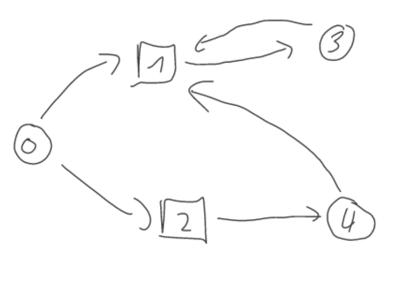
Note that not all sets V' induce a sub-arena: for instance the rule that a successor needs to exist may be violated.

Definition 2

Sub-Arena

Let $\mathcal{A}$ be an arena and $V' \subseteq V$ a subset of vertices.
The sub-arena of $\mathcal{A}$ induced by V' called $\mathcal{A}_{V'}$ is defined as
$\mathcal{A}_{V'} = (V', V_0 \cap V', V_1 \cap V', E \cap (V' \times V'))$

Note that not all sets V' induce a sub-arena: for instance the rule that a successor needs to exist may be violated.



- $V' = \{1, 3\}$ induces a sub-arena
- $V' = \{2, 4\}$ does not

Definition 3

Intuitively, during the play the players push a token along the edges of the arena, whoever own the vertex decides on the next edge to use. During this process, we record the vertices (and sometimes) edges seen.

A Play

A play in $\mathcal{A}$ is an infinite sequence $\rho = \rho_0\rho_1\rho_2\rho_3\cdots \in V^\omega$ such that $\rho_n\rho_{n+1} \in E$ holds for all $n \in \mathbb{N}$.

Definition 3

Intuitively, during the play the players push a token along the edges of the arena, whoever own the vertex decides on the next edge to use. During this process, we record the vertices (and sometimes) edges seen.

A Play

A play in $\mathcal{A}$ is an infinite sequence $\rho = \rho_0\rho_1\rho_2\rho_3\cdots \in V^\omega$ such that $\rho_n\rho_{n+1} \in E$ holds for all $n \in \mathbb{N}$.

A Strategy

A strategy for player i ($i \in \{0, 1\}$) in an arena $\mathcal{A}$ is a function $\delta_i : V^*V_i \to V$ s.t. $\delta_i(wv) = v'$ implies $(v, v') \in E$ for every w and v.

Definition 3

Intuitively, during the play the players push a token along the edges of the arena, whoever own the vertex decides on the next edge to use. During this process, we record the vertices (and sometimes) edges seen.

A Play

A play in $\mathcal{A}$ is an infinite sequence $\rho = \rho_0\rho_1\rho_2\rho_3\cdots \in V^\omega$ such that $\rho_n\rho_{n+1} \in E$ holds for all $n \in \mathbb{N}$.

A Strategy

A strategy for player i ($i \in \{0, 1\}$) in an arena $\mathcal{A}$ is a function $\delta_i : V^*V_i \to V$ s.t. $\delta_i(wv) = v'$ implies $(v, v') \in E$ for every w and v.

In other words, a strategy decides what to do next given the history of the game and its choice is possible in $\mathcal{A}$.

Definition 4

A consistent play

A play in $\mathcal{A}$ is **consistent** with a strategy δ_i for player i if $\rho_{n+i} = \delta_i(\rho_0, \ldots, \rho_n)$ for every $n \in \mathbb{N}$.

Definition 4

A consistent play

A play in $\mathcal{A}$ is **consistent** with a strategy δ_i for player i if $\rho_{n+i} = \delta_i(\rho_0, \dots, \rho_n)$ for every $n \in \mathbb{N}$.

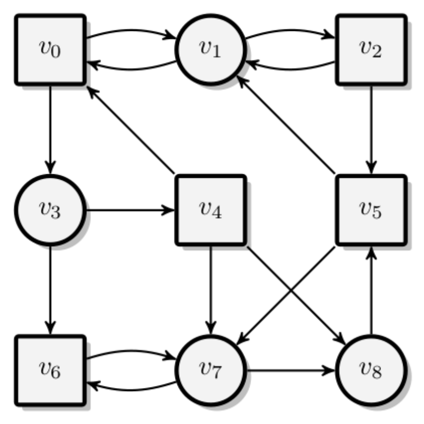
And

Positional strategy

A strategy δ_i is called positional (or memoryless) if $\delta_i(wv) = \delta_i(v)$ for all $w \in V^*$ and $v \in V$.

Since positional strategies are function from V to V (instead of $(V^* \times V)$ to V) we also denote them as such.

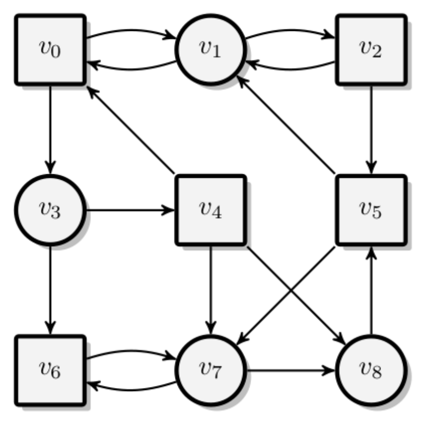
Definition 3&4 illustrations



player 0: circles
player 1: squares

- $v_0 v_3 (v_6 v_7)^\omega$ is a play
- *go right* is a positional strategy for player 0
- $v_0 (v_1 v_2 v_1 v_2 v_5 v_1)^\omega$ is a play consistent with the positional strategy *go right* for player 0

Definition 3&4 illustrations



player 0: circles
player 1: squares

- $v_0 v_3 (v_6 v_7)^\omega$ is a play
- *go right* is a positional strategy for player 0
- $v_0 (v_1 v_2 v_1 v_2 v_5 v_1)^\omega$ is a play consistent with the positional strategy *go right* for player 0
- Give the positional strategy δ_1 consistent with $v_0 (v_1 v_2 v_1 v_2 v_5 v_1)^\omega$
- Given we start in v_0, is there a strategy for player 1 to reach v_6?

Definition 5 (and final) [at least for now]

We can finally properly define a game

Game

A game $\mathcal{G}$ is defined by the tuple $(\mathcal{A}, Win)$ with $\mathcal{A}$ being the arena and $Win \subseteq V^\omega$ being the winning sequences.
We say the game is won by player 1 if and only if $\rho \in Win$. Otherwise it is won by player 0.

Definition 5 (and final) [at least for now]

We can finally properly define a game

Game

A game $\mathcal{G}$ is defined by the tuple $(\mathcal{A}, Win)$ with $\mathcal{A}$ being the arena and $Win \subseteq V^{\omega}$ being the winning sequences.
We say the game is won by player 1 if and only if $\rho \in Win$. Otherwise it is won by player 0.

Winning Strategy

A strategy δ_i for player i is called a **winning strategy** for vertex v if all plays starting v and consistent with δ_i are won by player i.

Definition 5 (and final) [at least for now]

We can finally properly define a game

Game

A game $\mathcal{G}$ is defined by the tuple $(\mathcal{A}, Win)$ with $\mathcal{A}$ being the arena and $Win \subseteq V^\omega$ being the winning sequences.
We say the game is won by player 1 if and only if $\rho \in Win$. Otherwise it is won by player 0.

Winning Strategy

A strategy δ_i for player i is called a **winning strategy** for vertex v if all plays starting v and consistent with δ_i are won by player i.

A Winning Region

The winning region $W_i(\mathcal{G})$ of player i is the set of vertices from which player i has a winning strategy.

On winning regions

Lemma: Winning regions do not intersect

or $W_0(\mathcal{G}) \cap W_1(G) = \emptyset$

On winning regions

Lemma: Winning regions do not intersect

or $W_0(\mathcal{G}) \cap W_1(G) = \emptyset$

Proof. Let $\mathcal{G} = (\mathcal{A}, Win)$. Towards a contradiction, assume there exists a vertex $v \in W_0(\mathcal{G}) \cap W_1(\mathcal{G})$.

Then, both players have a winning strategy from v, call them δ_0 and δ_1. Let $\rho = \rho(v, \delta_0, \delta_1)$, i.e., we let the players both use their winning strategy against each other, the play is consistent with both strategies.

Then, $\rho \in Win$, as δ_1 is a winning strategy for player 1 and $\rho \notin Win$, as δ_0 is a winning strategy for player 0.

Hence, we have derived the desired contradiction.

On winning regions and determinacy

Last slide we have shown that winning regions are always disjoint.

On winning regions and determinacy

Last slide we have shown that winning regions are always disjoint.

The other question is whether they also partition the vertices, that is whether $W_0(\mathcal{G}) \cup W_1(\mathcal{G}) = V$.

On winning regions and determinacy

Last slide we have shown that winning regions are always disjoint.

The other question is whether they also partition the vertices, that is whether $W_0(\mathcal{G}) \cup W_1(\mathcal{G}) = V$.

[Positional] Determinancy

Let $\mathcal{G}$be a game with vertex set V.
We say that $\mathcal{G}$is determined if $W_0($\mathcal{G}$) ∪W_1($\mathcal{G}$) = V. Furthermore, we say that $\mathcal{G}$is positionally determined if, from every vertex $v \in V$ one of the players has a positional winning strategy.

On winning regions and determinacy

Last slide we have shown that winning regions are always disjoint.

The other question is whether they also partition the vertices, that is whether $W_0(\mathcal{G}) \cup W_1(\mathcal{G}) = V$.

[Positional] Determinancy

Let $\mathcal{G}$be a game with vertex set V.
We say that $\mathcal{G}$is determined if $W_0($\mathcal{G}$) ∪W_1($\mathcal{G}$) = V. Furthermore, we say that $\mathcal{G}$is positionally determined if, from every vertex $v \in V$ one of the players has a positional winning strategy.

Note the subtility "one of the playes has **a** positional winning stragtegy [for each vertex v]".

This indicates that the strategy may depend on the initial vertex.

On winning regions and determinacy

Last slide we have shown that winning regions are always disjoint.

The other question is whether they also partition the vertices, that is whether $W_0(\mathcal{G}) \cup W_1(\mathcal{G}) = V$.

[Positional] Determinancy

Let $\mathcal{G}$be a game with vertex set V.
We say that $\mathcal{G}$is determined if $W_0($\mathcal{G}$) ∪W_1($\mathcal{G}$) = V. Furthermore, we say that $\mathcal{G}$is positionally determined if, from every vertex $v \in V$ one of the players has a positional winning strategy.

Note the subtility "one of the playes has **a** positional winning stragtegy [for each vertex v]".

This indicates that the strategy may depend on the initial vertex.

a