CS 4830/7485
System Specification, Verification and Synthesis
Fall 2019

Controller and Program Synthesis

Stavros Tripakis

Northeastern University
Khoury College of
Computer Sciences

From verification to synthesis

Verification:

first write program (or model of a system), then specify formal properties, then check correctness.

Synthesis:

first specify formal properties, then let synthesizer automatically generate a correct program.

Put another way:

from imperative (how) to declarative (what) design;

"*raising the level of abstraction*".

What is synthesis?

Roughly:

$$\exists P: \forall x: \varphi(x, P(x))$$

Many different variants, depending on what is P, φ, and how search is done.

Very old topic (Church, 1960s) recently rejuvenated.

Program synthesis and proofs

From 2nd order formula

$$\exists P: \forall x: \varphi(x, P(x))$$

to 1st order formula

$$\forall x: \exists y: \varphi(x, y)$$

Synthesizing program P can be done by proving ***constructively*** that the above formula is valid.

Deductive program synthesis.

PPDP 2010: “Dimensions in Program Synthesis”, Gulwani.

Dimensions in Synthesis (Gulwani)

Concept Language (Application)

- Programs
  - Straight-line programs
- Automata
- Queries
- Sequences

Also: logic synthesis

User Intent (Ambiguity)

- Logic, Natural Language
- Examples, Demonstrations/Traces

Search Technique (Algorithm)

- SAT/SMT solvers (Formal Methods)
- A*-style goal-directed search (AI)
- Version space algebras (Machine Learning)

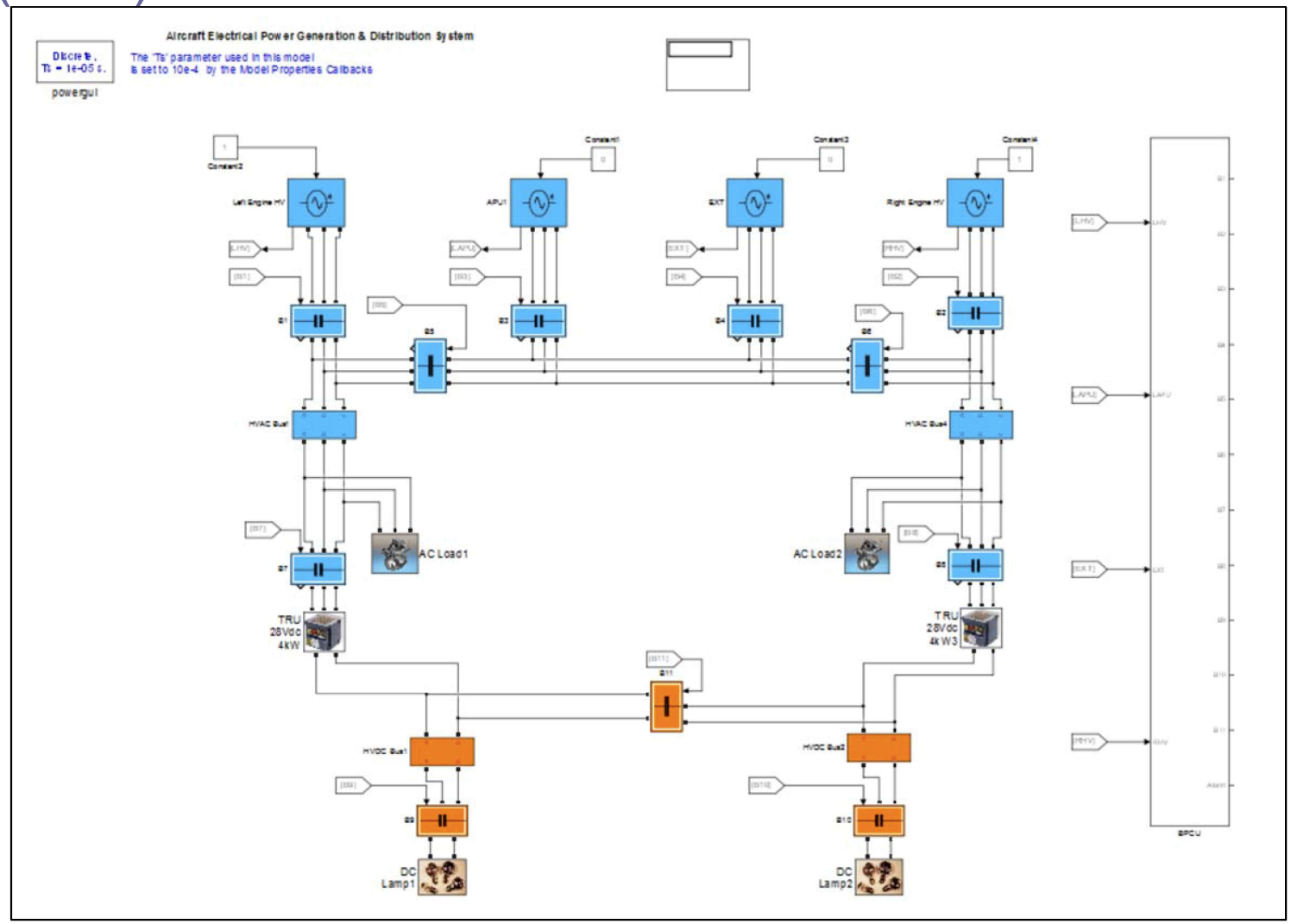
Compilers vs. Synthesizers (Gulwani)

Dimension	Compilers	Synthesizers
Concept Language	Executable Program	Variety of concepts: Program, Automata, Query, Sequence
User Intent	Structured language	Variety/mixed form of constraints: logic, examples, traces
Search Technique	Syntax-directed translation (No new algorithmic insights)	Uses some kind of search (Discovers new algorithmic insights)

MOTIVATING EXAMPLE

Designing controllers can be tricky and time consuming

Example: Electrical Power Generation and Distribution System (EPS) of a modern aircraft



Thanks to:
Pierluigi Nuzzo
Antonio Iannopollo

Designing controllers can be tricky and time consuming

Example: EPS requirements (in English)

Assumptions:

A2) At least one power source is always "healthy" (i.e. it is operational and can be inserted into the network to deliver power);

A3) Failures can only affect the power sources; once a power source becomes "unhealthy" (i.e. it is not operational and cannot be inserted into the network to deliver power), it will never return to be "healthy" (e.g., turned back on) during the cruising phase of the mission;

A4) An AC bus is correctly powered if the root-mean-square (RMS) voltage at its loads is between 110 V and 120 V and the frequency is 400 Hz.

Under the above assumptions, the BPCU offers the following guarantees:

Guarantees:

G1) At start-up all the power source contactors are "open";

G2) In normal conditions (i.e. no faults or failures in the system) G_L and G_R are "on" and provide power for the left side and the right side of the system, respectively; auxiliary power units are "off"; C_9 and C_{10} are open ("off");

G3) No AC bus is powered by more than one power source at the same time, i.e. AC power sources can never be paralleled;

G4) It never happens that both the APUs are inserted into the network at the same time;

G5) AC buses cannot be unpowered for more than a well-defined length of time;

G6) DC buses must always stay powered, at least in a "reduced performance" mode, which occurs when only one HVRU is used;

G7) The left AC bus B_1 must always be powered from the first available source from the ordered list (G_L, A_L, A_R, G_R);

G8) The right AC bus B_2 must always be powered from the first available source from the ordered list (G_R, A_R, A_L, G_L).

Designing controllers can be tricky and time consuming

Example: EPS requirements (in English) – zooming in

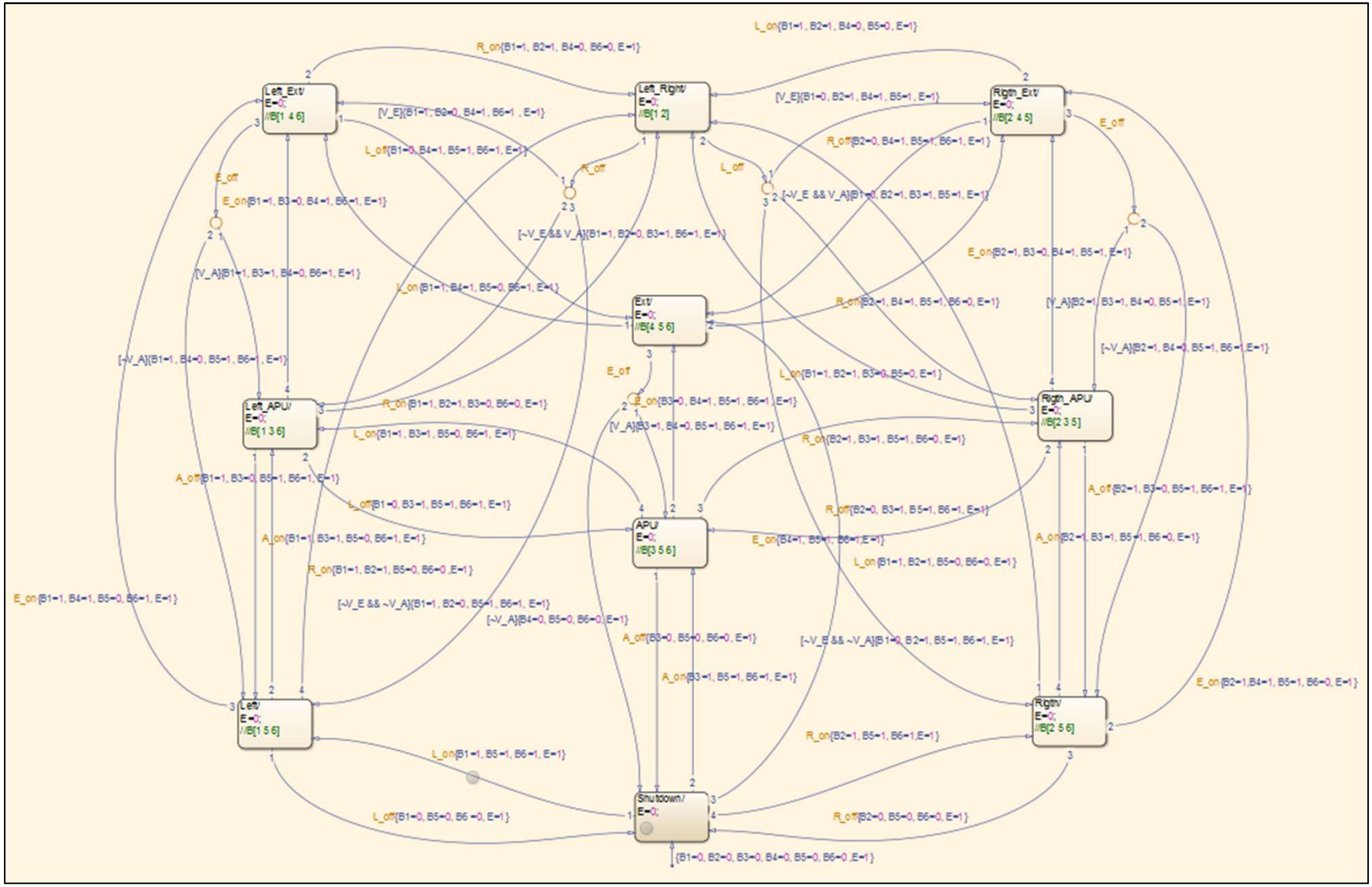
A2) At least one power source is always "healthy" (i.e. it is operational and can be inserted into the network to deliver power);

G1) At start-up all the power source contactors are "open";

G3) No AC bus is powered by more than one power source at the same time, i.e. AC power sources can never be paralleled;

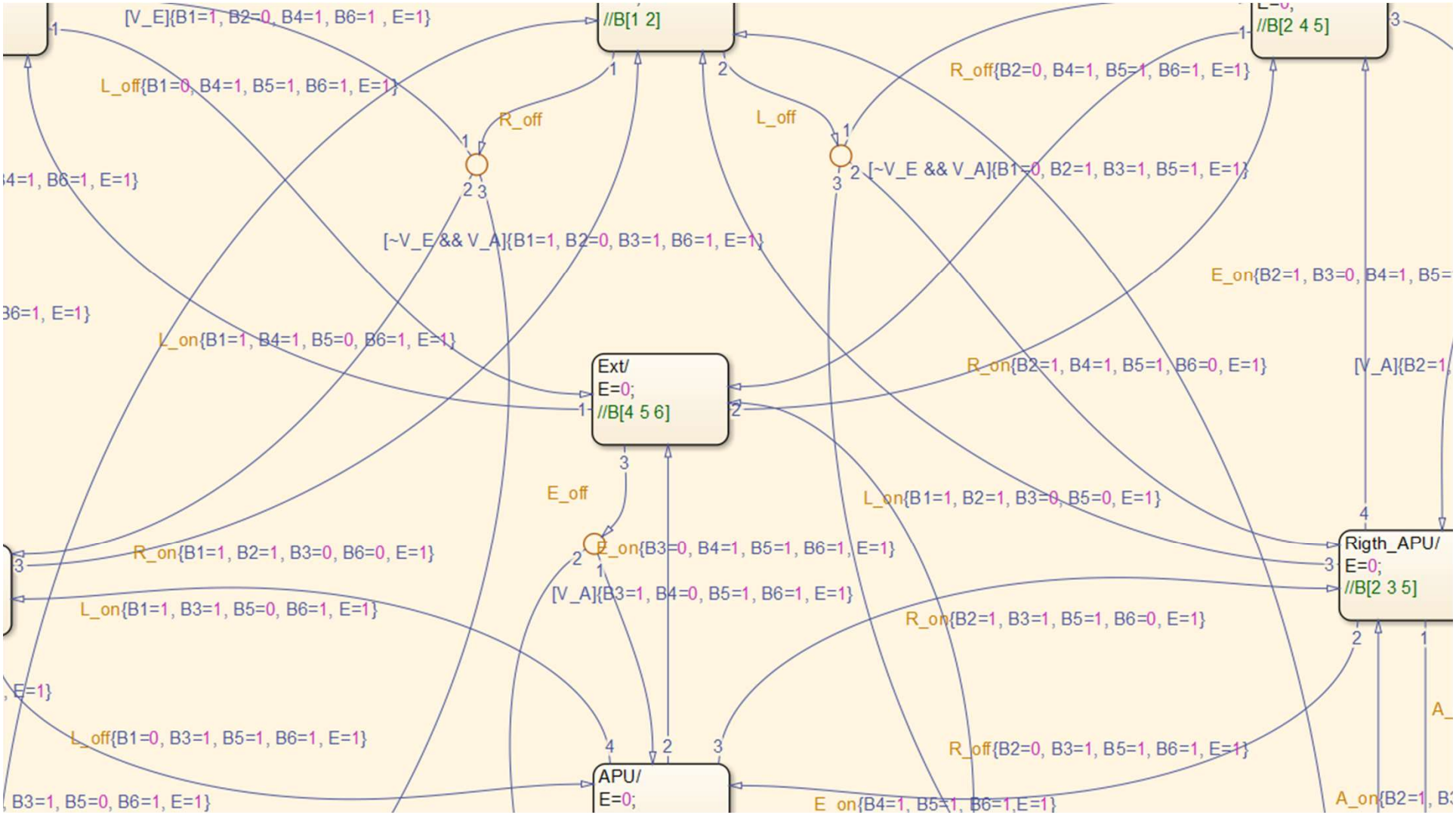
Designing controllers can be tricky and time consuming

Example: EPS “hand-written” controller



Designing controllers can be tricky and time consuming

Example: EPS “hand-written” controller – zooming in



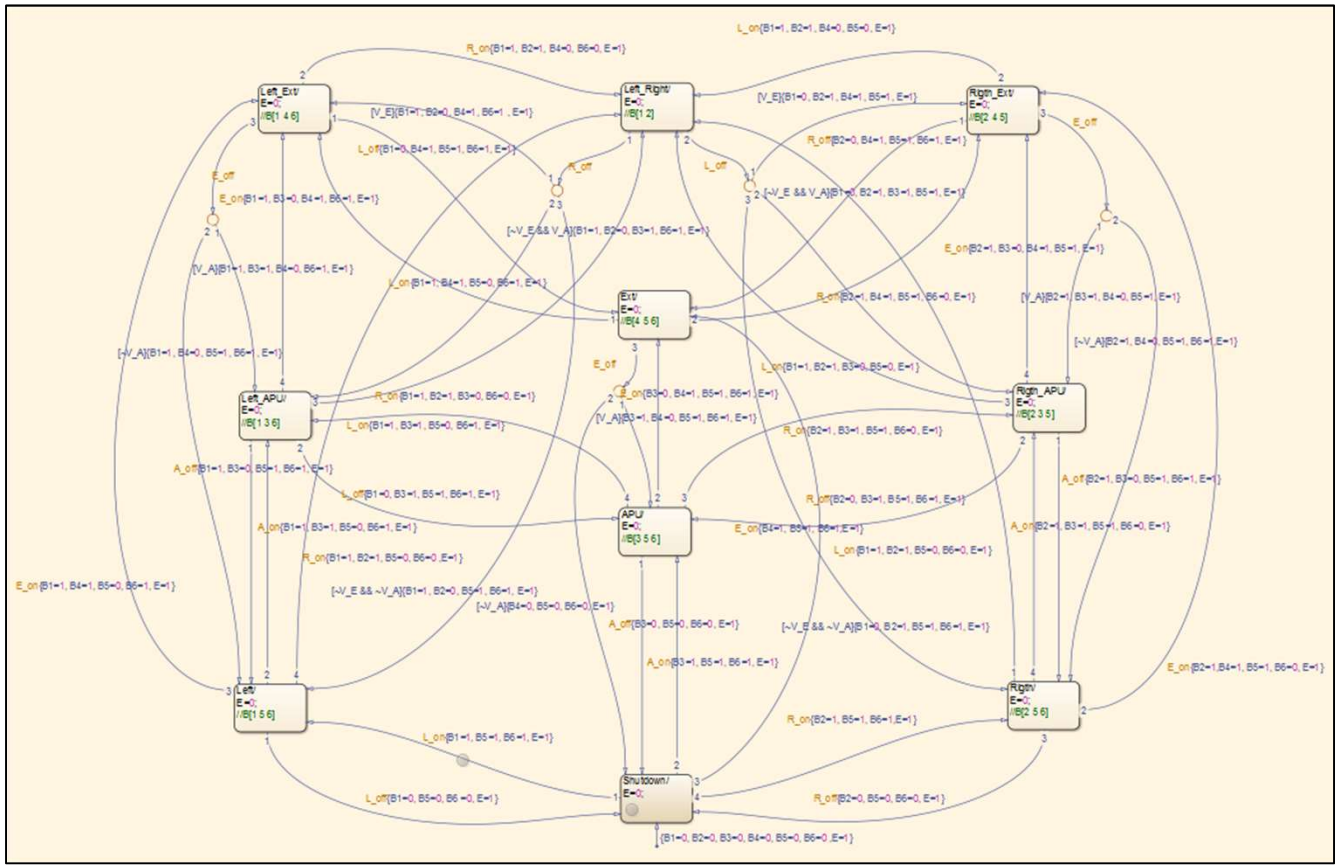
Designing controllers can be tricky and time consuming

Example: EPS “hand-written” controller

Design time ~ 1 week [Nuzzo] (but have to verify also)

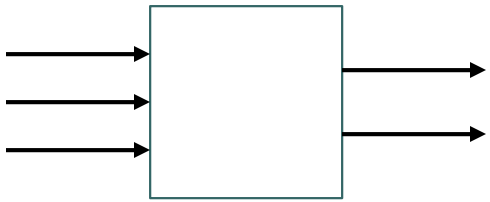
For a real controller, it could be months [e.g., robotic controllers, Willow Garage]

Can design time be improved?



Declarative specification of controllers

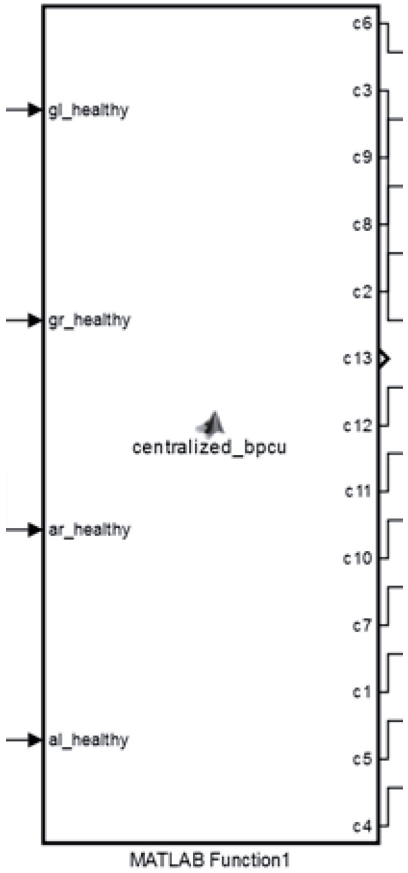
At the outset the controller is just a box with inputs and outputs:



Declarative specification of controllers

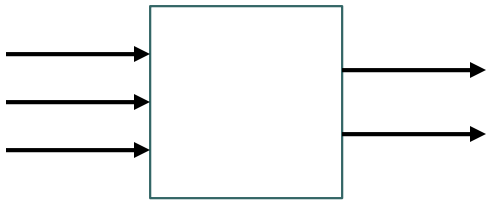
At the outset the controller is just a box with inputs and outputs:

Example: EPS controller



Declarative specification of controllers

At the outset the controller is just a box with inputs and outputs:



We can specify the input-output behavior of the controller in a high-level language, e.g., in **temporal logic**.

Declarative specification of controllers

Example: LTL specification for EPS

~40 lines

```
#Assumptions
(gl_healthy & gr_healthy & al_healthy & ar_healthy)
[](gl_healthy | gr_healthy | al_healthy | ar_healthy)
[](!gl_healthy -> X(!gl_healthy) )
[](!gr_healthy -> X(!gr_healthy) )
[](!al_healthy -> X(!al_healthy) )
[](!ar_healthy -> X(!ar_healthy) )

#Guarantees
(!c1 & !c2 & !c3 & !c4 & !c5 & !c6 & !c7 & !c8 & !c9 & !c10 &
!c11 & !c12 & !c13)
[](X(c7) & X(c8) & X(c11) & X(c12) & X(c13))

[](!(c2 & c3))
[](!(c1 & c5 & (al_healthy | ar_healthy)))
[](!(c4 & c6 & (al_healthy | ar_healthy)))
[]((X(gl_healthy) & X(gr_healthy) ) -> X(!c2) & X(!c3) & X(!c9) &
X(!c10))
[]((X(!gl_healthy) & X(!gr_healthy) ) -> X(c9) & X(c10))

[](X(!gl_healthy)-> X(!c1) )
[](X(!gr_healthy)-> X(!c4) )
[](X(!al_healthy)-> X(!c2) )
[](X(!ar_healthy)-> X(!c3) )

[](X(gl_healthy) -> X(c1) )
[](X(gr_healthy) -> X(c4) )

…
```

```
#Guarantees
…

[](!gl_healthy -> X(c5))
[](!gr_healthy -> X(c6))

[]((X(gl_healthy)  & X(gr_healthy) ) -> (X(!c5) & X(!c6) ))

[]((X(!gl_healthy) & X(al_healthy)  & X(gr_healthy) ) -> (
X(c2) & X(c3)) )


[]((X(!gl_healthy)  & X(!gr_healthy)  & X(al_healthy)  & !c3
& !c2) -> X(c2) )

[]((X(al_healthy)  & c2) -> X(c2) )
[]((X(ar_healthy)  & c3) -> X(c3) )

[]((X(!gl_healthy) & X(!al_healthy)  & X(ar_healthy) & !c2) -
> X(c3) )

[]((X(!gr_healthy) & X(!ar_healthy)  & X(al_healthy) & !c3) -
> X(c2) )

[]((!gl_healthy & !al_healthy & !ar_healthy) -> X(c6) )

[]((!gr_healthy & !ar_healthy & !al_healthy) -> X(c5) )
```

Declarative specification of controllers

Example: LTL specification for EPS

Close mapping from English to LTL:

A2) At least one power source is always “healthy” (i.e. it is operational and can be inserted into the network to deliver power);

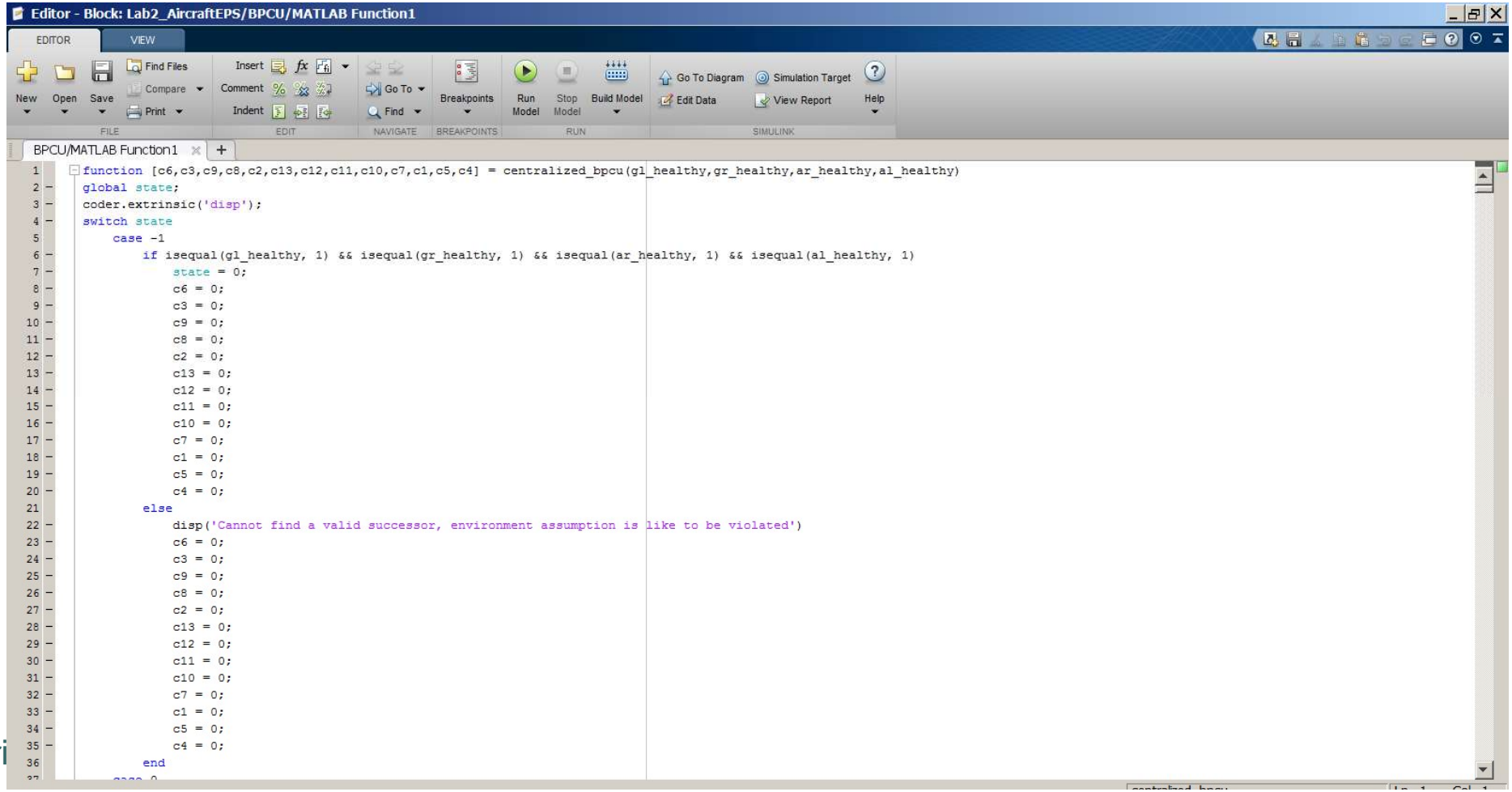

```
[](gl_healthy | gr_healthy | al_healthy | ar_healthy)
```

The controller synthesis problem

Given formula specification (e.g., in LTL) synthesize controller (e.g., FSM) which implements the specification (or state that such a controller does not exist).

Automatic controller synthesis from declarative specifications

Example: controller for EPS synthesized from previous LTL spec using Tulip (Caltech) ~3k lines of Matlab



Automatic controller synthesis from declarative specifications

Example: controller for EPS synthesized using Tulip (Caltech), ~40 states – zooming in

```
switch state
    case -1
        if isequal(gl_healthy, 1) && isequal(gr_healthy, 1) && isequal(ar_healthy, 1) && isequal(al_healthy, 1)
            state = 0;
            c6 = 0;
            c3 = 0;
            c9 = 0;
            c8 = 0;
            c2 = 0;
            c13 = 0;
            c12 = 0;
            c11 = 0;
            c10 = 0;
            c7 = 0;
            c1 = 0;
            c5 = 0;
            c4 = 0;
        else
            disp('Cannot find a valid successor, environment assumption is like to be violated')
            c6 = 0;
            c3 = 0;
            c9 = 0;
            c8 = 0;
            c2 = 0;
```

Synthesis in these two lectures

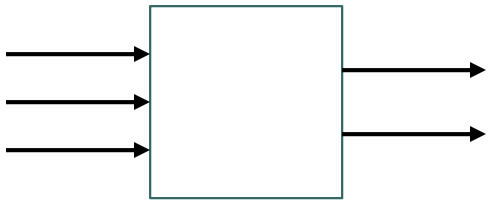
Part 1: Controller synthesis and game solving.

Part 2: Example-guided and syntax-guided synthesis.

CONTROLLER SYNTHESIS

Declarative specification of controllers

At the outset the controller is just a box with inputs and outputs:

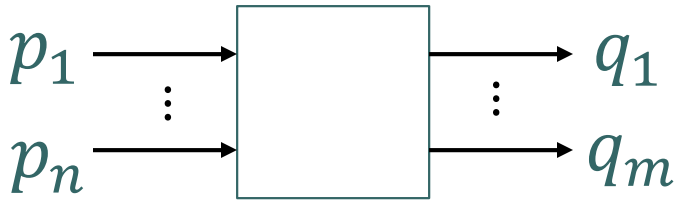


We can specify the input-output behavior of the controller in a high-level language, e.g., in **temporal logic**.

Controller synthesis (reactive synthesis)

[Pnueli-Rosner, POPL 1989]

Given interface of controller:



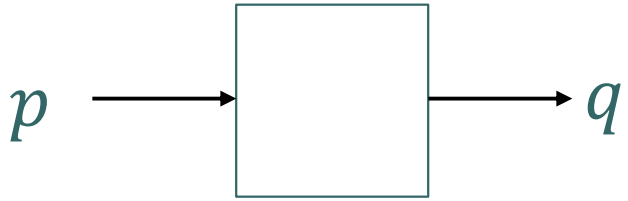
and given temporal logic formula φ over set of input/output variables,

synthesize a controller (= state machine) M, such that **all** behaviors of M (for any sequence of inputs) satisfy φ.

Note: other notions of controller synthesis exist in the literature. For details, see "Bridging the gap" paper [3], also available from instructor's web site.

Examples

Consider controller interface:



and specifications

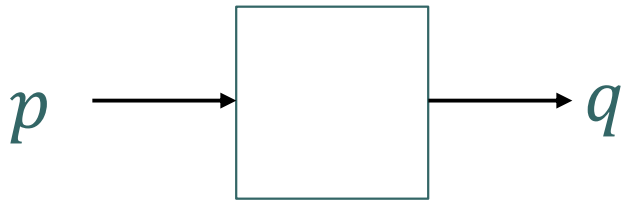
$$\varphi_1 = G(p \rightarrow Xq)$$

$$\varphi_2 = G(p \leftrightarrow Xq)$$

$$\varphi_3 = G(q \leftrightarrow Xp)$$

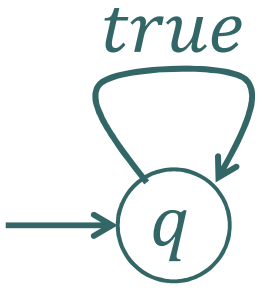
Examples

Consider controller interface:

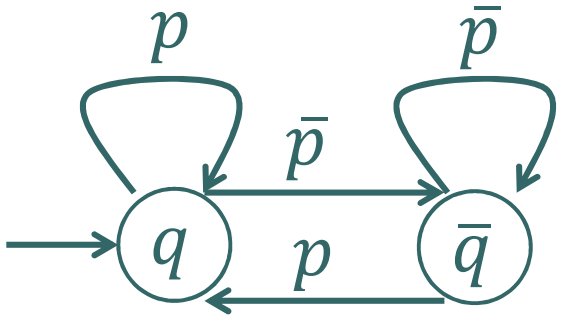


and specifications

$$\varphi_1 = G(p \rightarrow Xq)$$



$$\varphi_2 = G(p \leftrightarrow Xq)$$



$$\varphi_3 = G(q \leftrightarrow Xp)$$

No solution: controller cannot foresee the future!

Satisfiability vs. realizability

Satisfiability: exists some behavior that satisfies the specification. (In this behavior, we may choose both inputs and outputs as we wish.)

Realizability: exists controller that implements the specification. Must work for all input sequences, since inputs are uncontrollable.

Inherently different problems, also w.r.t. complexity:

- LTL satisfiability: PSPACE
- LTL realizability: 2EXPTIME

Controller synthesis algorithms: computing strategies in games

Solving safety games

Solving reachability games

Solving deterministic Büchi games (liveness)

Remarks on the general LTL synthesis problem

Controller synthesis algorithms

Solving safety games

Solving reachability games

Solving deterministic Büchi games (liveness)

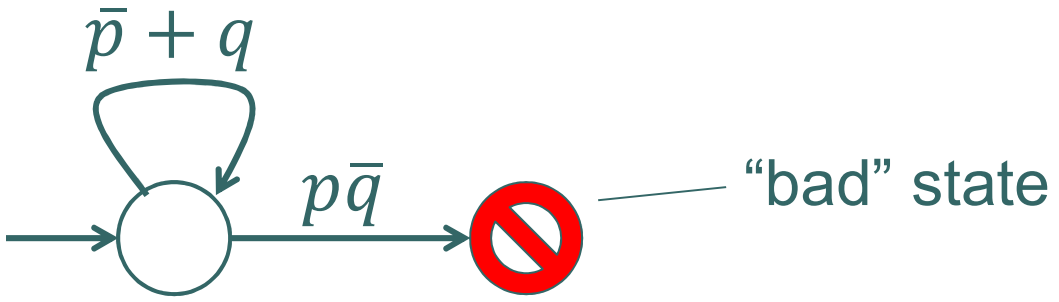
Remarks on the general LTL synthesis problem

Safety automata

In some fortunate cases, the LTL specification can be translated to a safety automaton.

Example: $\varphi = G(p \rightarrow q)$

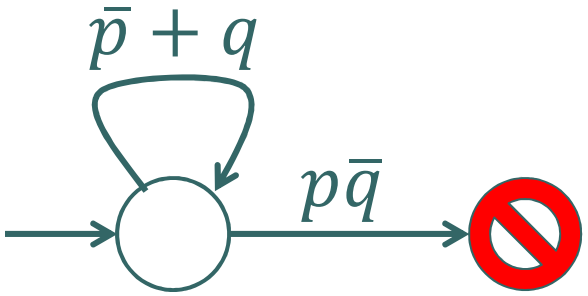
Automaton:



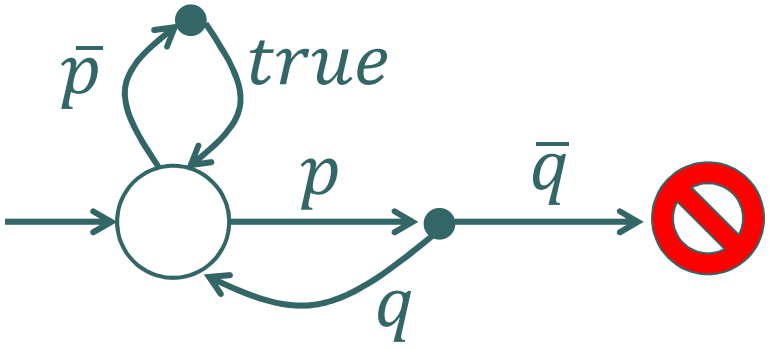
“Spreading” a safety automaton to a game
[Ehlers PhD thesis, 2013]

We need to separate the input moves from the output moves:

Automaton:



Game:

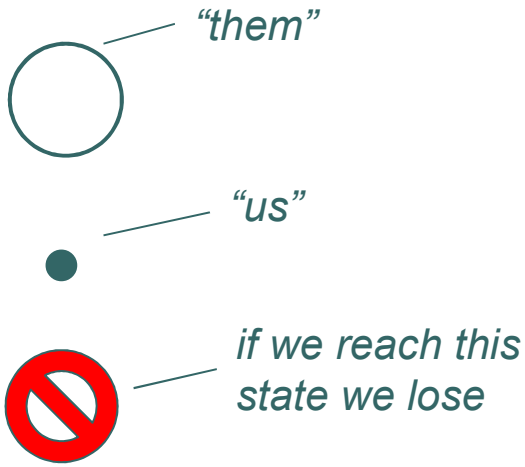


Safety games

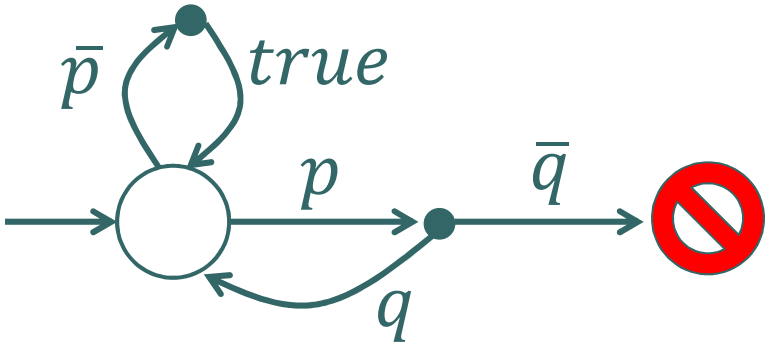
Input (environment) states:

Output (controller) states:

Bad state:

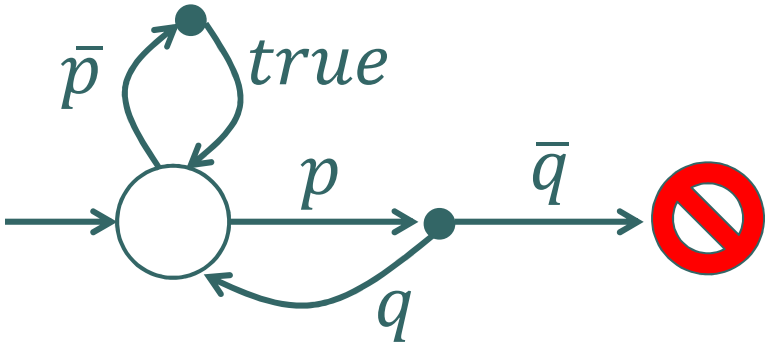


Goal: find **winning strategy** = avoiding bad state



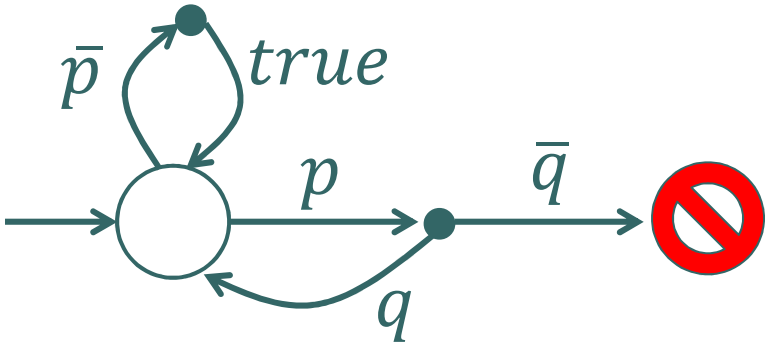
Solving safety games

1. Compute set of losing states, starting with *Losing* := {⛔};
2. If initial state in *Losing*, no strategy exists.
3. Otherwise, all remaining states are winning. Extract strategy from them by choosing outputs that avoid the losing states.



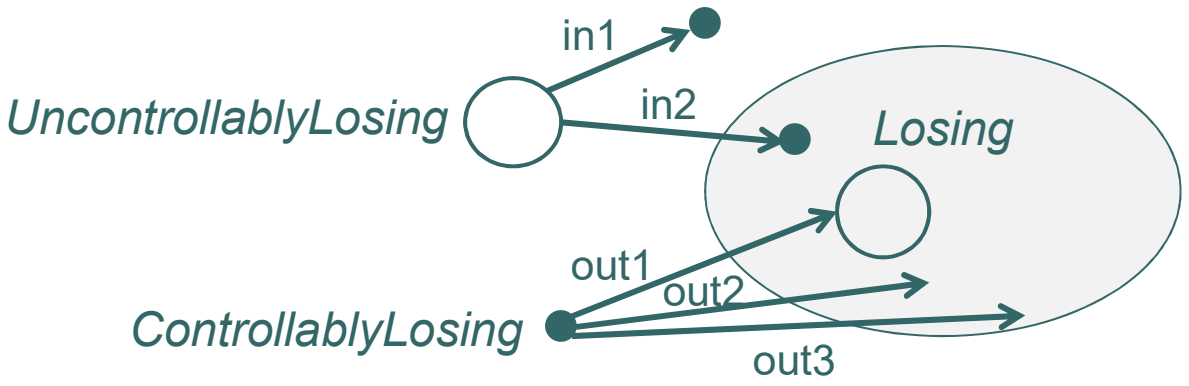
Solving safety games

1. Compute set of losing states, starting with *Losing* := {⛔};
   - repeat
     - *UncontrollablyLosing* := { s | s has uncontrollable succ in *Losing* };
     - *ControllablyLosing* := { s | all controllable succs of s are in *Losing*};
     - *Losing* := *Losing* U *UncontrollablyLosing* U *ControllablyLosing* ;
   - until *Losing* does not change;



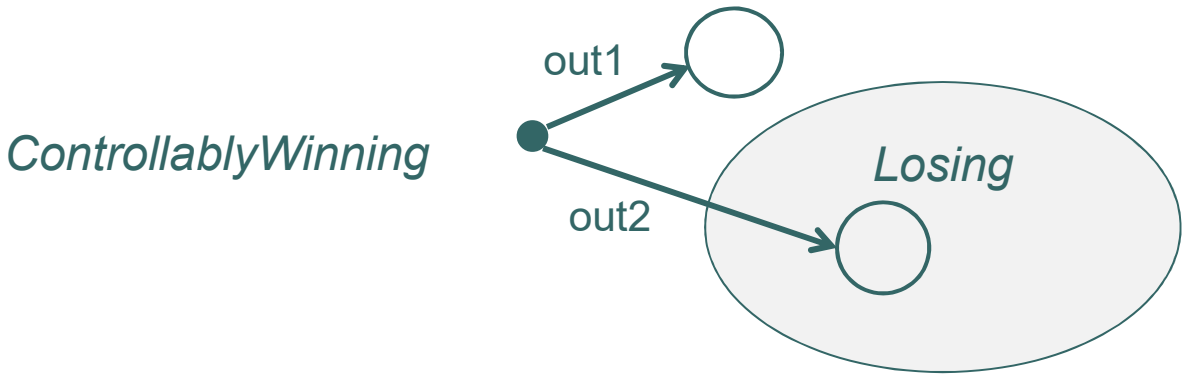
Solving safety games

1. Compute set of losing states, starting with *Losing* := {⛔};
   - repeat
     - *UncontrollablyLosing* := { s | s has uncontrollable succ in *Losing* };
     - *ControllablyLosing* := { s | all controllable succs of s are in *Losing*};
     - *Losing* := *Losing* U *UncontrollablyLosing* U *ControllablyLosing* ;
   - until *Losing* does not change;



Solving safety games

- Extracting the strategy: “cut” controllable transitions in order to avoid losing states.
- Strategy is **state-based** (also called “positional”, or “memoryless”).



Controller synthesis algorithms

Solving safety games

Solving reachability games

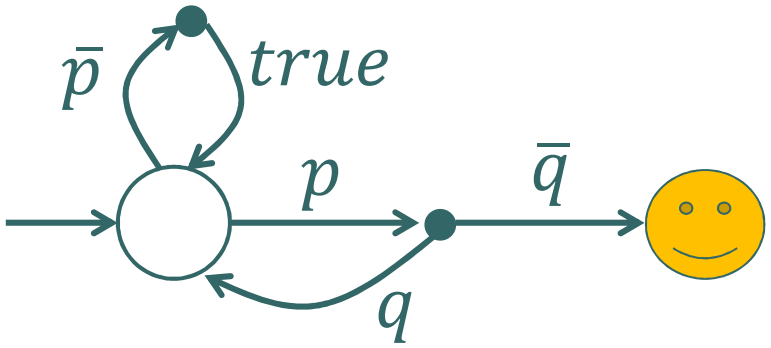
Solving deterministic Büchi games (liveness)

Remarks on the general LTL synthesis problem

Reachability games: dual of safety games

Reachability game: trying to reach a target state.

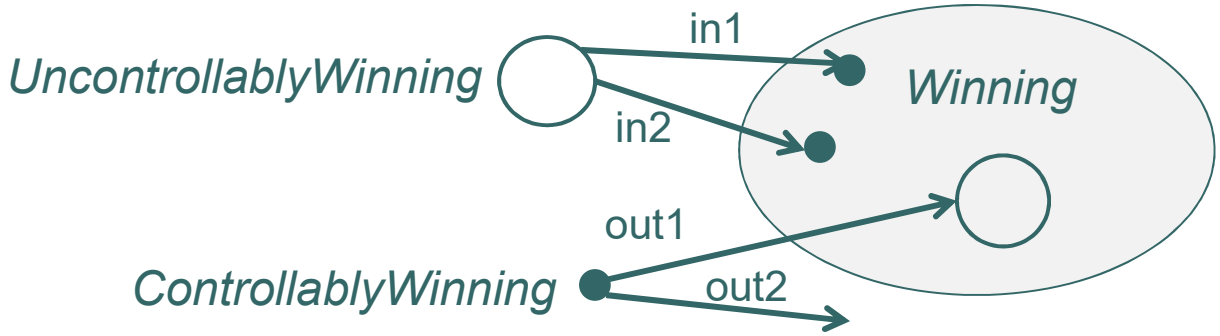
Observation: what is *Losing* for the safety player is *Winning* for the reachability player (and vice versa).



Solving reachability games: direct algorithm

1. Compute set of *Winning* states;
   - *Winning* := { };
   - repeat
     - *Winning* := *Winning* U **ForceNext**(*Winning*);
   - until *Winning* does not change;

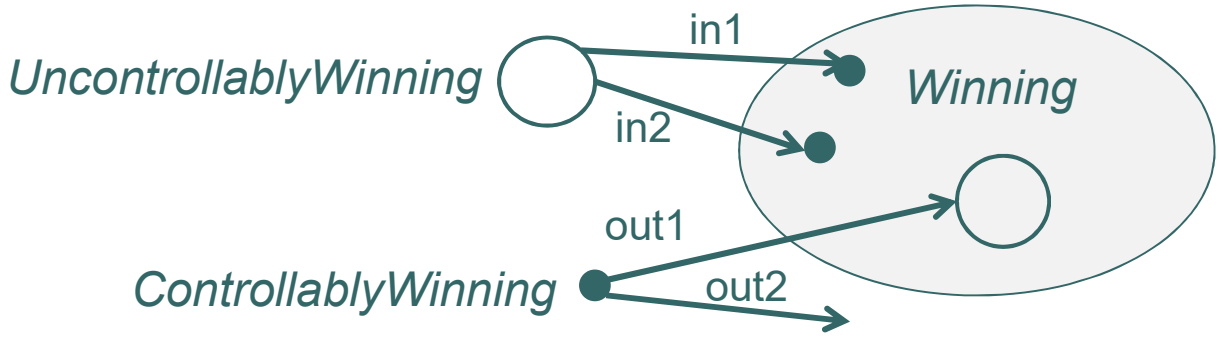
     - **ForceNext**(S) := { s | all uncontrollable succs of s are in S }
       U { s | s has controllable succ in S }



How to extract strategies in reachability games?

Similarly as for safety games:

Extract strategy from **ForceNext**(S): ensure you choose the right controllable transition that leads in winning state.

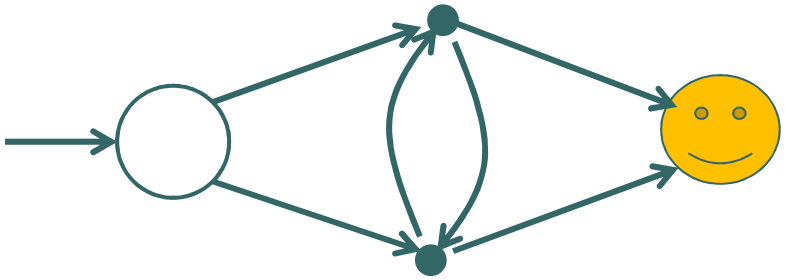


Is strategy state-based?

Yes!

How to extract strategies in reachability games?

Similarly as for safety games: BUT, a subtlety:



Need to fix successor the first time state is added in *Winning*.

Controller synthesis algorithms

Solving safety games

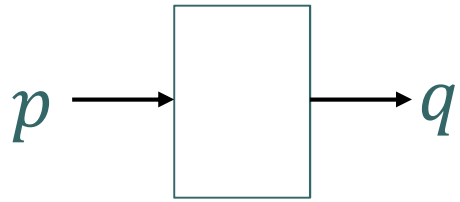
Solving reachability games

Beyond safety and reachability games

Remarks on the general LTL synthesis problem

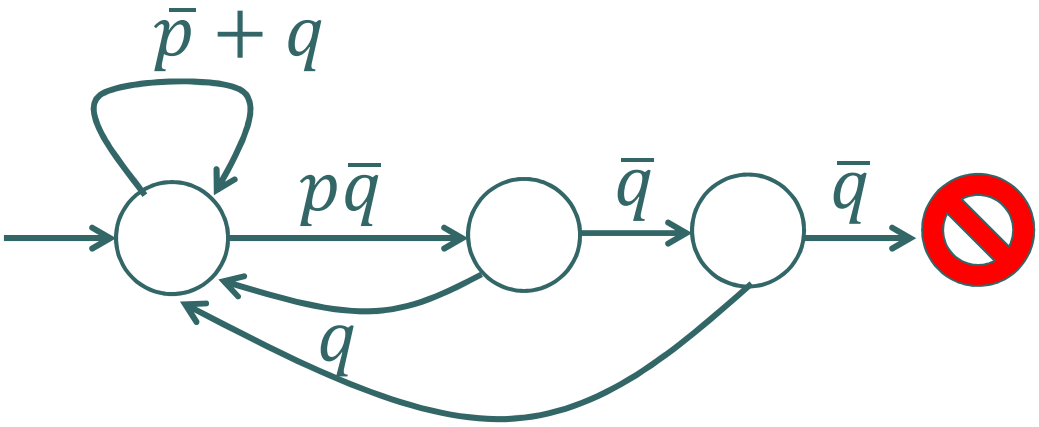
What about other types of properties?

Bounded response specifications can be translated to safety automata/games:



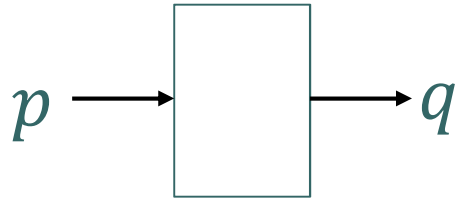
$$\varphi = G(p \rightarrow (q \; or \; Xq \; or \; XXq))$$

Automaton:



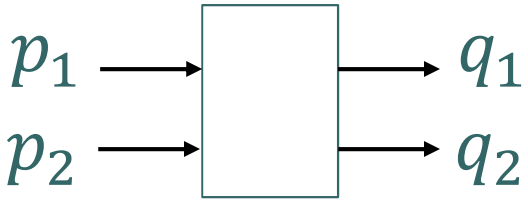
What about liveness properties?

What about **unbounded response**?



$$\varphi = G(p \rightarrow Fq)$$

More interesting example:



$$\varphi = G(p_1 \rightarrow Fq_1)\ \&\ G(p_2 \rightarrow Fq_2) \\ \&\ G(\neg(q_1\ \&\ q_2))$$

Synthesis for general LTL specifications

Given LTL specification φ:

If φ can be translated to a deterministic Büchi automaton, then can extend the previous ideas to solving Büchi games.

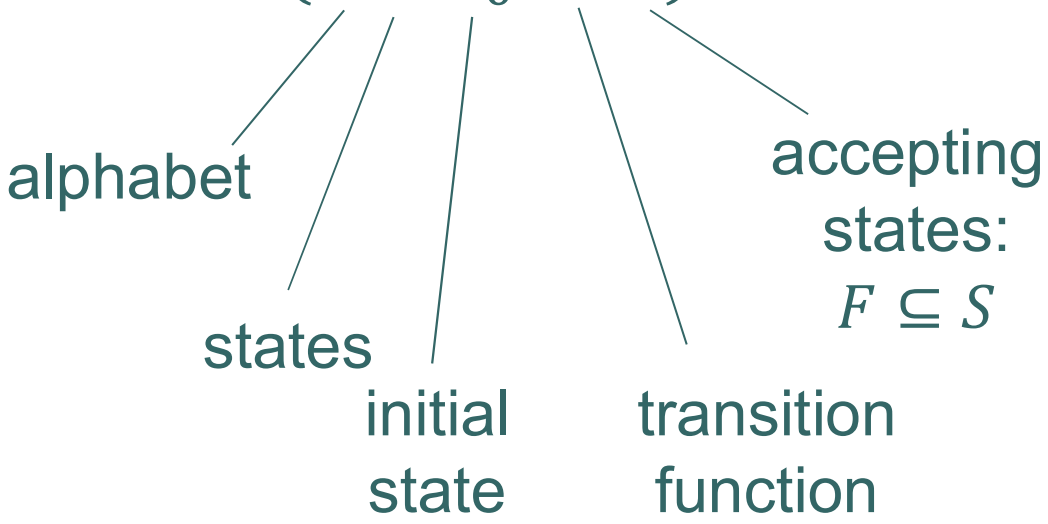
Otherwise, solution involves more advanced topics, such as tree automata. Will not be covered in this course.

Note: LTL **cannot** always be translated to deterministic Büchi automata.

Büchi automata

Syntactically same as finite state automata:

$$A = (\Sigma, S, s_0, \delta, F)$$



But Büchi automata accept **infinite words**.

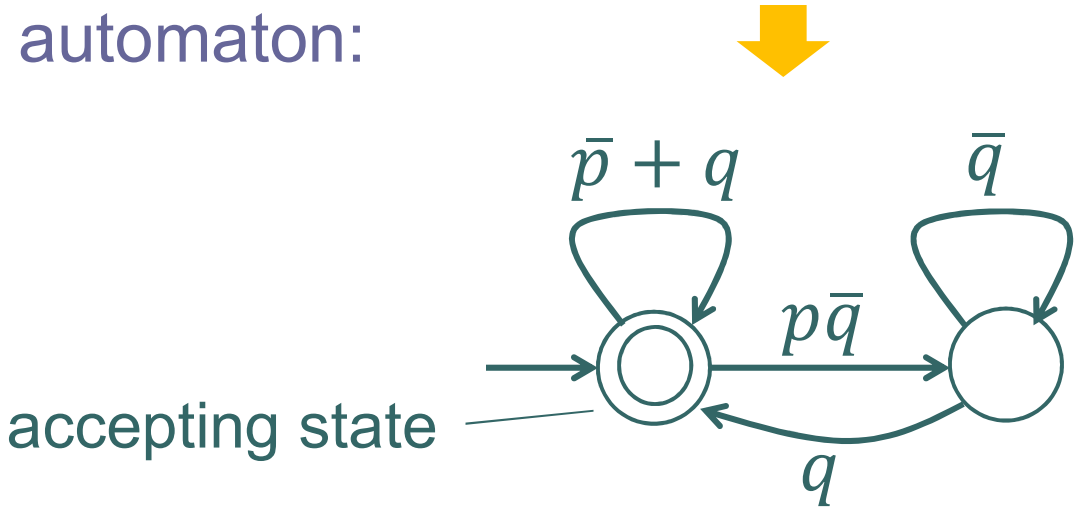
A run must visit an accepting state infinitely often.

From LTL to Büchi automata

Consider unbounded response property:

$$\varphi = G(p \rightarrow Fq)$$

Büchi automaton:



Controller synthesis algorithms

Solving safety games

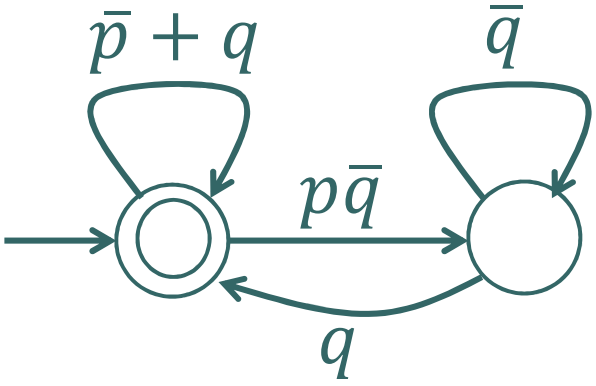
Solving reachability games

Solving deterministic Büchi games (liveness)

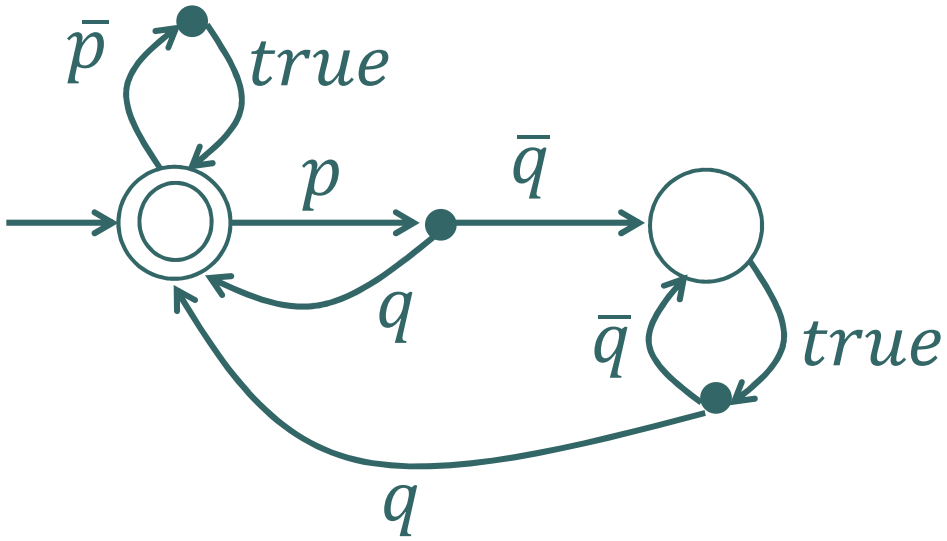
Remarks on the general LTL synthesis problem

Spreading Büchi automata to Büchi games

Büchi automaton:

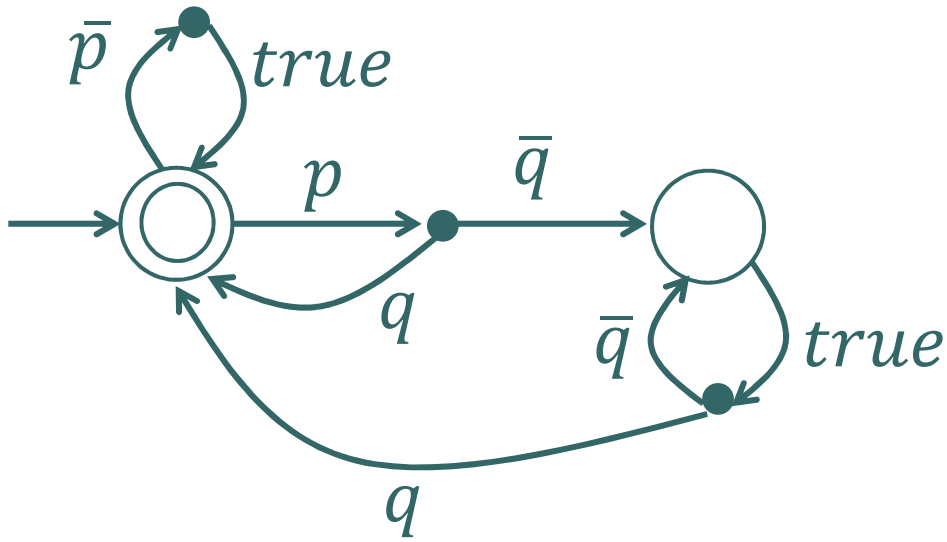


Büchi game:



Solving deterministic Büchi games

1. Compute set of ***RecurrentAccepting*** states = accepting states from which controller can force returning to an accepting state infinitely often.
2. Solve reachability game with target = RecurrentAccepting.

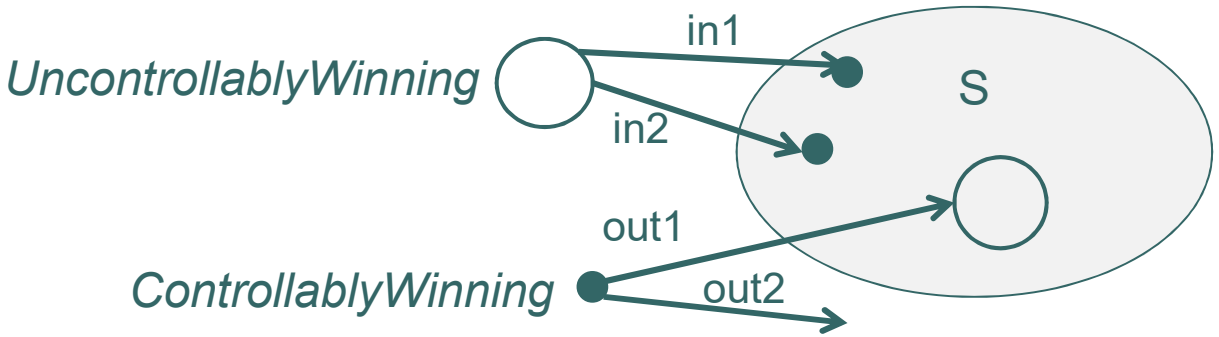


Solving deterministic Büchi games

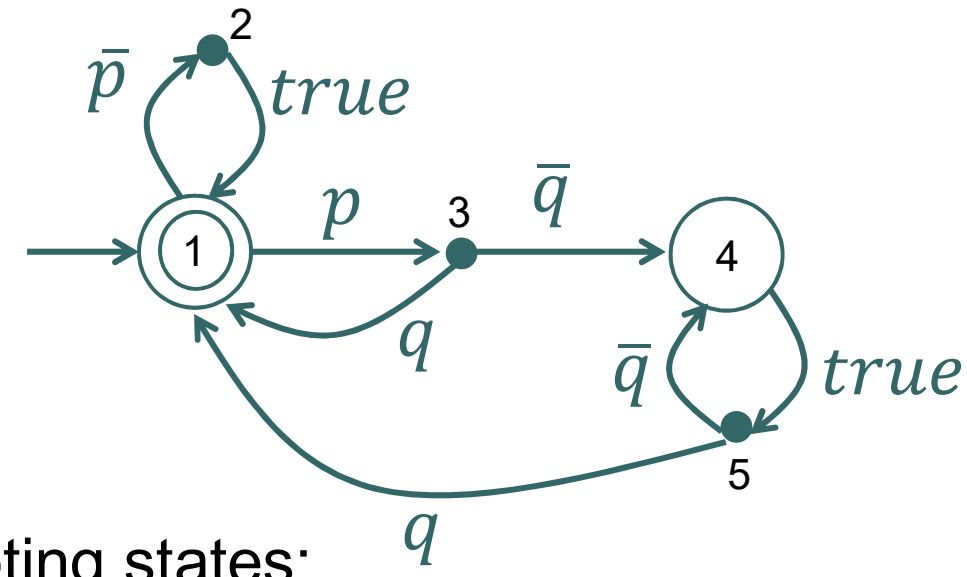
1. Compute set of ***RecurrentAccepting*** states = accepting states from which controller can force returning to an accepting state infinitely often.
   - *RecAcc* := set of all accepting states;
   - repeat
     - *Revisit* := { };
     - repeat
       Revisit := *Revisit* ∪ **ForceNext**(*Revisit* ∪ *RecAcc*);
     - until *Revisit* does not change;
     - *RecAcc* := *RecAcc* ∩ *Revisit*;
   - until set *RecAcc* does not change;

Recall

ForceNext(S) := { s | all uncontrollable succs of s are in S }
U { s | s has controllable succ in S }



Solving deterministic Büchi games – Example



- *RecAcc* := set of all accepting states;
- repeat
  - *Revisit* := { };
  - repeat
    Revisit := *Revisit* U **ForceNext**(*Revisit* U *RecAcc*);
  - until *Revisit* does not change;
  - *RecAcc* := *RecAcc* ∩ *Revisit*;
- until set *RecAcc* does not change;

Computing recurrent accepting states: a subtle relation with reachability games

1. Compute set of ***RecurrentAccepting*** states = accepting states from which controller can force returning to an accepting state infinitely often.
   - *RecAcc* := set of all accepting states;
   - repeat
     - *Revisit* := { };
     - repeat
       Revisit := *Revisit* ∪ **ForceNext**(*Revisit* ∪ *RecAcc*);
     - until *Revisit* does not change;
     - *RecAcc* := *RecAcc* ∩ *Revisit*;
   - until set *RecAcc* does not change;

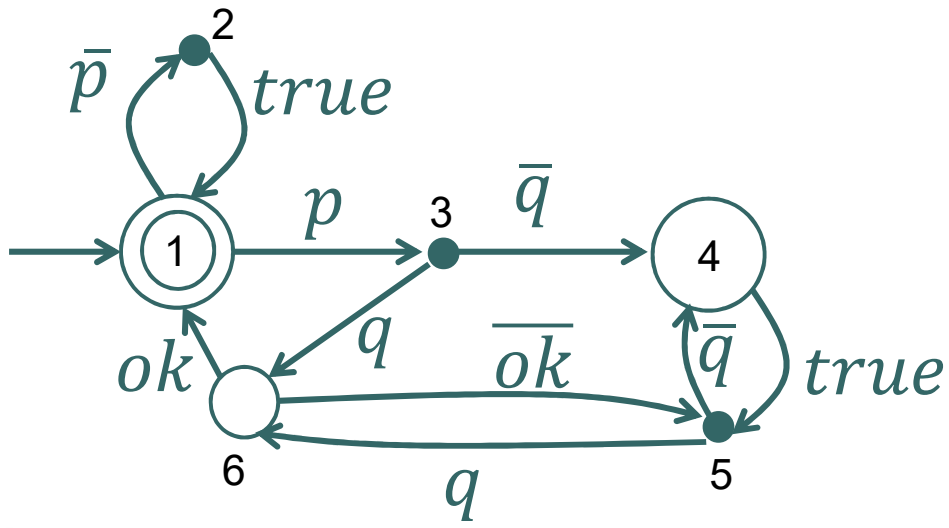
(almost) a reachability game iteration

Solving reachability games vs. computing Revisit

1. Compute set of *Winning* states:
   - *Winning* := {  };
   - repeat
     - *Winning* := *Winning* U **ForceNext**(*Winning*);
   - until *Winning* does not change;
2. Compute *Revisit*:
   - *Revisit* := { };
   - repeat
     Revisit := *Revisit* U **ForceNext**(*Revisit* U *RecAcc*);
   - until *Revisit* does not change;

What is the difference?
Does it matter?

Solving deterministic Büchi games – modified example

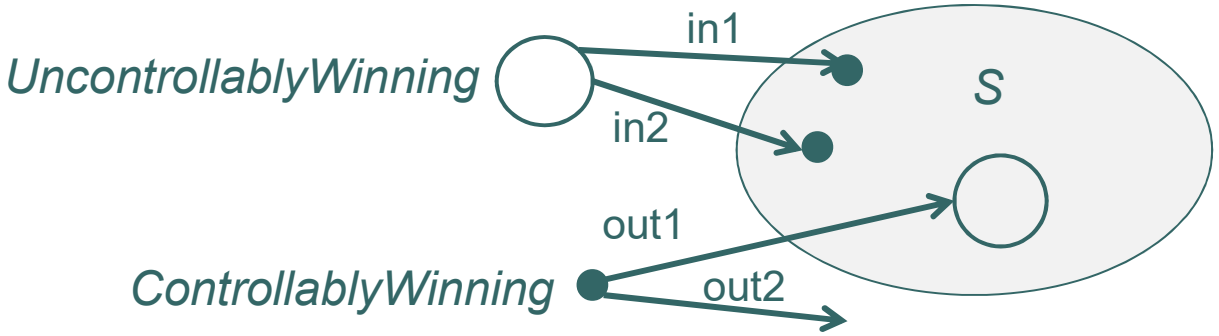


- *RecAcc* := set of all accepting states;
- repeat
  - *Revisit* := { };
  - repeat
    Revisit := *Revisit* U **ForceNext**(*Revisit* U *RecAcc*);
  - until *Revisit* does not change;
  - *RecAcc* := *RecAcc* ∩ *Revisit*;
- until set *RecAcc* does not change;

How to extract strategies in deterministic Büchi games?

Similarly as for reachability games:

Extract strategy from **ForceNext**(S): ensure you choose the right controllable transition that leads in winning state.



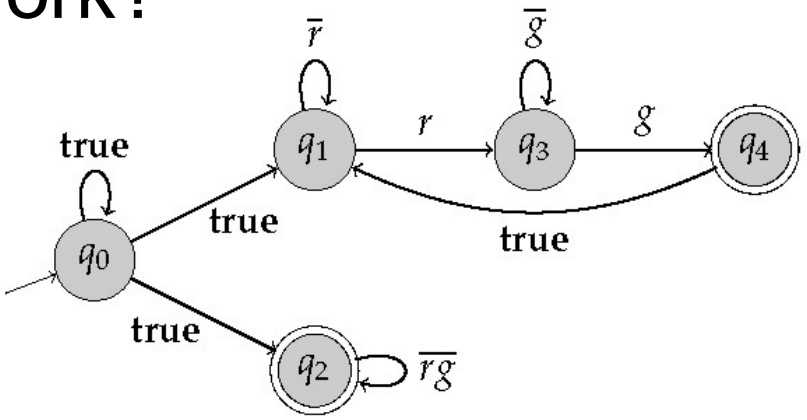
Careful to choose the transition the first time state is added to S.

Is strategy state-based?

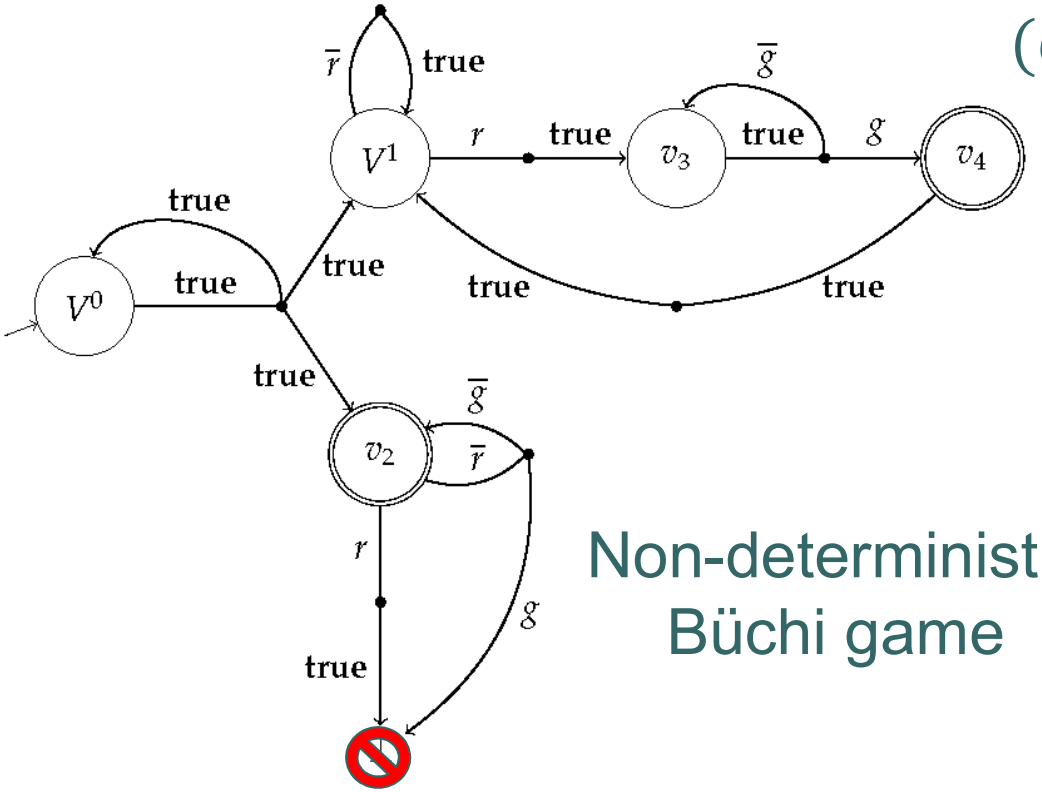
Yes!

What about non-deterministic Büchi games? Does same algorithm work?

Not quite:
algorithm sound
but incomplete.



Non-deterministic automaton for
$(GFr \wedge GFg) \vee (FG\neg r \wedge FG\neg g)$



Non-deterministic
Büchi game

r : input
g : output

[Ruediger Ehlers,
PhD thesis, 2013]

Controller synthesis algorithms

Solving safety games

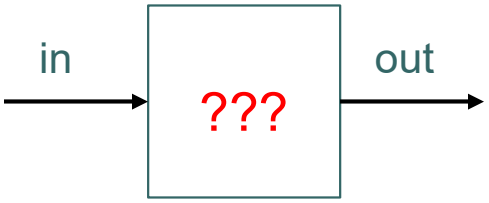
Solving reachability games

Solving deterministic Büchi games (liveness)

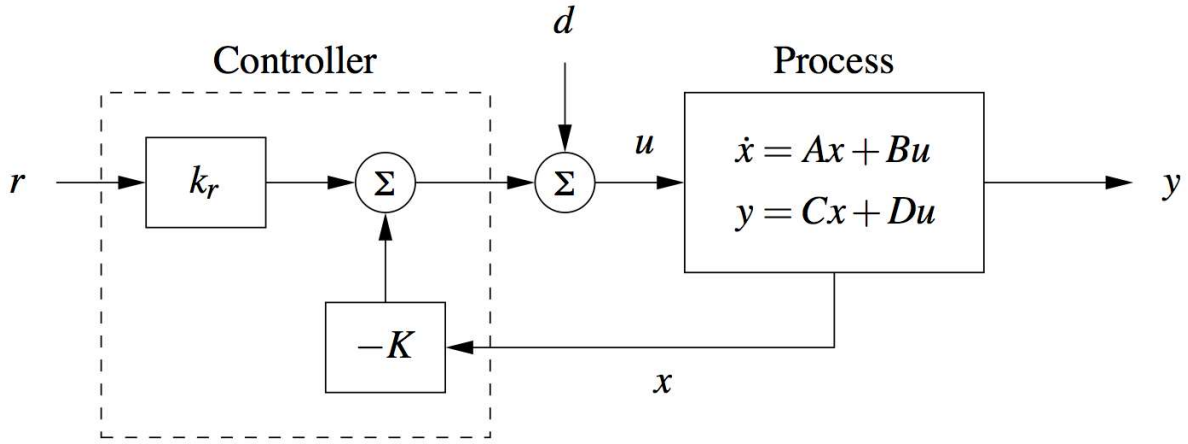
Remarks on the general LTL synthesis problem

Controller synthesis: EE vs. CS ?

CS: synthesize outputs to implement φ:



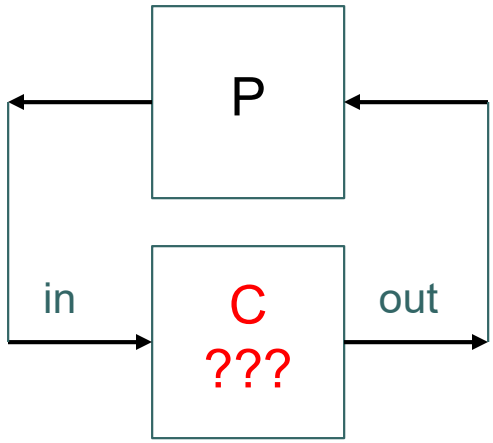
EE: synthesize inputs to stabilize a physical process/plant:



Not different: plant inputs = controller outputs (and vice versa).

Can we capture plants in the CS synthesis problem?

CS: given plant P (say, a FSM), synthesize controller C, so that closed-loop system satisfies φ:



Can we reduce this problem to the standard LTL synthesis problem?

Remarks, assessment

Despite some (mostly isolated) success stories, controller synthesis hasn't really caught on yet in practice.

Why is that?

- Normal: things like that take time (c.f. model-checking)
- 2EXPTIME is a horrible (worst-case) complexity (remember: even linear is too expensive because of state explosion!)
- Tools still impractical
- Synthesis of real, complex systems from complete specs impractical (imagine full synthesis of complete Intel microchip from LTL specs …)
- Lack of good debugging (e.g., counter-examples)
- Need: better tools, better methods (incremental, interactive, …)
- Great opportunities for research!

Automatic synthesis of distributed protocols

Joint work with Rajeev Alur, Christos Stergiou et al (UPenn)

Sponsors: NSF Expeditions ExCAPE

Motivation: distributed protocols

- Notoriously hard to get right

Can we **synthesize** such protocols **automatically**?

COMMUNICATIONS OF THE ACM

HOME | CURRENT ISSUE | NEWS | BLOGS | OPINION | RESEARCH | PRACTICE | CAREERS | ARCHIVE | VIDEOS

Home / Magazine Archive / April 2015 (Vol. 58, No. 4) / How Amazon Web Services Uses Formal Methods / Full Text

CONTRIBUTED ARTICLES

How Amazon Web Services Uses Formal Methods

(to model and verify distributed protocols)

By Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, Michael Deardeuff
Communications of the ACM, Vol. 58 No. 4, Pages 66-73
10.1145/2699417
Comments (1)

VIEW AS: SHARE:

Since 2011, engineers at Amazon Web Services (AWS) have use formal specification and model checking to help solve difficult design problems in critical systems. Here, we describe our motivation and experience, what has worked well in our proble domain, and what has not. When discussing personal experien we refer to the authors by their initials.

At AWS we strive to build services that are simple for customer to use. External simplicity is built on a hidden substrate of

Key Insights

- **Formal methods find bugs in system designs that cannot be found through any other technique we know of.**
- **Formal methods are surprisingly feasible for mainstream software development and give good return on investment.**
- **At Amazon, formal methods are routinely applied to the design of complex real-world software, including public cloud services.**

Verification and synthesis in a nutshell

- Verification:
  1. Design system "by hand": S
  2. State system requirements: ϕ
  3. Check: does S satisfy ϕ ?
- Synthesis (ideally):
  1. State system requirements: ϕ
  2. Generate automatically system S that satisfies ϕ by construction.

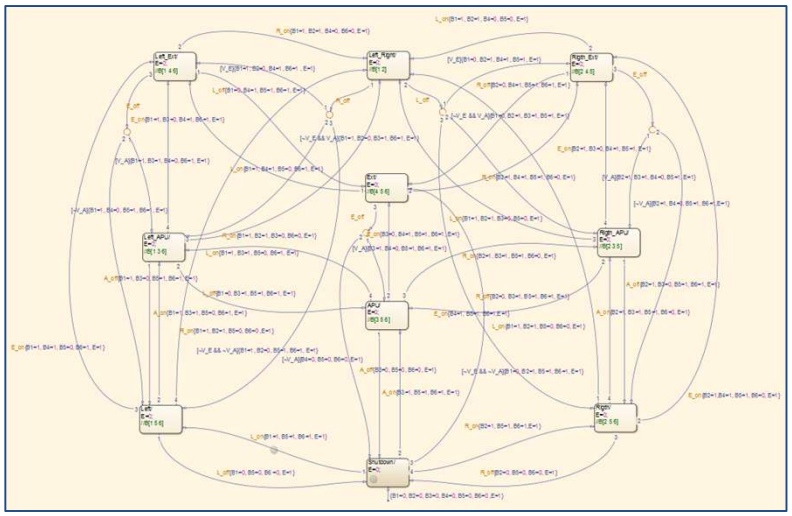
State of the art synthesis

- From formal specs to discrete controllers:

```
#Assumptions
(gl_healthy & gr_healthy & al_healthy & ar_healthy)
[](gl_healthy | gr_healthy | al_healthy | ar_healthy)
[](!gl_healthy -> X(!gl_healthy) )
[](!gr_healthy -> X(!gr_healthy) )
[](!al_healthy -> X(!al_healthy) )
[](!ar_healthy -> X(!ar_healthy) )

#Guarantees
(!c1 & !c2 & !c3 & !c4 & !c5 & !c6 & !c7 & !c8 & !c9 & !c10 &
!c11 & !c12 & !c13)
[](X(c7) & X(c8) & X(c11) & X(c12) & X(c13))
[](!(c2 & c3))
[](!(c1 & c5 & (al_healthy | ar_healthy)))
[](!(c4 & c6 & (al_healthy | ar_healthy)))
[]((X(gl_healthy) & X(gr_healthy) ) -> X(!c2) & X(!c3) &
X(!c9) & X(!c10))
[]((X(!gl_healthy) & X(!gr_healthy) ) -> X(c9) & X(c10))
…
```

Specification (temporal logic formulas)

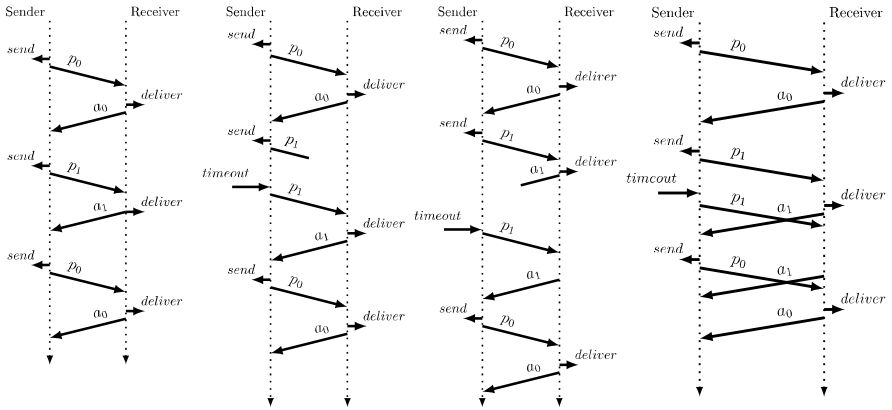


Controller (state machine)

- Limitations:
  - Scalability (writing full specs & synthesizing from them)
  - Not applicable to distributed protocols (undecidable)

Synthesis of Distributed Protocols from Scenarios and Requirements

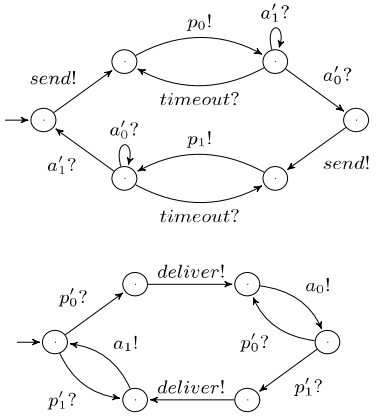
- Idea: **combine requirements** + example **scenarios**



example scenarios

Synthesis tool

formal requirements (safety, liveness, deadlock-freedom, …)

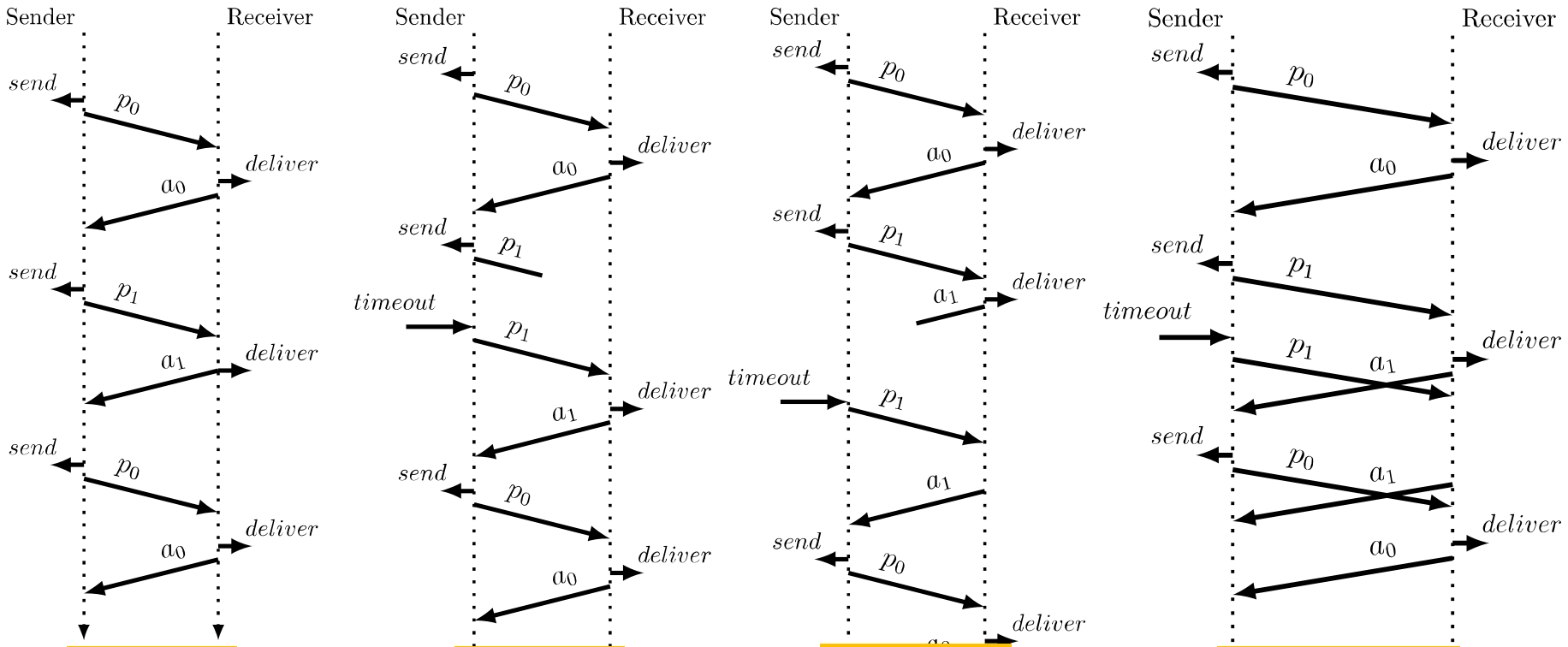


synthesized protocol (state machines)

These are typically not complete specs!

Scenarios: message sequence charts

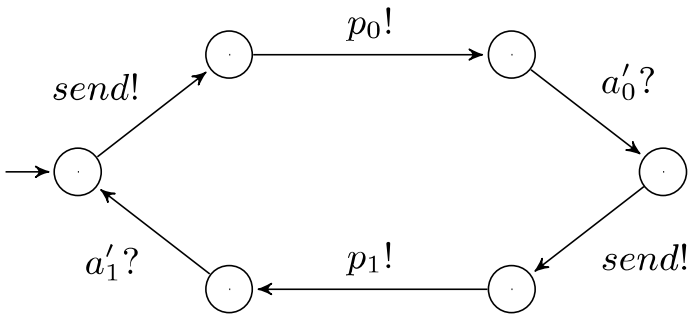
- Describe what the protocol must do in **some** cases
- Intuitive language ⇒ good for the designer
- Only a few scenarios required (1-10)

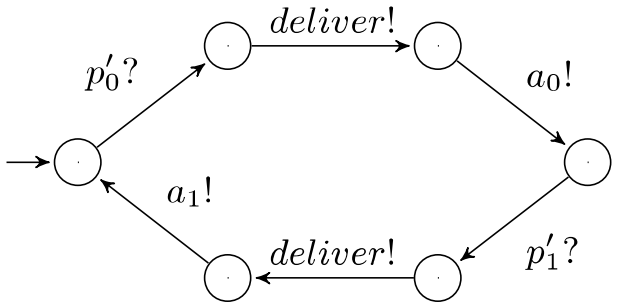


Synthesis becomes a completion problem

Incomplete automata learned from first scenario:

ABP Sender

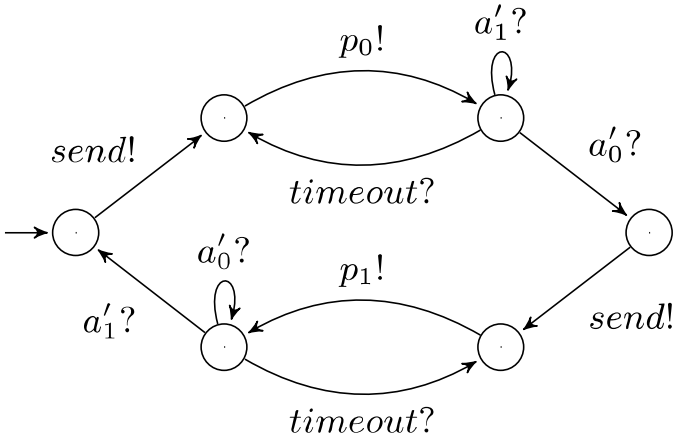


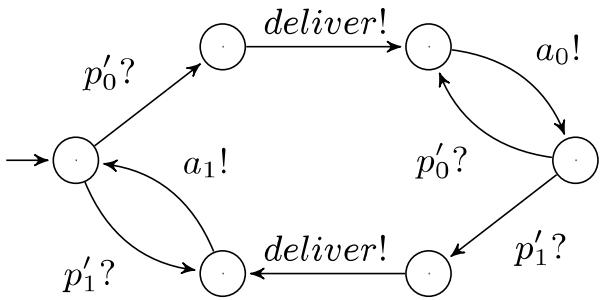


ABP Receiver

Automatically completed automata:

ABP Sender

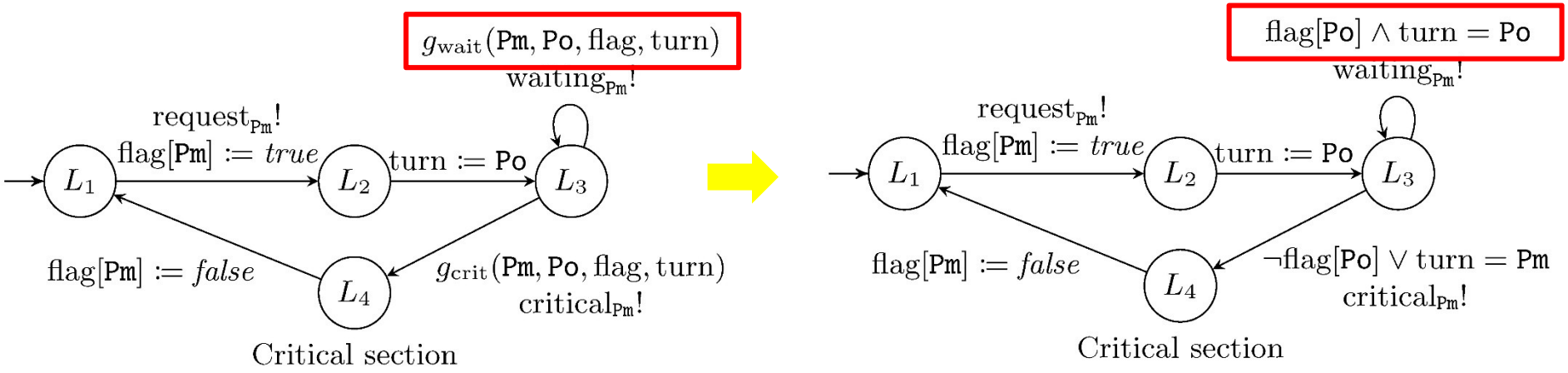




ABP Receiver

Results

- Able to synthesize the distributed Alternating Bit Protocol (ABP) and other simple finite-state protocols (cache coherence, consensus, …) fully automatically [HVC'14, ACM SIGACT'17].
- Towards industrial-level protocols described as **extended state machines** [CAV'15].



Algorithmic technique: counter-example guided completion of (extended) state machines

- Completion of incomplete machines: find missing transitions, guards, assignments, etc.

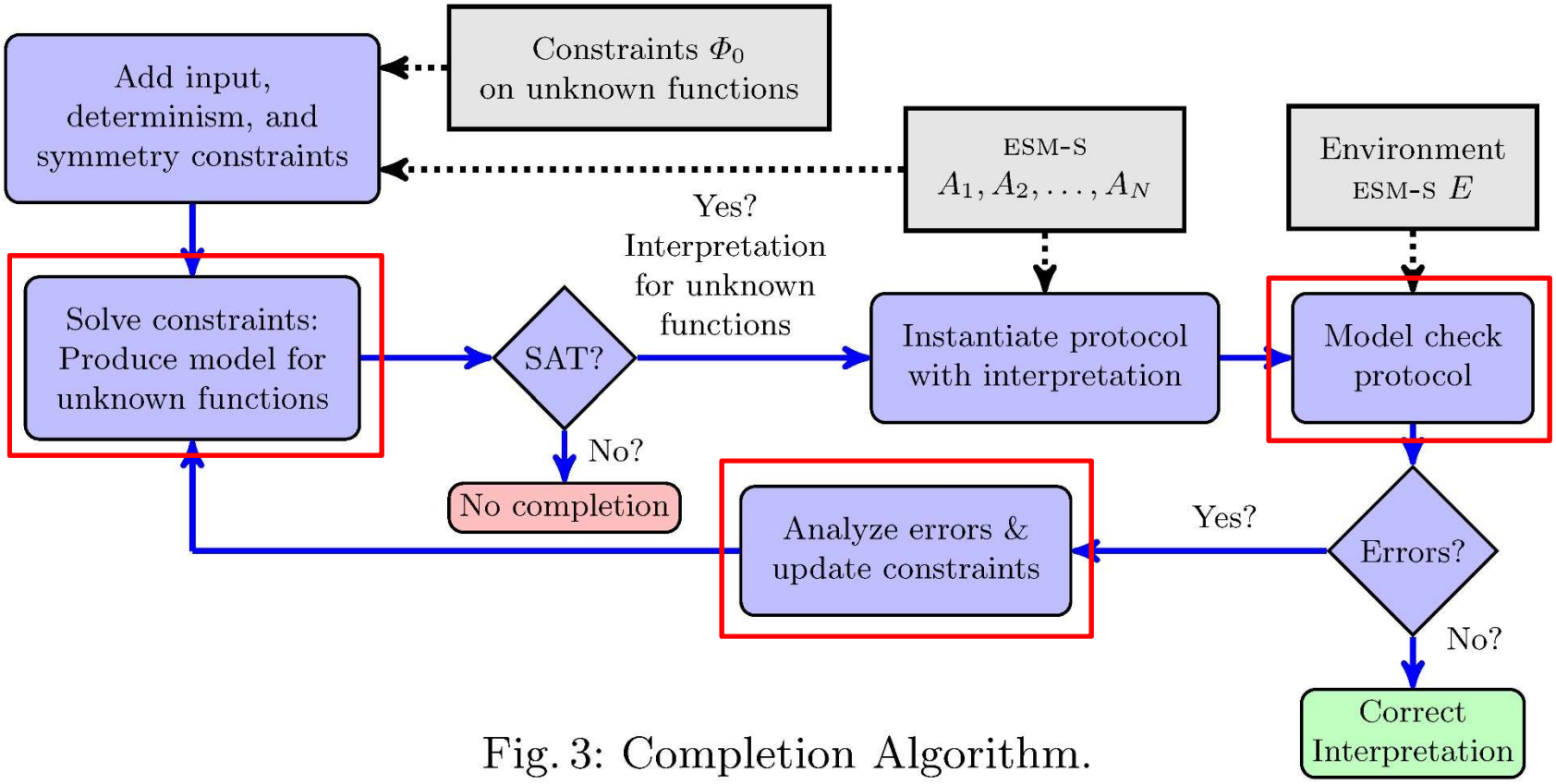
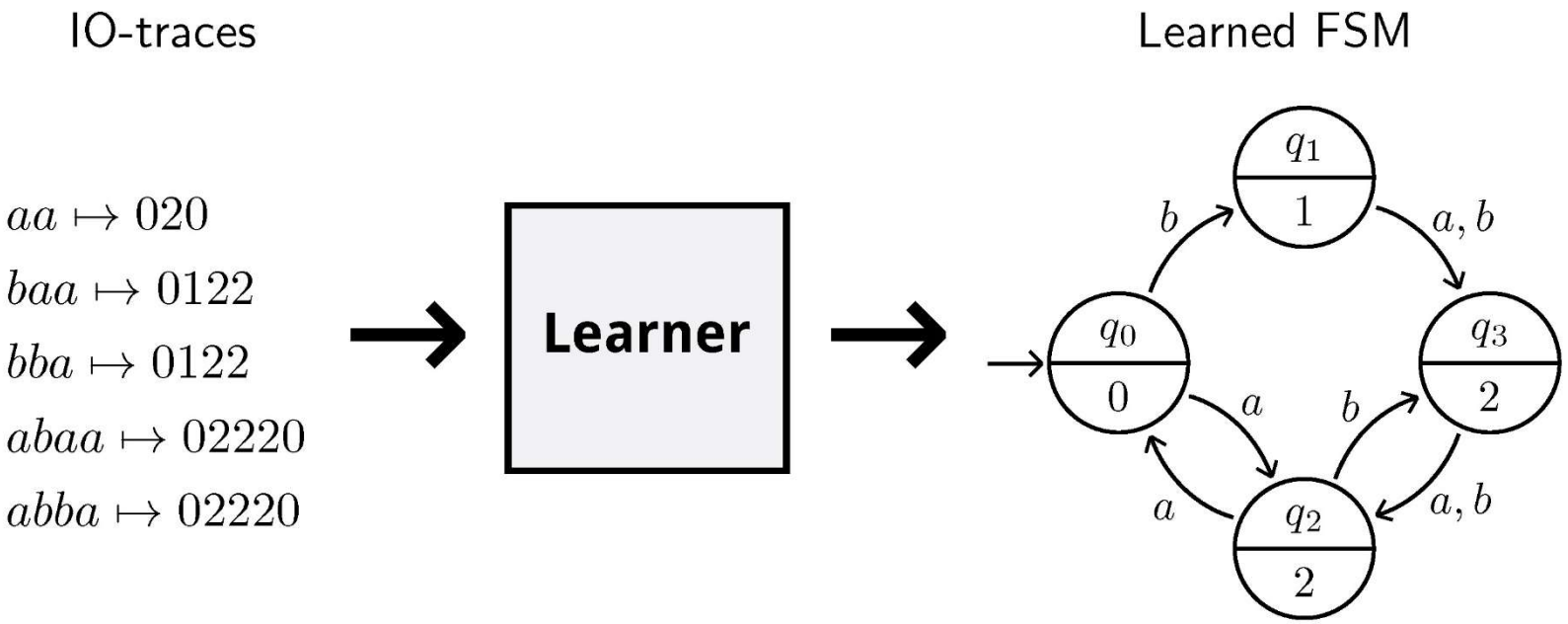


Fig. 3: Completion Algorithm.

Other recent work: learning Moore machines from input-output traces

[Giantamidis, Tripakis, 2016]

- Model learning



Fundamental question: what is the right way to generalize?

Combining synthesis with learning

- Synthesis: given specification ϕ, find system S, such that $S \vDash \phi$
- Learning: given set of examples E, find system S, such that S is consistent with E and "generalizes well" …
- Synthesis from spec + examples: given set of examples E and specification ϕ, find system S, such that S is consistent with E and $S \vDash \phi$
  - Key advantage: ϕ guides the generalization!

References

1. Pnueli, A., Rosner R., *On the Synthesis of a Reactive Module*, POPL 1989.
2. Ehlers, R., *Symmetric and Efficient Synthesis*, PhD thesis, 2013.
3. Jobstmann, B., *Reachability and Buchi Games*, slides available online, 2010.
4. Ehlers, Lafortune, Tripakis, Vardi, *Supervisory Control and Reactive Synthesis: a Comparative Introduction*, DEDS 2017.
5. Bloem, Chatterjee, Jobsmann, *Graph Games and Reactive Synthesis*, in Handbook of Model Checking, 2019
6. Alur, Tripakis, Automatic Synthesis of Distributed Protocols, ACM SIGACT News on Distributed Computing, 2017
7. Kang, Lafortune, Tripakis, Automated Synthesis of Secure Platform Mappings, CAV 2019