Course ITI8531: Software Synthesis and Verification

Lecture 15: Recap of Software Synthesis part of the course

Spring 2019

Leonidas Tsiopoulos
leonidas.tsiopoulos@taltech.ee

Reusing some material from previous slide sets for this lecture

Programming and Verification

- Verification with Model Checking:
  - *Given* a program (model) *P* and a specification φ, *check* that *P* satisfies φ.
  - Success:
    - Usage is increasing with several available model checkers based on advanced algorithmic methods.
  - Issues:
    - Designing *P* is *hard* and *expensive*.
    - Redesigning *P* when *P* does not satisfy φ is *hard* and *expensive*.
- Verification with Theorem Proving:
  - Similar issues as above.
- Alternative solution:
  - *Start* from specification φ and *synthesize P* such that *P* satisfies φ.

Synthesis – What it is about?

- The main motivation is that „*if we can verify why not go directly from specification to correct-by-construction systems by synthesis*?“
- Various approaches exist:
  - Deductive approach where first the realizability of a function is proved and then the program is extracted from the proof.
  - Computational approach where a transformational program is synthesized to produce a result on termination in terms of input and output relations.
    - By some called *classical* approach or just *program synthesis*.
  - Reactive/Temporal approach where programs are synthesized for *ongoing* computations (protocols, operating systems, robot controllers, etc).
- The focus of this course is on the reactive/temporal approach.

Reactive Synthesis

- *„if we can verify why not go **directly** from specification to correct-by-construction systems by synthesis*?“

- Now we are in 2019 and still „***directly“*** involves several transformational steps in the background no matter the underlying approach.

- Old topic started already in 1960s by Church.
  - Given a circuit interface specification partitioned to inputs and outputs and a behavioral specification in first order logic, determine if there is an automaton that realizes the specification. If the specification is realizable, construct an implementing automaton.

History of reactive synthesis

- Vardi, Wolper and Sistla in 1983 [5] showed that the translation from LTL to automata is of *elementary* (exponential) complexity.

- Safra in 1988 [6] showed that the construction of tree automata for strategy trees wrt LTL specification is doubly exponential (using [5]).
  - Procedure for the *determinization* of Rabin automata.

- In 1989 Pnueli and Rosner [7] established LTL *realizability* to be 2EXPTIME complete using Safra's approach.
  - Very high complexity when determinizing non-deterministic automata!
  - Caused halting of research in this field for many years.

History of reactive synthesis

- From beginning of 2000s this research topic was brought back to the scene with several approaches offering „Safraless“ solutions to avoid the very expensive determinisation step and also better algorithms working on „symbolic“ representation of the state space.

- The focus of this part of the course has been on one such approach implemented with the tool Acacia+.

System specification: satisfiability vs realizability

- Satisfiability: Exists some behavior that satisfies the specification.
- Realizability: Exists system that **implements** the specification and it **must work for all inputs** (controlled by the environment) to the system.

Example on satisfiability

- **Example**: Printer specification
- J_i - job i submitted, P_i - job i printing, $i \in \{1,2\}$.
- *Safety* property: two jobs are not printing together - $always\ \neg(P_1 \wedge P_2)$
- *Liveness* property: every job is eventually printed
  - $always\ \bigwedge_{i=1}^{2}(J_i \rightarrow eventually\ P_i)$
- Is specification satisfiable? Yes!
- Model *M*: A single state where J_i , P_i are all *false.*
- Can we extract a program from *M*? No!
- Why? *M* handles only one input sequence.
- J_i are inputs controlled by the environment. We need a system that handles all input sequences.
- Only satisfiability is not enough for synthesis!

Formal context for synthesis

- A *specification* will be in LTL over *input* and *output propositions*.
- A system will be an *automaton with output*.
- Input and output are combined to create a *sequence* of assignments to propositions.
- **All possible** *infinite paths* over the automaton should *satisfy* the specification.

LTL preliminaries

- Formal language which extends the propositional Boolean logic.
- Variables: atomic propositions, e.g., *p* and *q.*
- A set of atomic propositions partitioned to inputs and outputs denoting the basic *facts* about a system and its environment.
- Usual Boolean operators are allowed, e.g., $p \rightarrow q$ ($\neg p \vee q$) is an LTL formula, but it refers to the *first element* of an infinite sequence.

LTL preliminaries – Operators and formulae

Temporal operators

- **G**: **g**lobally (always), e.g., **G**($p \rightarrow q$) means "in each element of the sequence, $p \rightarrow q$ holds".
- **F**: in the **f**uture, e.g., **F**($p \rightarrow q$) means "for some element of the sequence, $p \rightarrow q$ holds".
- **X**: on the ne**x**t step, e.g., **X**($p \rightarrow q$) means "$p \rightarrow q$ holds for the second element of the sequence".
- **U**: **u**ntil, e.g., p **U** q means "q must happen at some step, and the sequence must satisfy p until (non-inclusive) q happens".
- Linear Temporal Logic **formulae** are constructed as follows:
  - $\varphi ::= p \,\|\, \varphi \wedge \varphi \,\|\, \neg\varphi \,\|\, \varphi_1 \,\mathbf{U}\, \varphi_2 \,\|\, \ldots$

Automata

- Systems with *discrete states*.
- Formally, $A = \langle \Sigma, Q, \delta, q_0 \rangle$, where
  - Σ – a finite input alphabet.
  - Q– a finite set of states.
  - δ: $Q \times \Sigma \rightarrow 2^Q$ – a transition function associating with state and an input letter a set of successor states.
  - q_0– an initial state.
- An input word $w = \sigma_0, \sigma_1, \ldots$ is a sequence of letters from Σ.
- A run $r = q_0, q_1, \ldots$ over w is a sequence of states starting from q_0 such that for every $i \geq 0$ we have $q_{i+1} \in \delta(q_i, \sigma_i)$.
- An automaton is deterministic if for every $q \in Q$ and $\sigma \in \Sigma$ we have $|\delta(q, \sigma)| \leq 1$.
- Several variations exist: Rabin, Büchi, ”Safety”, Uppaal Timed Automata, ...

Games for Synthesis – Why?

- We need to synthesize a system that **implements** the specification and it **must work for all inputs** (controlled by the environment) to the system.
- Controlling so that *uncontrollable* events do not lead to damage.
- This can be a *two-player game*.
- **Realizability** with regard to games: Existence of *winning strategy* for the system in a game *against* the environment.
  - Addressed (rather) efficiently by Pnueli and Rosner [7] providing better algorithms (based on μ-calculus and least fixpoint) compared to previous approaches.
  - But still these approaches were based on Safra's highly complex determinizing step.

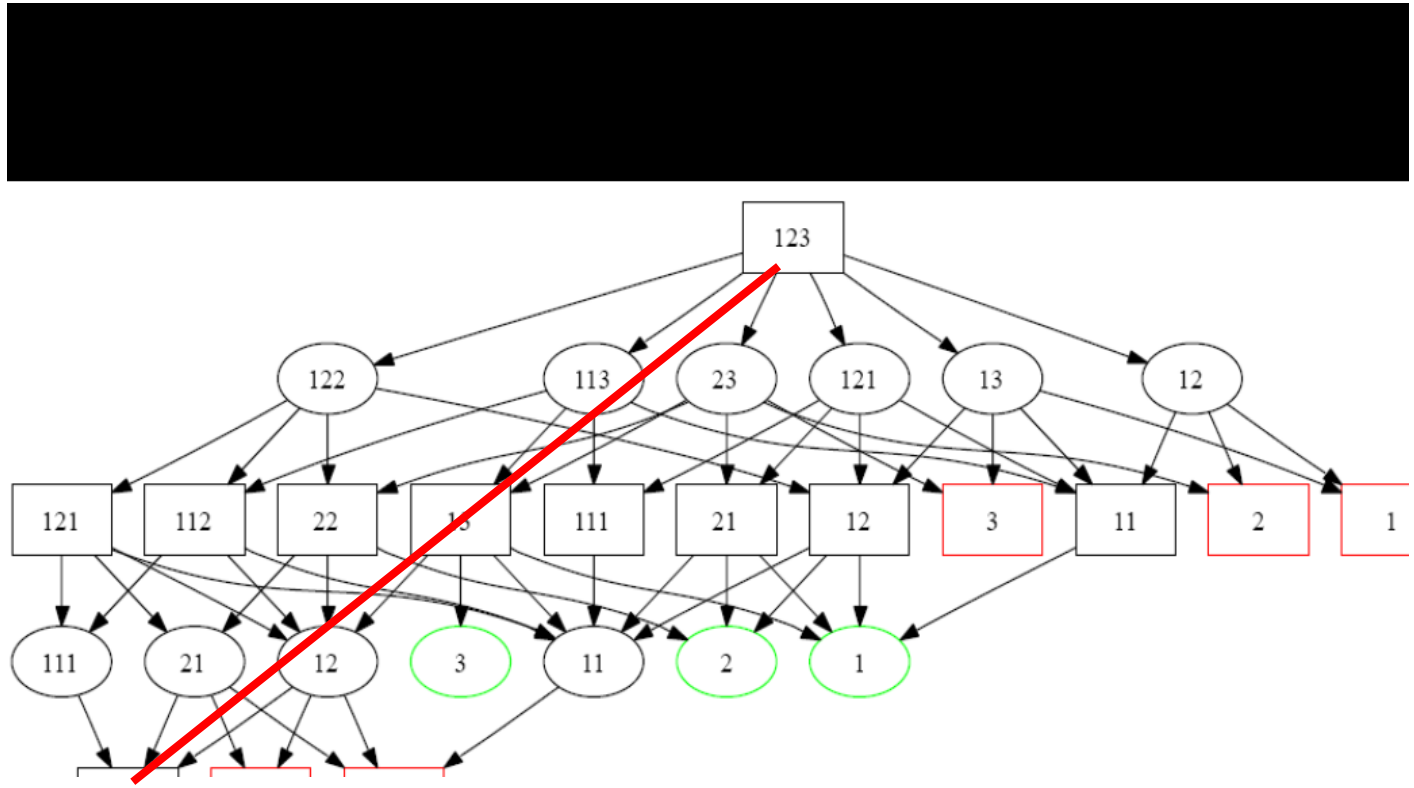
Games

- Formally, a game is $G = \langle V, V_0, V_1, E, \alpha \rangle$, where
  - V is a set of nodes.
  - V_0 and V_1 form a partition of V. V_0 concern the System and V_1 the Environment.
  - $E \subseteq V \times V$ is a set of edges.
- A *play* is $\pi = v_0, v_1, \ldots$
  - α is a *set of winning plays*.
- A *strategy* for player i is a function $f_i : V^* \cdot V_i \longrightarrow V$ such that
  $(v, f_i(w \cdot v)) \in E$.
- A play $\pi = v_0, v_1, \ldots$ is *compatible* with f_i if for every $j \geq 0$ such that $v_j \in V_i$ we have $v_{j+1} = f_i(v_0 \cdots v_j)$.
- A strategy for player 0 is *winning* if every play compatible with it is in α. A strategy for player 1 is winning if every play compatible with it is not in α.
- A node v is won by player i if she has a winning strategy for all plays starting from v.

A play of a game

Environment

System



Games - Realizability and Synthesis

- *Realizability*: Exists winning strategy for System.
- *Synthesis*: Obtain such winning strategy. How?
  - With a Transducer - Moore Machine (also Mealy depending on approach).
- Formally, $T = \langle \Delta, \Sigma, Q, q_0, \alpha, \beta \rangle$, where
  - Δ – input alphabet
  - Σ – output alphabet
  - Q – states
  - q_0 – initial state
  - $\alpha : Q \times \Delta \rightarrow Q$ – transition function
  - $\beta : Q \rightarrow \Sigma$ – output function.
- A transducer representing a winning strategy can be extracted from the winning states of the system after solving the game.

Game types for synthesis

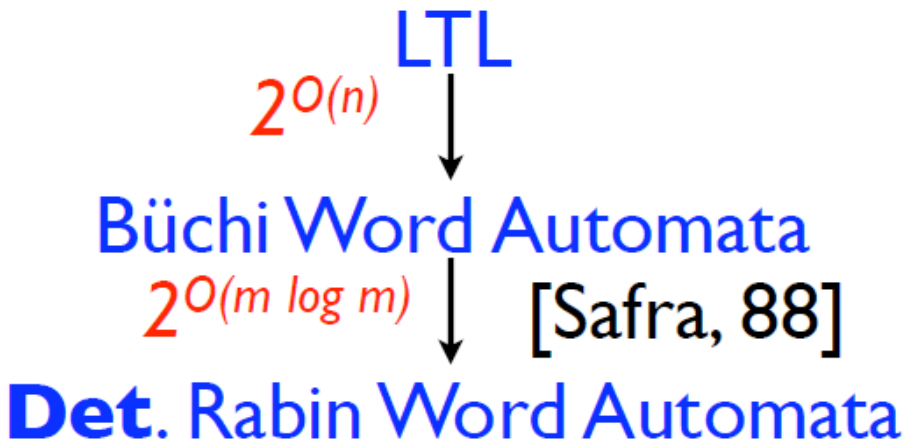
- Safety games
  - Avoiding the „bad“ state of the safety automaton.
- Reachability games – *dual* to safety games.
  - Trying to reach a target state.
- Büchi games
  - Accepting states from which system can force returning to an *accepting* state *infinitely often*.
  - Almost the same as for reachability games.

Avoiding the Classical Approach to LTL Synthesis

- LTL synthesis is a challenging problem due to 2EXPTIME theoretical complexity and lack of scalable algorithms for determinization of automata and solving games.
- There are some LTL-based synthesis approaches offering „Safraless“ solutions to avoid the very complex determinisation step and also better algorithms working on „symbolic“ representation of the state space during the game.
  - Even translating LTL formulae to symbolic automaton in the first place.

- Acacia+ and the techniques around it is one such „Safraless“ approach.

Classical solution by Pnueli and Rosner



Realizability

~

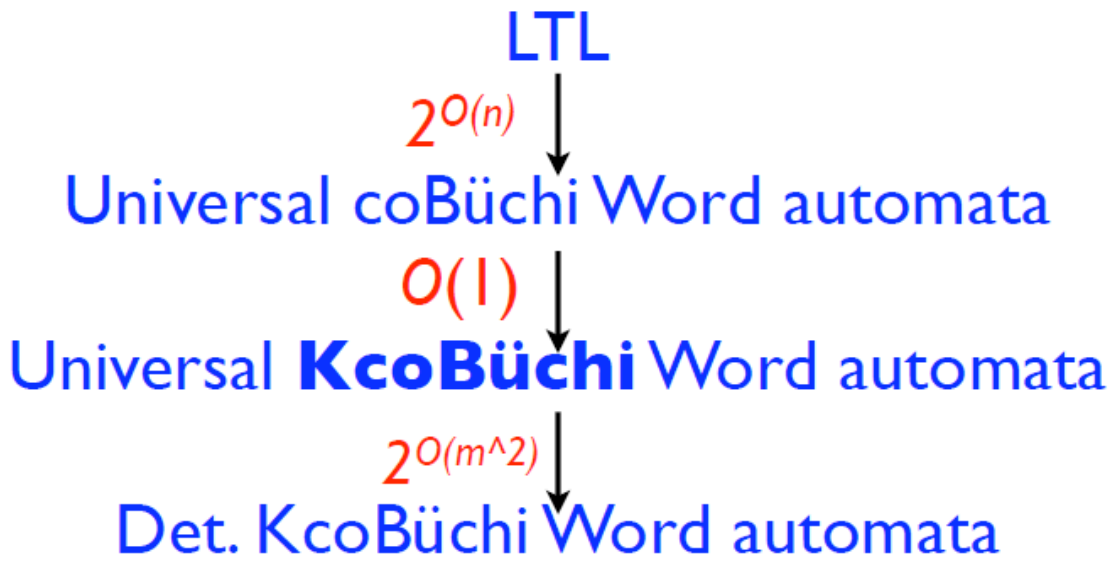
Rabin Game

The problem has been shown to be 2ExpTime-Complete by the same authors.

Acacia+: A tool for LTL synthesis

- Main contributions:
  - Efficient *symbolic* incremental algorithms based on *antichains* for game solving.
  - Synthesis of *small* winning strategies, when they exist.
  - *Compositional* approach for *large conjunctions* of LTL formulas.
  - Performance is better or similar to other existing tools but its *main advantage* is the generation of *compact strategies.*
- Application scenarios:
  - **Synthesis of control code from high-level LTL specifications**.
  - *Debugging* of unrealizable specifications by inspecting compact counter strategies.
  - *Generation of small deterministic automata* from LTL formulas, when they exist.

Acacia+ Safraless approach



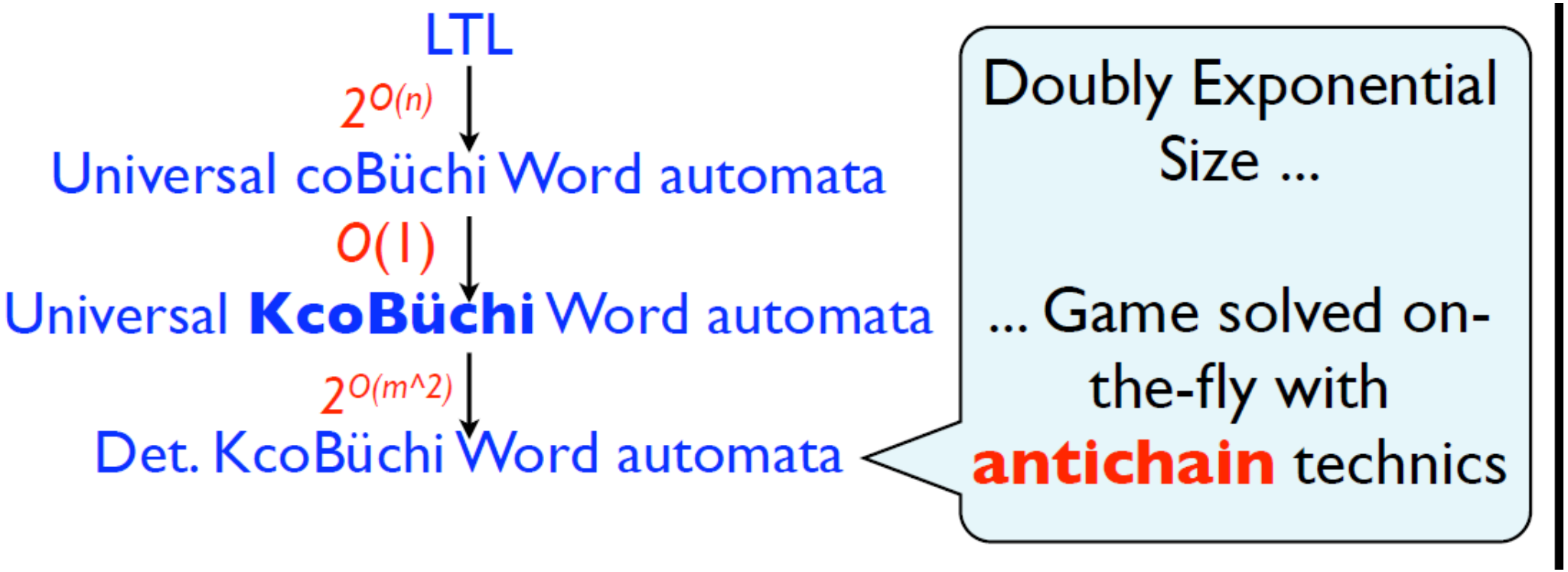
Realizability

~

Safety game

- Safety games are the simplest games to solve!

Acacia+ Safraless approach



- Safety games are the simplest games to solve!

Acacia+ and LTL Transformation to Automata (1)

- An *infinite word automaton* is a tuple $A = (\Sigma, Q, q_0, \alpha, \delta)$ where:
  - Σ is the *finite alphabet*,
  - Q is a *finite set of states*,
  - $q_0 \in Q$ is the *initial* state,
  - $\alpha \subseteq Q$ is a set of *final states* and
  - $\delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*.
    - For all $q \in Q$ and all $\sigma \in \Sigma$, $\delta(q, \sigma) = \{q' \mid (q, \sigma, q') \in \delta\}$.

- A is *deterministic* if $\forall q \in Q \cdot \forall \sigma \in \Sigma \cdot |\delta(q, \sigma)| \leq 1$.
- A is *complete* if $\forall q \in Q \cdot \forall \sigma \in \Sigma \cdot \delta(q, \sigma) = \emptyset$.

Acacia+ and LTL Transformation to Automata (2)

- A *run* of A on a *word* $w = \sigma_0\sigma_1 \cdot \cdot \cdot \in \Sigma^\omega$ is an infinite sequence of states $\rho = \rho_0\rho_1 \cdot \cdot \cdot \in Q^\omega$ such that $\rho_0 = q_0$ and $\forall i \geq 0 \cdot q_{i+1} \in \delta(q_i, \rho_i)$.
- The *set of runs* of A on w is denoted by $\text{Runs}_A(w)$.
- The number of times state q occurs along run ρ is denoted by $\text{Visit}(\rho, q)$.
- Three *acceptance conditions* (a.c.) are considered for infinite word automata. A word w is *accepted by* A if:
  - Non-deterministic Büchi : $\exists \rho \in \text{Runs}_A(w) \cdot \exists q \in \alpha \cdot \text{Visit}(\rho, q) = \infty$
    - Runs visits final states infinitely often.
  - Universal Co-Büchi : $\forall \rho \in \text{Runs}_A(w) \cdot \forall q \in \alpha \cdot \text{Visit}(\rho, q) < \infty$
    - Runs visit final states finitely often.
  - Universal K-Co-Büchi : $\forall \rho \in \text{Runs}_A(w) \cdot \forall q \in \alpha \cdot \text{Visit}(\rho, q) \leq K$
    - Runs visit at most K final states.

Acacia+ and LTL Transformation to Automata (3)

- The *set of words* accepted by A with the non-deterministic Büchi a.c. is denoted by $L_{\mathrm{b}}(A)$.
  - This implies that A is a non-deterministic Büchi word automaton (NBW).

- Similarly, the set of words accepted by A with the universal co-Büchi and universal *K*-co-Büchi a.c., are denoted respectively by $L_{\mathrm{uc}}(A)$ and $L_{\mathrm{uc},K}(A)$.
  - With those interpretations, A is a universal co-Büchi automaton (UCW) and that (A,*K*) is a universal *K*-co-Büchi automaton (U*K*CW) respectively.

- By duality, $L_{\mathrm{b}}(A) = \overline{L_{\mathrm{uc}}(A)}$ for any infinite word automaton A.

- Also, for any $0 \leq K_1 \leq K_2$, $L_{\mathrm{uc},K1}(A) \subseteq L_{\mathrm{uc},K2}(A) \subseteq L_{\mathrm{uc}}(A)$.

Turn-based Automata for Realizability of Games (1)

- To reflect the game point of view of the realizability problem the notion of *turn-based automata* is used to define the specification.
- *A turn-based automaton* A over the *input alphabet* Σ_I and the *output alphabet* Σ_O is a tuple $A = (\Sigma_I, \Sigma_O, Q_I, Q_O, q_0, \alpha, \delta_I, \delta_O)$ where:
  - Q_I, Q_O are *finite sets of input and output states* respectively,
  - $q_0 \in Q_O$ the *initial state*,
  - $\alpha \subseteq Q_I \cup Q_O$ is the set of *final states*,
  - $\delta_I \subseteq Q_I \times \Sigma_I \times Q_O$ and $\delta_O \subseteq Q_O \times \Sigma_O \times Q_I$ are the *input* and *output transition relations.*
- A is *complete* if for all $q_I \in Q_I$, and all $\sigma_I \in \Sigma_I$, $\delta_I(q_I, \sigma_I) \neq \emptyset$, **and** for all $q_O \in \Sigma_O$ and all $\sigma_O \in \Sigma_O$, $\delta_O(q_O, \sigma_O) \neq \emptyset$.

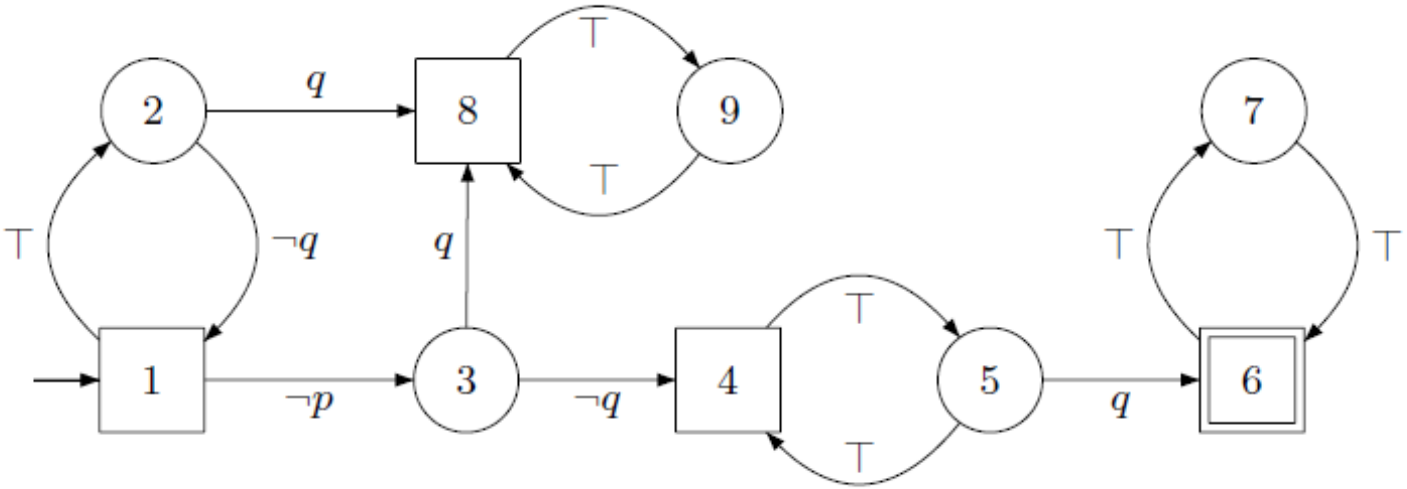
Turn-based Automata for Realizability of Games (2)

- Turn-based automata A run on words from Σ^ω.
- A *run* on a word $w = (o_0 \cup i_0)(o_1 \cup i_1) \cdots \in \Sigma^\omega$ is an infinite sequence of states $\rho = \rho_0\rho_1 \cdots \in (Q_O Q_I)^\omega$ such that $\rho_0 = q_0$ and for all $j \geq 0$,

  $(\rho_{2j}, o_j, \rho_{2j+1}) \in \delta_O$ and $(\rho_{2j+1}, i_j, \rho_{2j+2}) \in \delta_I$.
- All acceptance conditions we show carry over to turn-based automata.
- Every UCW (resp. NBW) with state set Q and transition set Δ is equivalent to a turn-based UCW (tbUCW) (resp. tbNBW) with $|Q| + |\Delta|$ states:
  - the new set of states is $Q \cup \Delta$,
  - final states remain the same,
  - and each transition $r = q \xrightarrow{\sigma_i \cup \sigma_o} q' \in \Delta$ where $\sigma_o \in \Sigma_O$ and $\sigma_i \in \Sigma_I$ is split into a transition $q \xrightarrow{\sigma_o} r$ and a transition $r \xrightarrow{\sigma_i} q'$.

Example of tbUCW

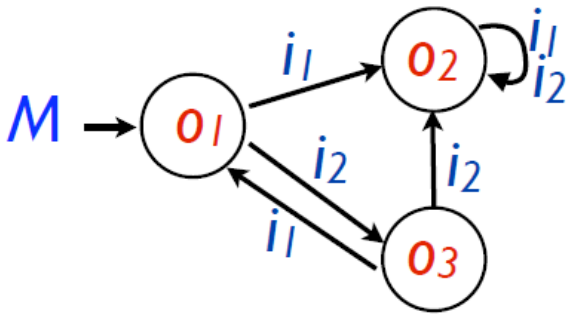
- tbUCW for **F**$q \rightarrow (p\mathbf{U}q)$ where $I = \{q\}$ and $O = \{p\}$
- Output states $Q_O = \{1, 4, 6, 8\}$ are depicted by squares and input states $Q_I = \{2, 3, 5, 7, 9\}$ by circles
- $\top$ stands for the sets Σ_I or Σ_O, depending on the context, $\neg q$ (resp. $\neg p$) stands for the sets that do not contain q (resp. p), i.e. the empty set.
- At state 1, if controller does not assert p and next the environment does not assert q, then the run is in state 4. From this state, whatever the controller does, if the environment asserts q, then the controller loses, as state 6 will be visited infinitely often.



- A strategy for the controller is to assert p all the time, therefore the runs will loop in states 1 and 2 until the environment asserts q. Afterwards the runs will loop in states 8 and 9, which are non-final.

Finite state strategies

- We know that if an LTL formula is realizable, there exists a finite-state strategy that realizes it [PR89].
- Finite-state strategies are represented as complete Moore machines in Acacia+.



$L(M)$ = *traces of infinite paths*

E.g. $(o_1 \cup i_1)(o_2 \cup i_2)^{\omega}$

- The LTL realizability problem reduces to decide, given a tbUCW A over inputs Σ_I and outputs Σ_O, whether there is a non-empty Moore machine *M* such that $L(M) \subseteq L_{uc}(A)$.
- The tbUCW is equivalent to an LTL formula given as input and is constructed by using tools *Wring* or *LTL2BA*.

Bounding the number of *visited* final states

Lemma 1. Given a Moore machine *M* with *m* states, and a tbUCW A with *n* states, if $L(M) \subseteq L_{\text{uc}}(A)$, then all runs on words of $L(M)$ visit at most *m*×*n* final states.

Proof. The infinite paths of *M* starting from the initial state define words that are accepted by A. Therefore in the product of *M* and A, there is no cycle visiting an accepting state of A, which allows one to bound the number of visited final states by the number of states in the product.

Corollary. $L(M) \subseteq L_{\text{uc}}(A)$ iff $L(M) \subseteq L_{\text{uc, } m\text{x}n}(A)$

Reduction to a bounded universal *K*-co-Büchi automaton

Lemma 2. Given a realizable tbUCW A over *inputs* Σ_I and *outputs* Σ_O with *n* states, there exists a non-empty Moore machine with at most $n^{2n+2} + 1$ states that realizes it.

***Proof**. In the paper. Re-using an older result by Safra.*

Theorem. Let A be a tbUCW over Σ_I, Σ_O with *n* states and $K = 2n(n^{2n+2} + 1)$ (from above proof). Then A is realizable iff (A,K) is realizable.

Determinization of U*K*CWs

- What is left is to reduce the tbU*K*CW realizability problem to a safety game.
- It is based on the determinization of tbU*K*CWs into complete turn-based deterministic 0-Co-Büchi automata, which can also be viewed as safety games.
- The resulting deterministic automaton is always equipped with a *partial-order on states* that can be used to efficiently manipulate its state space using the antichain method.

Determinization of U*K*CWs

- **Lemma**: U*K*CWs are determinizable.
- **Sketch of Proof**: Let $A = (\Sigma, Q, q_0, \alpha, \Delta, K)$ be a U*K*CW.
- *For each state q*, ***count*** *the maximal number of final states visited by runs ending up in q.*
  - Extending the usual subset construction with counters.
- Set of states $\mathbb{F}$: ***counting functions*** *F* from *Q* to [-1,0,...,*K*+1].
  - The counter of a state *q* is set to −1 when no run up to *q* visited final states.
- ***Initial*** *counting function* F_0: $q \longrightarrow (q_0 \in \alpha)$ **if** $q = q_0$, -1 otherwise.
- ***Final*** *states* are *functions F* such that $\exists q$: $F(q) > K$.
  - The final states are the sets in which a state has its counter **greater than** *K.*

Determinization of tbU*K*CWs

- Let A be a tbU*K*CW (Σ_O, Σ_I, Q_O, Q_I, q_0, α, Δ_O, Δ_I) with $K \in \mathbb{N}$.
  - Let $Q = Q_O \cup Q_I$ and $\Delta = \Delta_O \cup \Delta_I$.
- Let det(A,K) = (Σ_O,Σ_I,$\mathbb{F}_O$, $\mathbb{F}_I$, F_0, α', δ_O, δ_I) where:
  - Set of states $\mathbb{F}_O$: ***counting functions*** F_O from Q_O to [-1,0,...,K+1].
  - Set of states $\mathbb{F}_I$: ***counting functions*** F_I from Q_I to [-1,0,...,K+1].
  - ***Initial*** *counting function* F_0: $q \in Q_O \rightarrow (q_0 \in \alpha)$ **if** $q = q_0$, -1 otherwise.
  - $\alpha' = \{F \in \mathbb{F}_O \cup \mathbb{F}_I \mid \exists q, F(q) > K\}$.
  - succ(F, σ) = $q \rightarrow \texttt{max}\{\texttt{min}(K + 1, F(p) + (q \in \alpha)) \mid q \in \Delta(p, \sigma), F(p) \neq -1\}$
    - There is a successor state if the run up to p visited finaal states.
  - $\delta_O = \text{succ}|_{\mathbb{F}O \times \Sigma O}$, $\delta_I = \text{succ}|_{\mathbb{F}I \times \Sigma I}$

Reduction to Safety Games

- The *game* $G(A,K)$ can be defined as follows:
  - it is det(A,K) where *input states* are viewed as Player *I*'s states (env.) and output states as Player *O*'s states (system).
- $G(A,K) = (\mathbb{F}_O, \mathbb{F}_I, F_0, T,$ safe) where safe $= \mathbb{F} \setminus \alpha'$ and $T = \{(F, F') \mid$

  $\exists \sigma \in \Sigma_O \cup \Sigma_I, F' = \text{succ}(F, \sigma)\}$.

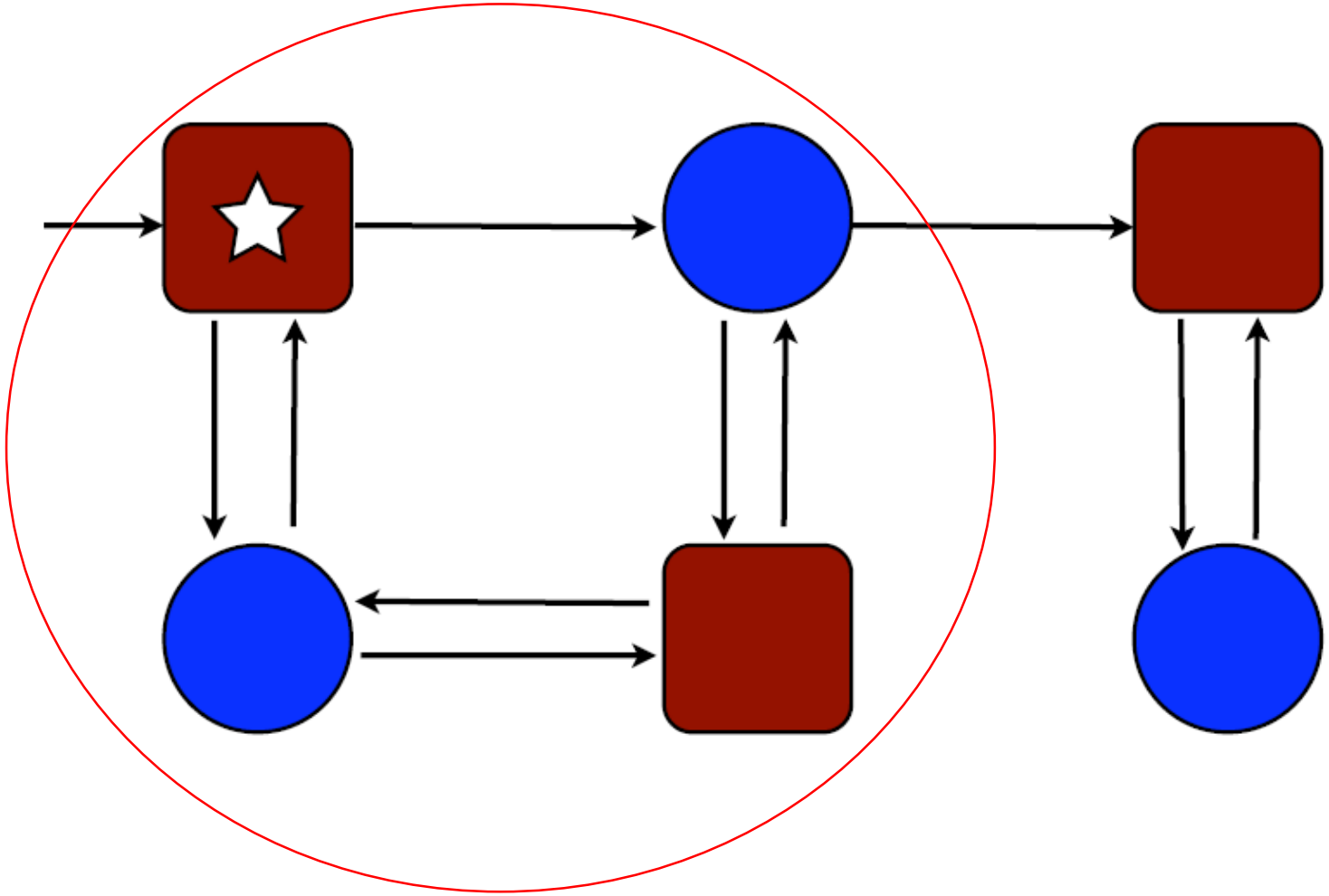
Theorem 2 (Reduction to a safety game). *Let* A *be a* tbU*K*CW *over inputs* Σ_I *and outputs* Σ_O *with n states* ($n > 0$)*, and let* $K = 2n(n^{2n+2} + 1)$*. The specification* A *is realizable iff Player O has a winning strategy in the game* $G(A,K)$*.*

Safety Game

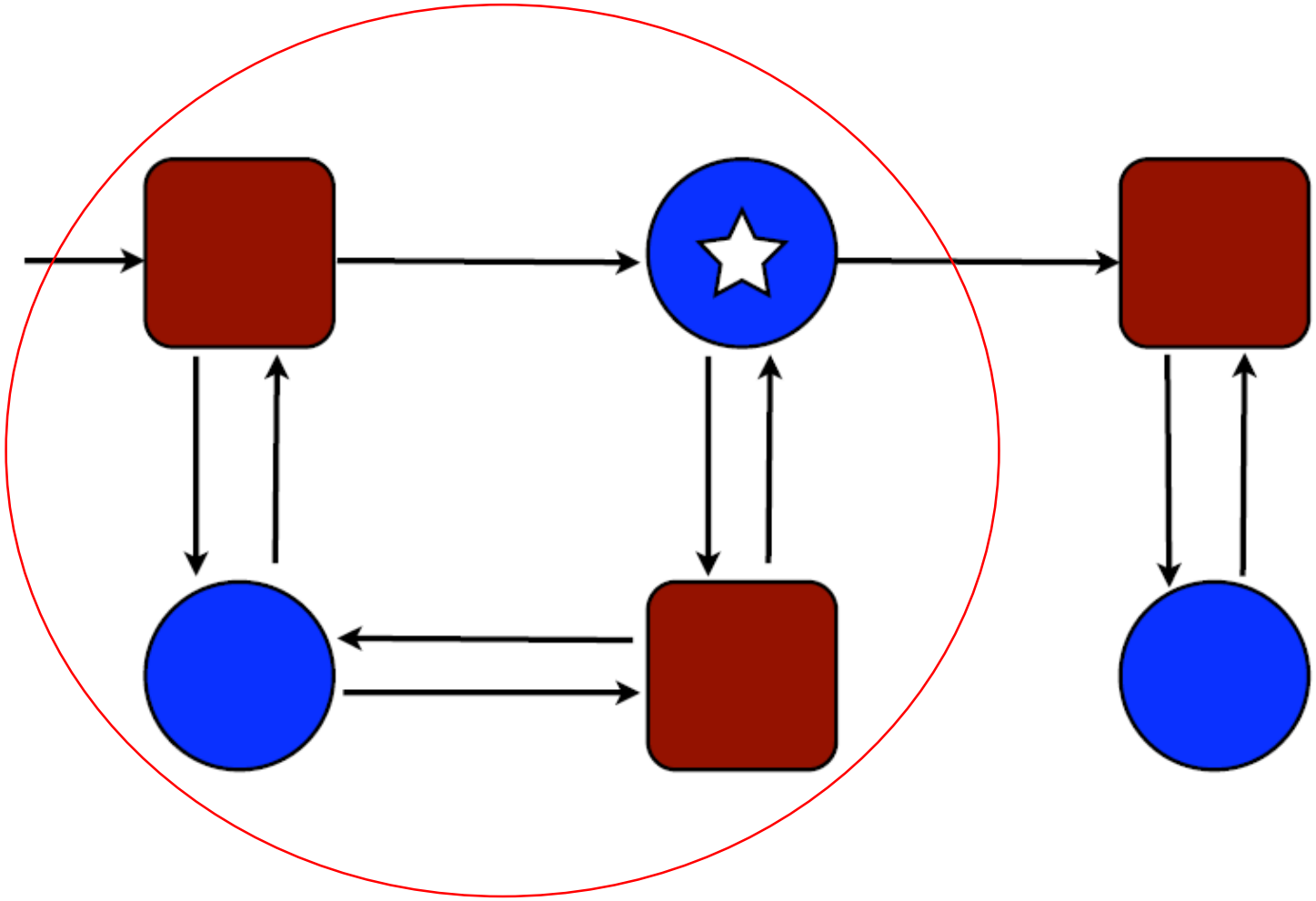
- *A game arena* is a tuple $G = (S_O, S_I, s_0, T,$ safe) where S_I, S_O are disjoint sets of player states, $s_0 \in S_O$ is the *initial state*, $T \subseteq S_O \times S_I \cup S_I \times S_O$ is the *transition relation* and safe is the *safety consition*.
- A *finite play* on G of length n is a finite word $\pi = \pi_0\pi_1 \ldots \pi_n \in (S_O \cup S_I)^*$

  s. t. $\pi_0 = s_0$ and for all $i = 0, \ldots, n-1$, $(\pi_i, \pi_{i+1}) \in T$.
- A *winning condition W* is a subset of $(S_O S_I)^*$.
- A *play* π is won by Player *O* if $\pi \in W$, otherwise it is won by Player *I*.
- A *strategy* λ_i for Player i ($i \in \{I,O\}$) is a *mapping* that maps any finite play whose last state s is in S_i to a state s' s. t. $(s, s') \in T$.
- The *outcome* of a strategy λ_i of Player i is the set $\text{Outcome}_G(\lambda_i)$ of infinite plays $\pi = \pi_0\pi_1\pi_2 \ldots$ s.t. for all $j \geq 0$, if $\pi_j \in S_i$, then $\pi_{j+1} = \lambda_i(\pi_0, \ldots, \pi_j)$.
- A strategy λ_O for Player *O* is *winning* if $\text{Outcome}_G(\lambda_O) \subseteq \text{safe}^\omega$.
  - Must void the *bad* states!

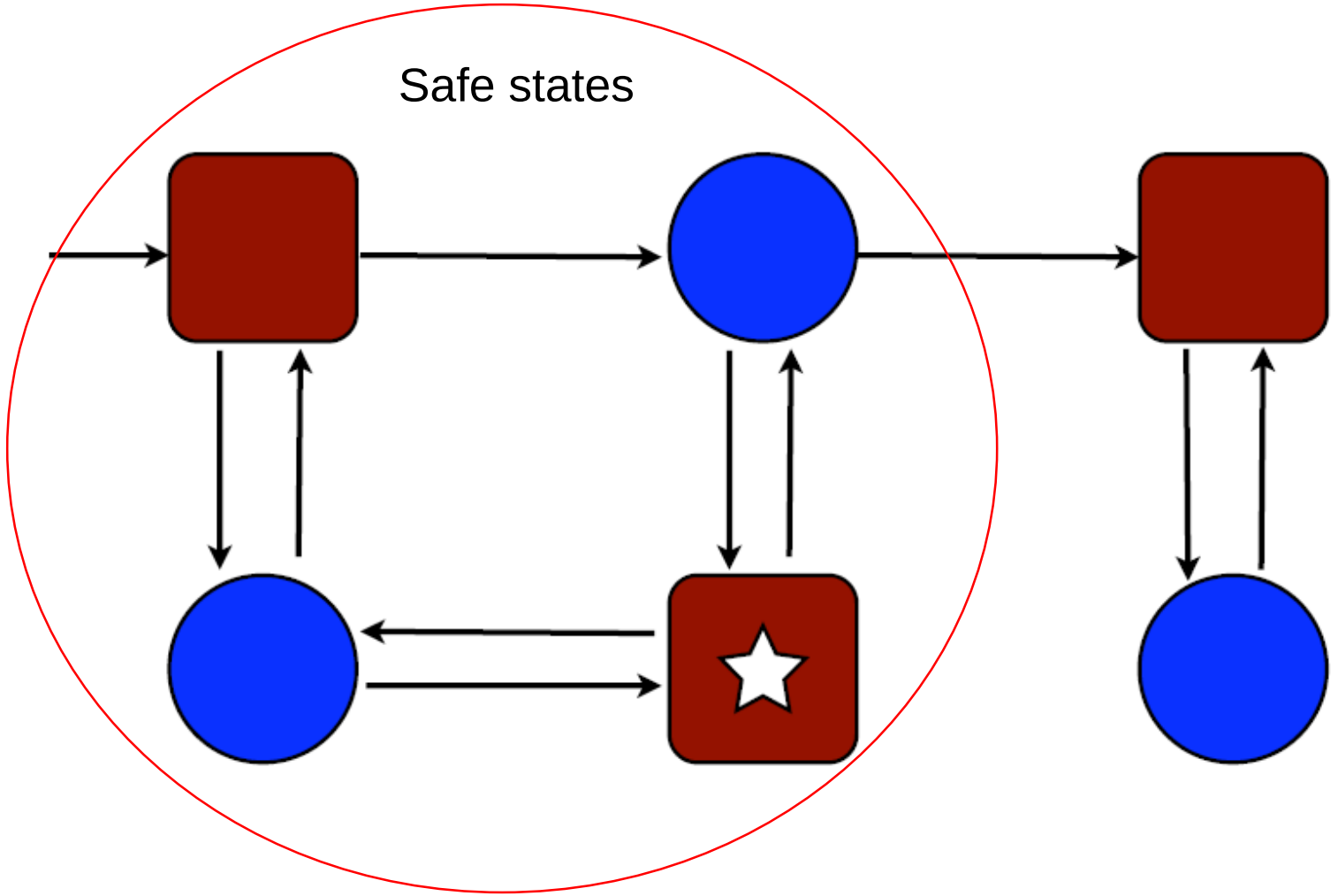
Safety Game



Safety Game



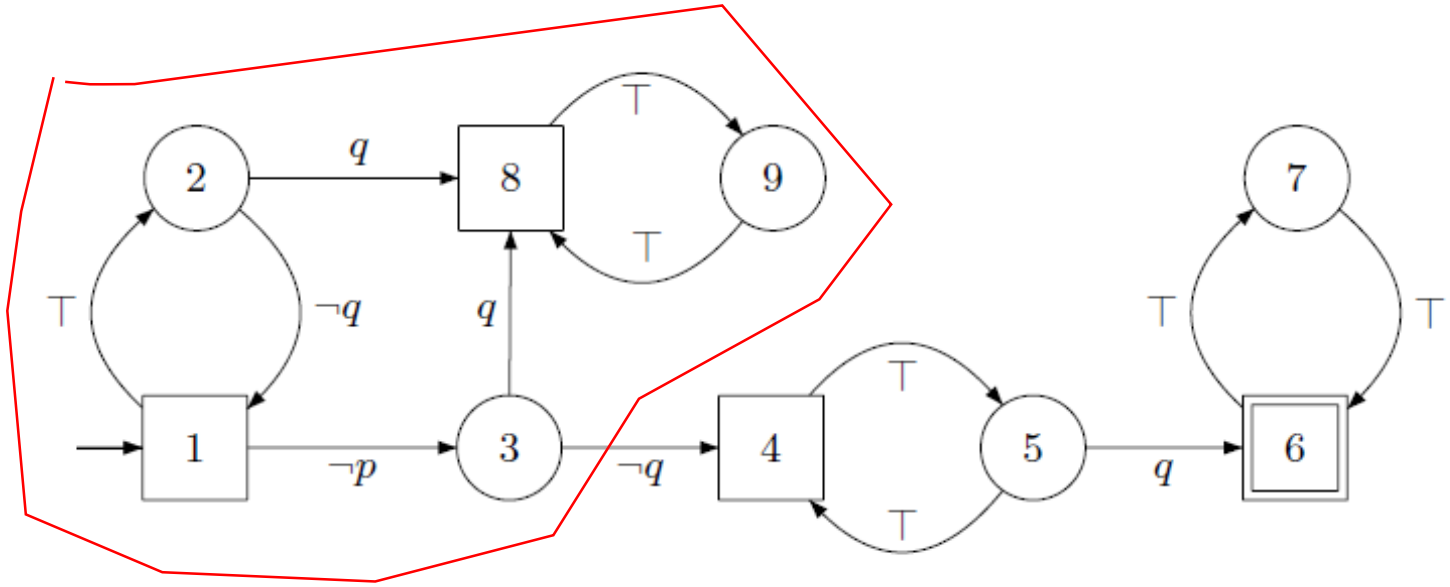
Safety Game



System controller wins if it has a strategy to keep the system in safe states.

Example of tbUCW

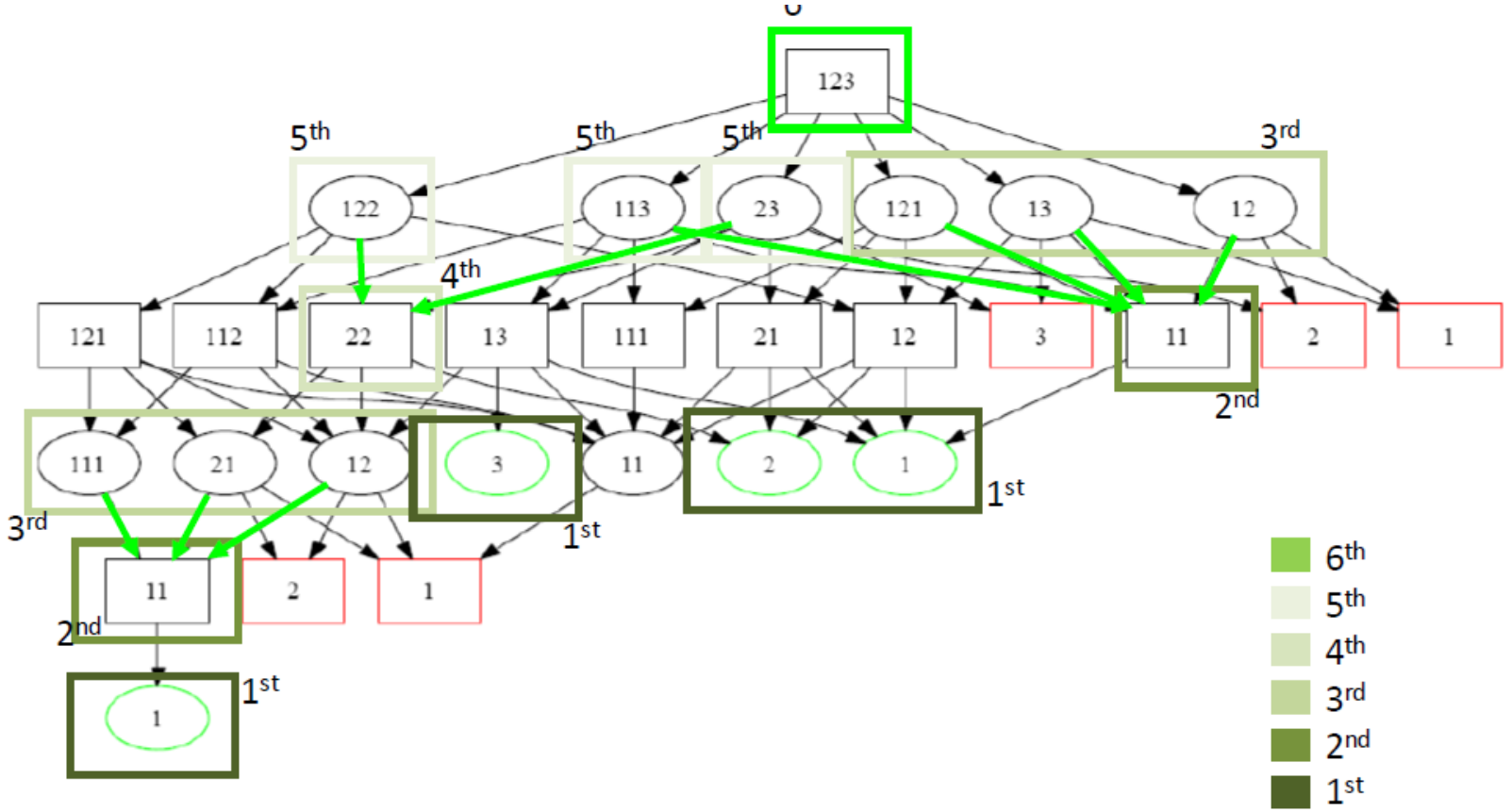
- tbUCW for **F***q* → (*p***U***q*) where $I = \{q\}$ and $O = \{p\}$
- Output states $Q_O = \{1, 4, 6, 8\}$ are depicted by squares and input states $Q_I = \{2, 3, 5, 7, 9\}$ by circles
- ⊤ stands for the sets Σ_I or Σ_O, depending on the context, ¬*q* (resp. ¬*p*) stands for the sets that do not contain *q* (resp. *p*), i.e. the empty set.
- At state 1, if controller does not assert *p* and next the environment does not assert *q*, then the run is in state 4. From this state, whatever the controller does, if the environment asserts *q*, then the controller loses, as state 6 will be visited infinitely often.



- A strategy for the controller is to assert *p* all the time, therefore the runs will loop in states 1 and 2 until the environment asserts *q*. Afterwards the runs will loop in states 8 and 9, which are non-final.

Solving safety games

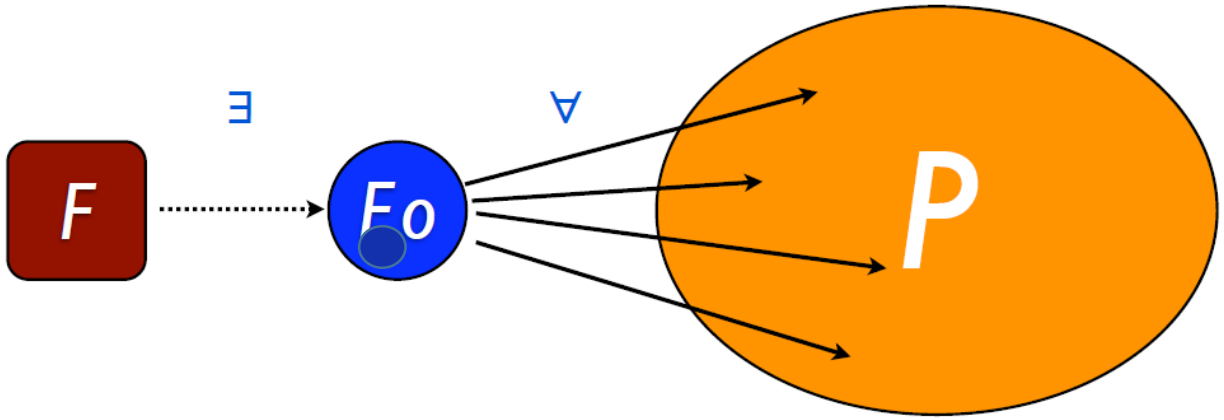
- Algorithms for solving safety games are constructed using the so-called *controllable predecessor operator*.



Solving safety games with Acacia+

- Let $G(A,K) = (\mathbb{F}_O, \mathbb{F}_I, F_0, T,$ safe) and set of all *counting functions* $\mathbb{F} \equiv \mathbb{F}_O \cup \mathbb{F}_I$.
- The controllable predecessor operator is based on the two following monotonic functions over the superset of the counting functions $2^{\mathbb{F}}$:
  - $\text{Pre}_I : 2^{\mathbb{F}O} \rightarrow 2^{\mathbb{F}I}$, $\text{Pre}_O : 2^{\mathbb{F}I} \rightarrow 2^{\mathbb{F}O}$.
- Let $P \subseteq \mathbb{F}$ be a subset of system positions. The safe controllable predecessors of P are then:

$$\text{CPre}(P) = \{F \mid \exists o \subseteq O, \forall F', ((Fo),F') \in T \Rightarrow F' \in P\} \cap \text{safe}$$



Properties of the controllable predecessor - 1

- Let CPre $=$ Pre$_O \circ$ Pre$_I$. Function CPre is monotonic over the *complete lattice* $(2^{F_O}, \subseteq)$, and so it has a ***greatest fixed point*** denoted by CPre*.

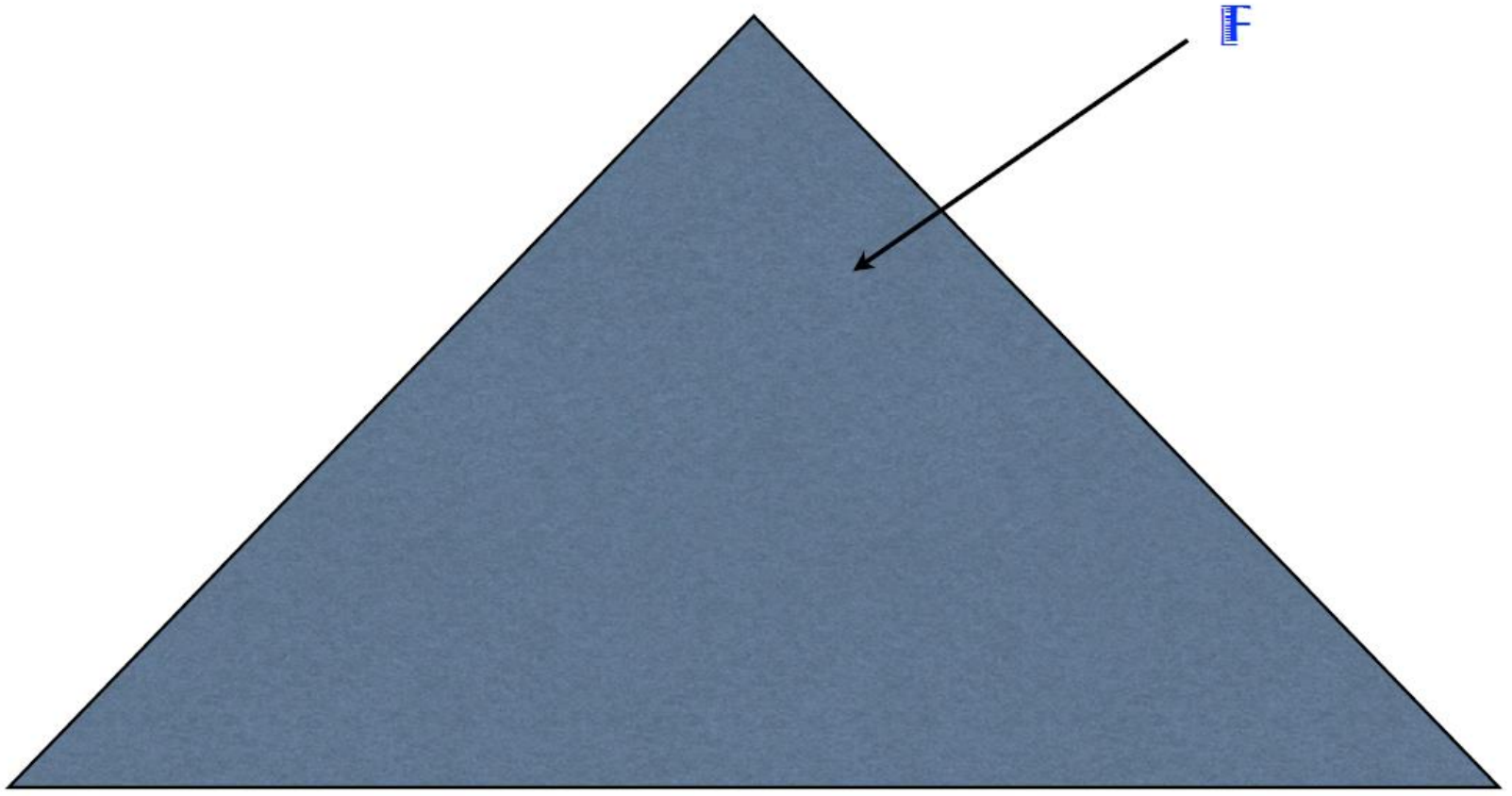
Theorem. *The set of states from which Player O* (the system) *has a winning strategy in G*(A,K) *is equal to* CPre**.*

- By Theorem for the Reduction to a Safety Game, system has a winning strategy in *G(A,K)* iff the initial state $F_0 \in$ CPre*.

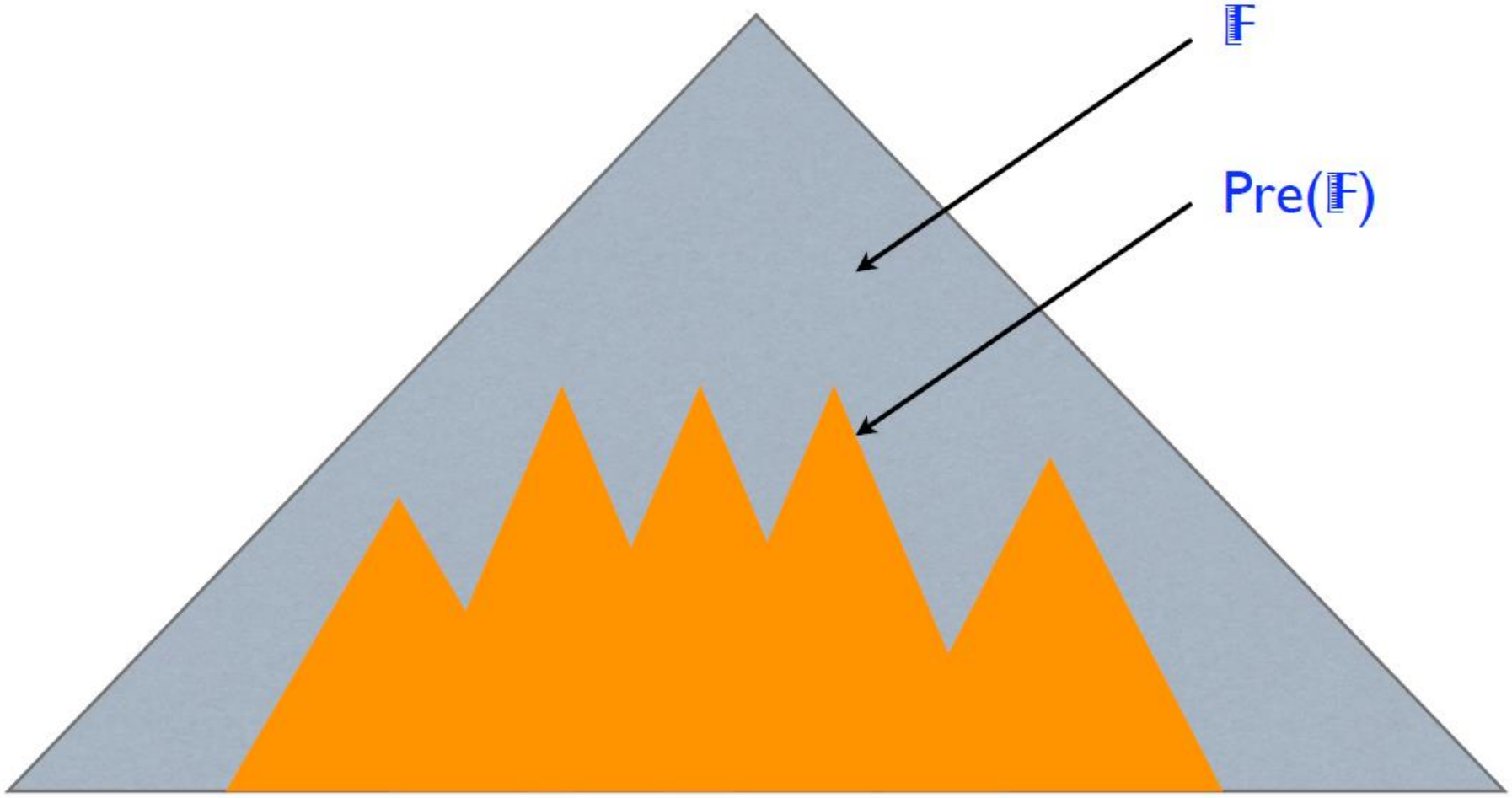
Properties of the controllable predecessor - 2

- $\mathbb{F}$ can be *partially ordered* by $F \preccurlyeq F'$ iff $\forall q, F(q) \leq F'(q)$.
  - If system wins from F', it can also win from F.
- CPre() preserves *downward*-closed sets.
  - A set $S \subseteq \mathbb{F}$ is *closed for* $\preccurlyeq$, if $\forall F \in S \ \cdot \ \forall F' \preccurlyeq F \ \cdot \ F' \in S$.
  - For all *closed* sets $S \subseteq \mathbb{F}$, the closure of S denoted by $\downarrow S$, is equal to S.
- A set $S \subseteq \mathbb{F}$ is an *antichain* if all elements of S are incomparable for $\preccurlyeq$.
- The set of *maximal elements* of S is an **antichain**, $\boldsymbol{S} = \{F \in S \mid \nexists F' \in S \cdot F' \neq F \wedge F \preccurlyeq F\}$.
- For Acacia+ antichains are a compact and efficient representation to manipulate closed sets in $\mathbb{F}$.
- Each (downward) set of the fixpoint computation is represented by its maximal elements.

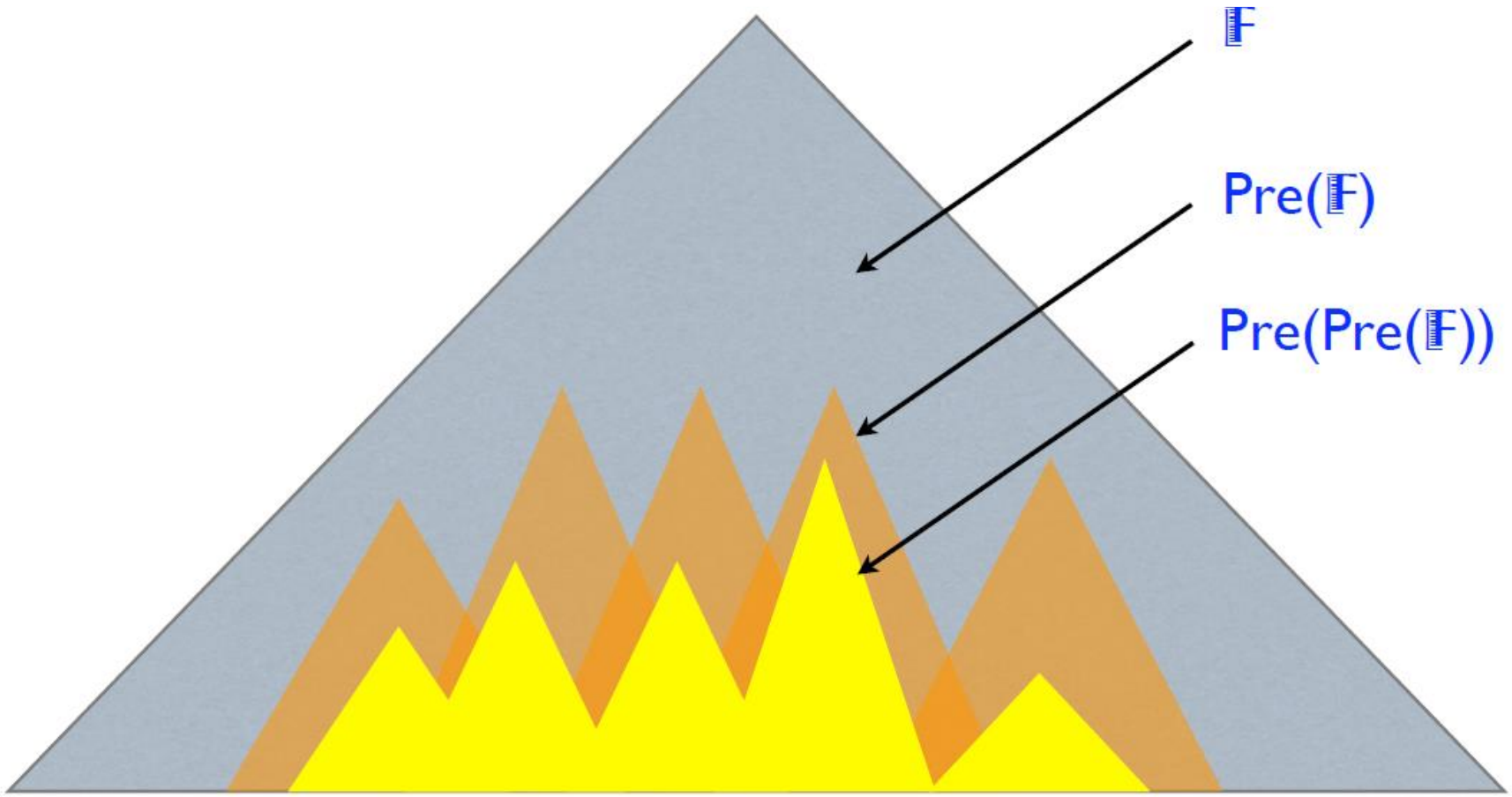
Symbolic Fixpoint Computation



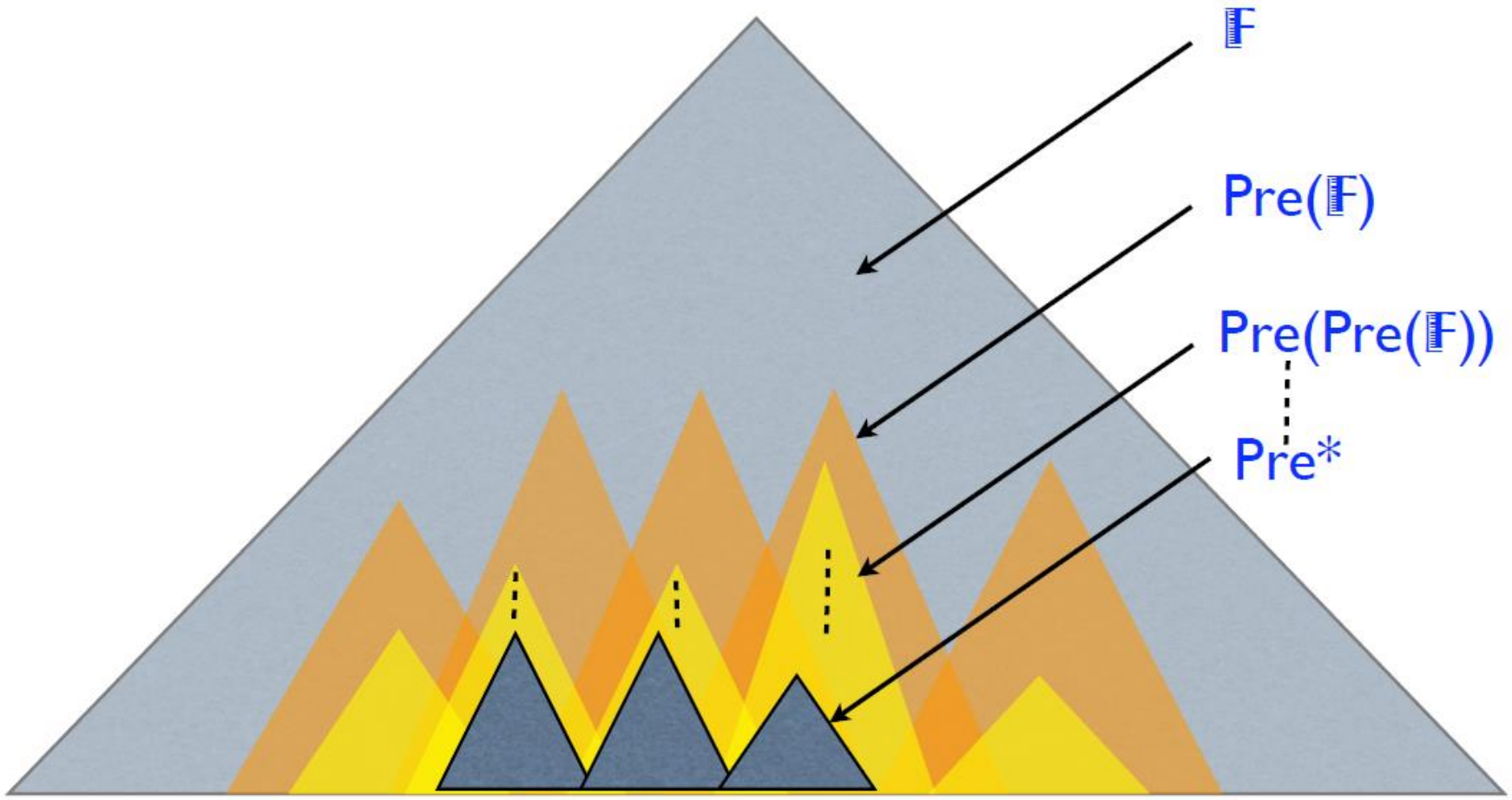
Symbolic Fixpoint Computation



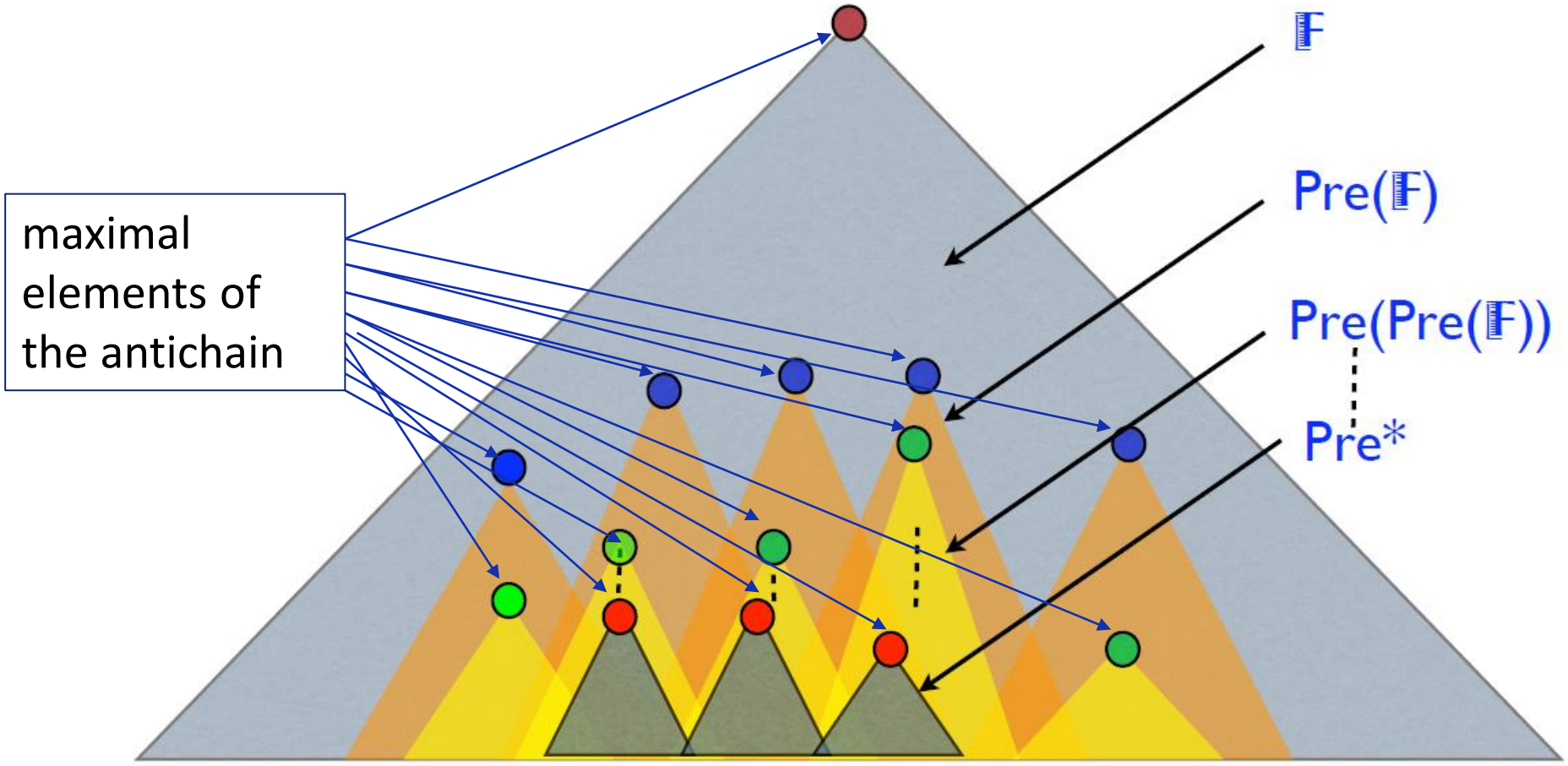
Symbolic Fixpoint Computation



Symbolic Fixpoint Computation



Symbolic Fixpoint Computation



Incremental realizability checking

- For checking the existence of a winning strategy for Player *O* in the safety game, the following property of U*K*CWs:

for all K_1, K_2 · $0 \leq K_1 \leq K_2$ · $L_{uc,K1}(A) \subseteq L_{uc,K2}(A) \subseteq L_{uc}(A)$.

1. **Input**: an LTL formula Φ, a partition I,O
2. A ← UCW with n states equivalent to Φ
3. $K \leftarrow n^{2n+2}$
4. **for** $k=0...K$ **do**
5. **if** System wins then $G(A,k)$ **return** realizable
6. **endfor**
7. **return** unrealizable

Not reasonable to test for unrealizable specifications. Need to reach the upper bound for *K*.

Unrealizability Checking

- As a consequence of the determinacy theorem for Borel games:
- φ is unrealizable for the System iff $\neg\varphi$ is realizable for the Environment.
- The previous algorithm is adapted to test unrealizability.
- Realizability by Player *O* of φ is checked, and *in parallel* realizability by Player *I* of $\neg\varphi$, incrementing the value of *K*.
- When one of the two processes stops, it is known if φ is realizable or not.
- In practice, realizability or unrealizability are obtained for small values of *K*.

Synthesis of winning strategies

- If a formula φ is realizable, *extract* from the *greatest fixpoint computation* a Moore machine that realizes it.
- Let $\Pi_I \subseteq \mathbb{F}_I \cap$ safe and $\Pi_O \subseteq \mathbb{F}_O \cap$ safe be the two sets obtained by the *greatest fixpoint computation*.
- $\mathrm{Pre}_O(\Pi_I) = \Pi_O$, $\mathrm{Pre}_I(\Pi_O) = \Pi_I$ --- Π_I and Π_O are *downward-closed.*
- By definition of Pre_O for all $F \in [\Pi_O]$, $\exists\ \sigma_F \in \Sigma$ such that

  $\mathrm{succ}(F, \sigma_F) \in \Pi_I$, and this σ_F can be computed.
- A Moore machine can be extracted:
  - set of states is $[\Pi_O]$,
  - the *output function* maps any state $F \in [\Pi_O]$ to σ_F ,
  - the *transition function* maps F to a partially-ordered state F' according to the succ operator,
  - and the *initial state* F_0 is a state partially-ordered with F.

Example of Moore machine synthesis

- tbUCW for **F***q* → (*p***U***q*)
- Start with the safe state in the game for the system, denoted by

*F*1 = (1 → 1, 4 → 1, 6 → 1, 8 → 1).

- Then, for the system predecessor (Env.),

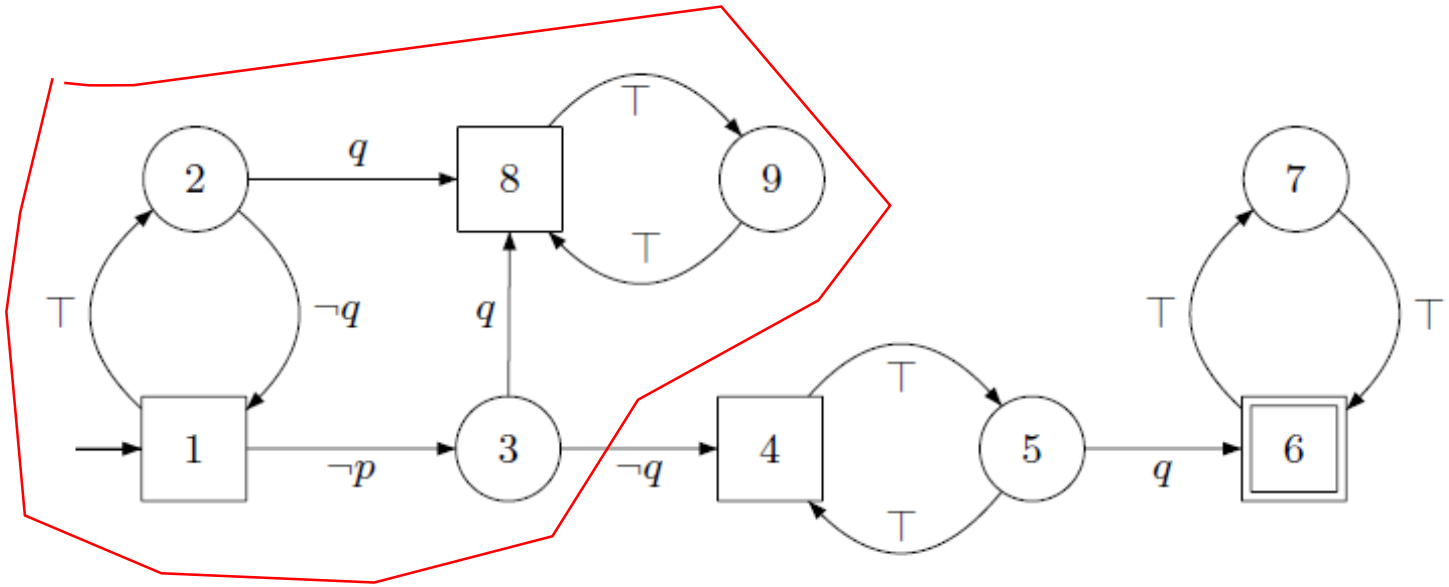
*F*2 := (2 → 1, 3 → 1, 5 → 0, 7 → 0, 9 → 1)

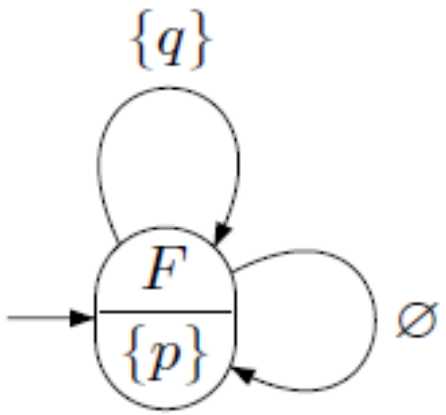
- Then for the controlled (System) predecessor

CPre = (1 → 1, 4 → 0, 6 → 0, 8 → 1)

- At end of computation, the fixpoint is:

F := (1 → 1, 4 → −1, 6 →−1, 8 → 1)





Forward algorithm for solving games

- In Acacia+ also a *forward* algorithm can be applied to solve games.
- Compared to the backward algorithm, the *forward* algorithm has the advantage that it computes ***only*** the winning positions *F* (for the System) which are reachable from the initial position.
  - But it can compute only one winning strategy.
- The algorithm explores the positions of the game and *once a position is known to be losing*, this information *is back propagated to the predecessors*.
- A position of Player System is *losing* iff it has *no* successors or *all* its successors are losing.
- A position of Player Env. is losing iff *one of* its successors is losing.

Forward algorithm – Sketch (1)

- At each step, maintain an *under-approximation* **Losing** of the set of losing positions.
- A waiting-list **Waiting** for reachable position exploration and re-evaluation of positions is used.
- An edge is put in the *waiting*-list if it is the first time it has been reached, or the status of its target position has changed.
- If a position is known that is losing, this is back-propagated to all its predecessors.
- A set **Passed** records the visited positions.
- a set **Depend** stores the edges (*s, s´*) which need to be re-evaluated when the value of *s´* changes.

Forward algorithm – Sketch (2)

- At each step, pick an edge *e* = (*s, s´*) in the *waiting*-list.
- If its target *s´* has never been visited, check if this target is losing
  - When it has no successors.
- If losing, add *e* in the *waiting*-list for re-evaluation.
  - Back propagate the information on *s´*.
- Otherwise add all the successors of *s´* in the *waiting*-list for re-evaluation.
- If *s´* has already been visited, then compute the value of *s*.
- If *s* is losing, this information is back propagated to the positions whose *safeness* depends on *s*.

Compositional safety games and LTL synthesis

- Acacia+ implements a compositional approach for synthesis of large conjunctions of LTL formulas.
- Realistic systems cannot be specified by just a couple of simple LTL formulae.
- A scalable approach is very beneficial!

Overview of compositional algorithms

- Two *compositional* algorithms for LTL formulas of the form

  $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_n$ are implemented in Acacia+.

- *Backward* algorithm: At each stage of the *parenthesizing*, the antichains W_i of the subformulae φ_i are computed backward and the antichain of the formula φ itself is also computed backward from the W_i's.
  - **All** winning strategies for φ are computed and compactly represented by the final antichain.
- *Forward* algorithm: At each stage of the *parenthesizing*, the antichains W_i of the subformulae φ_i are computed backward, **except** at the last stage where a forward algorithm seeks for **one** winning strategy by exploring the game arena *on the fly* in a forward fashion.

Compositional safety games

- Compositional reasoning on safety games is supported by the existence of a most permissive strategy, a *master plan*.
- The *master plan* of System can be interpreted as a *compact representation* of **all** the winning strategies of System against the Environment.
  - It contains **all** the moves that System can play in a state *s* in order to win the safety game.
- The master plan associated with a game can be computed in a *backward* fashion by using variants of the controllable operator CPre and sequence of positions *W.*

Composition of safety games

- Let G^i , $i \in \{1, \ldots, n\}$, be n safety games $G^i = (S^i_1, S^i 2, \Gamma^i_1, \Delta^i_1, \Delta^i_2)$ defined on the same sets of moves, Moves = $\text{Moves}_1 \uplus \text{Moves}_2$.

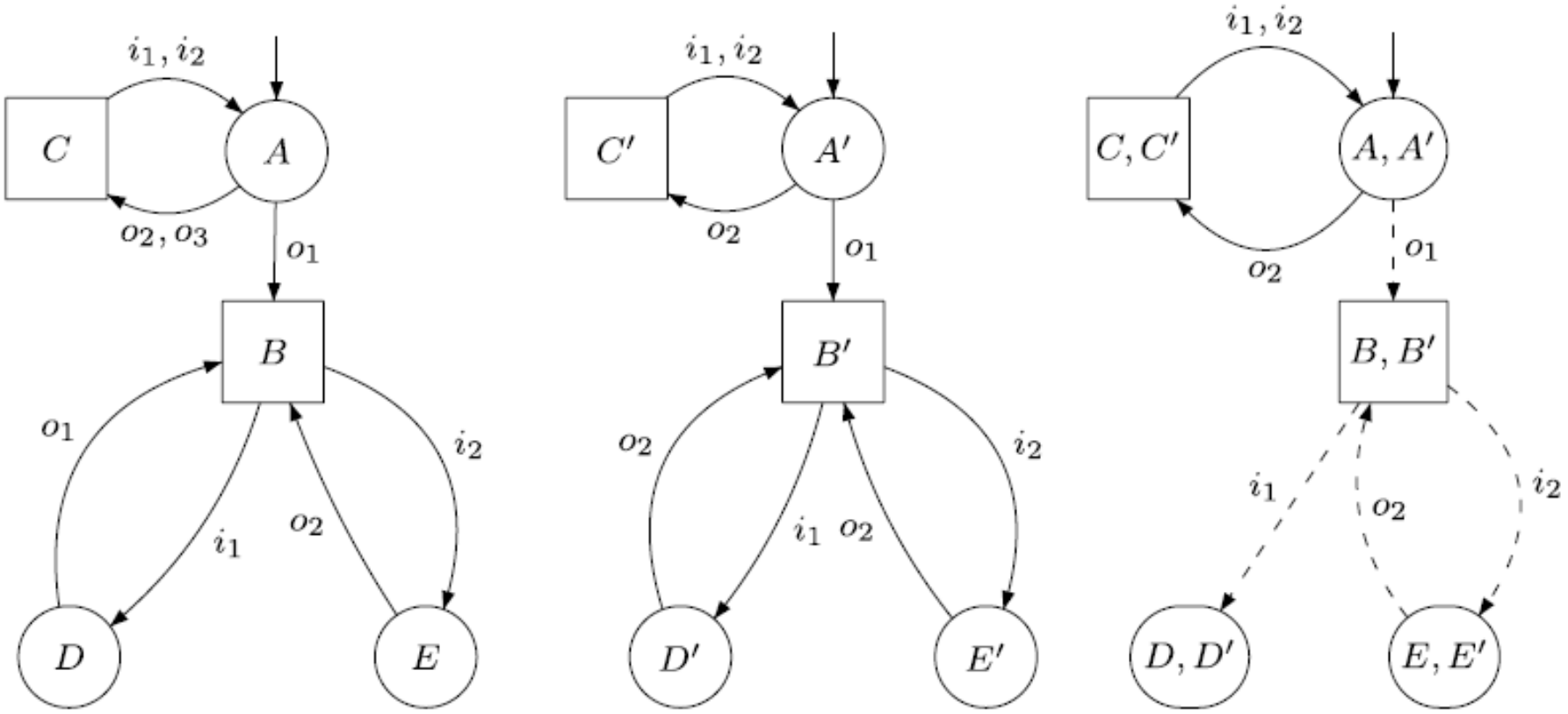
- Their product is the safety game $G^{\otimes} = (S^{\otimes}_1 , S^{\otimes}_2 , \Gamma^{\otimes}_1 , \Delta^{\otimes}_1 , \Delta^{\otimes}_2)$ over the *product of the state spaces* of the players, the *intersection (common) winning strategies* of the System and the *transitions* conforming to the winning strategy of System or to the moves of the Environment.

Backward compositional solving of $G^{\otimes}$

- First, compute locally the master plans of the components.
- Then compose the local master plans and apply one time the CPre operator to this composition to compute a function that contains information about the one-step inconsistencies between local master plans.
- Project back on the local components the information gained by the function , and iterate.

Forward compositional solving of G ⊗

- Interested in computing a master plan only for the winning and *reachable* positions, **common for all** sub-games. **Example**:



Game G_1 Game G_2 $(G_1 \otimes G_2)[\mathsf{MP}_{\mathsf{Reach}}(G_1 \otimes G_2)]$

Compositional LTL synthesis

- When a formula is given as a *conjunction* of subformulas i.e.,

  $\psi = \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n$ the safety game associated with this formula can be defined *compositionally*.
- For each subformula φ_i the corresponding tbUKCW $A_{\varphi i}$ on the alphabet of ψ is constructed and also their associated safety games $G(\varphi_i, K)$.
  - The notion of product is used at the level of turn-based automata.
  - Executing the $A_1 \otimes A_2$ *o*n a word *w* is equivalent to execute both A_1 and A_2 on this word.
- The game $G(\psi, K)$ for the conjunction ψ is *isomorphic* to the game composition.
- The game is then solved compositionally by first computing the local master plans to finally produce a compact (global) Moore machine, if it exists.

About the full exam for this part of the course

- Possible questions for the final exam (closed book) for this part of the course will be around these slides.
  - With focus on Acacia+ background and preliminaries.
    - Need to read the papers.
    - If a question will be around the proofs of the theorems/lemmas, only abstract elaboration/sketch of proof will be enough for answering.
  - Possibly a question regarding the Safra-based old approach.
    - The elaboration is found in the two Acacia+ papers.
    - No need to learn the details from the original paper.
  - Possibly a question will be about improving (filling up) some (not complex) spec in LTL to make it realizable.

About the open book test for this part

- Most of the questions will be around LTL.
  - Formalising some requirements with LTL.
  - Translating LTL to plain English.
  - Correspondence between LTL formula and automaton.
  - ...
- Possibly a question will be about improving (filling up) some (not complex) spec in LTL to make it realizable.

References

- An Antichain Algorithm for LTL Realizability . http://lit2.ulb.ac.be/acaciaplus/slides/cav09.pdf
- Filiot E., Jin N., Raskin JF. (2009) An Antichain Algorithm for LTL Realizability. In: Bouajjani A., Maler O. (eds) Computer Aided Verification. CAV 2009. Lecture Notes in Computer Science, vol 5643. Springer, Berlin, Heidelberg.
- A. Pnueli. The temporal logic of programs. In Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS '77). IEEE Computer Society, Washington, DC, USA, 46-57. DOI: https://doi.org/10.1109/SFCS.1977.32, 1977.
- Filiot, E., Jin, N. & Raskin, JF. Antichains and compositional algorithms for LTL synthesis, Form Methods Syst Des (2011) 39: 261. https://doi.org/10.1007/s10703-011-0115-3
- S. Safra, On the complexity of ω-automata. In: Proc. 29th Annual Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society Press (1988).
- A. Pnueli and R. Rosner. On the synthesis of a reactive module. In Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '89) ACM, NY, USA, 179-190. DOI=http://dx.doi.org/10.1145/75277.75293, 1989
- M. Y. Vardi, The Siren Song of Temporal Synthesis
  - http://www.dis.uniroma1.it/~kr18actions/slides/invitedMosheVardi.pdf
- N. Piterman, Games and Synthesis, EATCS Young Researchers School, 2014
  - http://eatcs-school.fi.muni.cz/_media/piterman.pdf