Supervisory Control: Advanced Theory and Applications

Dr Rong Su
S1-B1b-59, School of EEE
Nanyang Technological University
Tel: +65 6790-6042, Email: rsu@ntu.edu.sg

Introduction to Supervisory Control Theory

Outline

- Introduction to Supervisory Control
- Ramadge-Wonham Supervisory Control Theory
- Example – A Pusher-Lift System
- Primary Goals of EE6226

The Concept of Discrete Event Systems (DES)

- A DES is a structure with ‘states’ having duration in time, ‘events’ happening *instantaneously* and *asynchronously*.
  - States: e.g. machine is idle, is operating, is broken down, is under repair
  - Events: e.g. machine starts work, breaks down, completes work or repair

- State space discrete in time and space.

- State transitions ‘labeled’ by events.

The Motivation of Developing Supervisory Control Theory (SCT) for DES (till 1980)

- Control problems ***implicit*** in the literature (enforcement of resource constraints, synchronization, ...)

But

- Emphasis on modeling, simulation, verification
- Little formalization of control synthesis
- Absence of control-theoretic ideas
- No standard model or approach to control

Related Areas

- Programming languages for modeling & simulation
- Queues, Markov chains
- Petri nets
- Boolean models
- Formal languages
- Process algebras (CSP, CCS)

“Great” Expectations for SCT

- System model
  - Discrete in time and (usually) space
  - Asynchronous (event-driven)
  - Nondeterministic
    - support transitional choices
- Amenable to formal control synthesis
  - exploit control concepts
- Applicable: manufacturing, traffic, logistic,...

Relationship with Systems Control Concepts

- State space framework well-established:
  - Controllability
  - Observability
  - Optimality (Quadratic, H_∞)

- Use of geometric constructs and partial order
  - Controllability subspaces
    - Supremal subspaces!

Ramadge-Wonham SCT (1982)

- Automaton representation
  - state descriptions for concrete modeling and computation
- Language representation
  - i/o descriptions for implementation-independent concept formulation
- Simple control "technology"

Outline

- Introduction to Supervisory Control
- Ramadge-Wonham Supervisory Control Theory
- Example – A Pusher-Lift System
- Primary Goals of EE6226

RW paradigm is based on *languages*, but implemented on *finite-state automata*

Basic Concepts of Languages

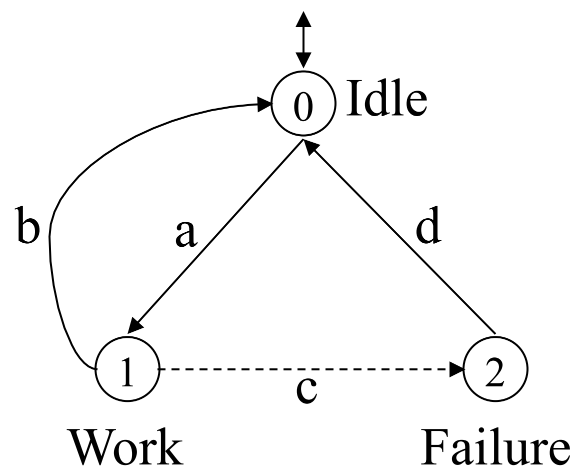
- Given an alphabet Σ (e.g. $\Sigma = \{ a, b, c, d \}$)
  - A string is a finite sequence of events from Σ, e.g. $s = ababa$
  - $\Sigma^+ := \{$ all strings generated from $\Sigma \}$, $\Sigma^* := \Sigma^+ \cup \{\varepsilon\}$
    - ε is called the *empty* string: $s\varepsilon = \varepsilon s = s$
  - Given $s_1, s_2 \in \Sigma^*$, s_1 is a *prefix* substring of s_2, if $(\exists t \in \Sigma^*)\ s_1 t = s_2$
    - We use $s_1 \leq s_2$ to denote that s_1 is a prefix substring of s_2
  - A language $W \subseteq \Sigma^*$: most time we require W to be *regular*
  - The *prefix closure* of a language W is : $\overline{W} := \{s \in \Sigma^* \mid (\exists s' \in W) s \leq s'\}$
    - W is *prefix closed* if $W = \overline{W}$

Finite-State Automaton (FSA)

- A finite-state automaton is a 5-tuple $G = (X, \Sigma, \xi, x_0, X_m)$, where
  - X : the state set
  - Σ : the alphabet
  - x_0 : the initial state
  - X_m : the marker state set (or the final state set)
  - $\xi : X \times \Sigma \to X$: the transition map
    - ξ is called a *partial* map, if it is not defined at some pair $(x,\sigma) \in X \times \Sigma$.
    - Otherwise, it is called a *total* map.
    - Extension of the transition map: $\xi : X \times \Sigma^* \to X : (x,s\sigma) \mapsto \xi(x,s\sigma) := \xi(\xi(x,s),\sigma)$

The Famous "Small Machine" Model

- $G = (X , \Sigma , \xi , x_0 , X_m)$
  - $X = \{ 0 , 1 , 2 \}$
  - $\Sigma = \{ a , b , c , d \}$
  - $x_0 = 0$
  - $X_m = \{ 0 \}$



a : starts work

b : finishes work

c : machine fails

d : machine is repaired

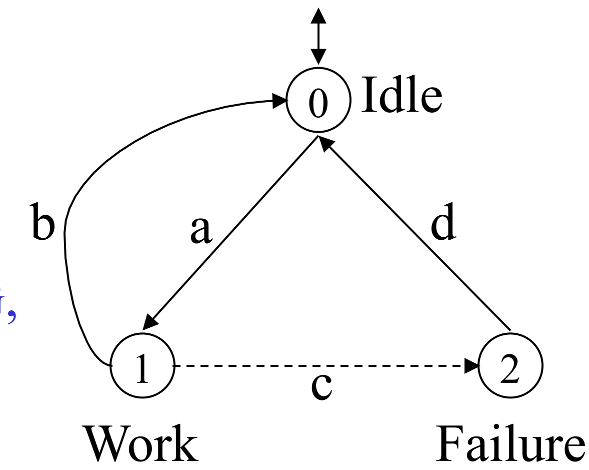
Connection between Language and FSA

- Give a FSA $G = (X, \Sigma, \xi, x_0, X_m)$,
  - *closed* behavior of G:

    $$L(G) := \{s \in \Sigma^* | \xi(x_0, s) \text{ is defined}\}$$

  - *marked* behavior of G, i.e. the language *recognized* by G,

    $$L_m(G) := \{s \in L(G) \mid \xi(x_0, s) \in X_m\}$$



- G is *nonblocking*, if $\overline{L_m(G)} = L(G)$.
- A language is *regular*, if it is recognizable by a FSA.
  - We can use Arden's rule to derive a language from a FSA.

Natural Projection over Languages

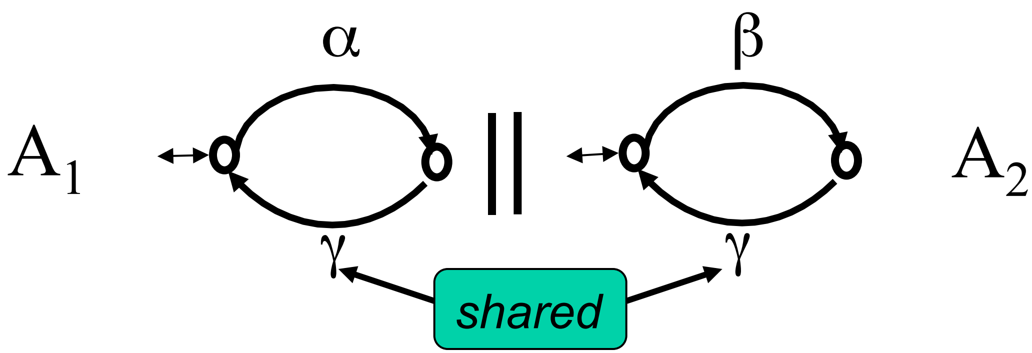
- Given Σ and $\Sigma' \subseteq \Sigma$, $P:\Sigma^* \rightarrow \Sigma'^*$ is a *natural projection* if
  - $P(\varepsilon) = \varepsilon$
  - $(\forall \sigma \in \Sigma)\ P(\sigma) = \begin{cases} \sigma & \text{if } \sigma \in \Sigma' \\ \varepsilon & \text{if } \sigma \notin \Sigma' \end{cases}$
  - $(\forall s\sigma \in \Sigma^*)\ P(s\sigma) = P(s)P(\sigma)$
- The inverse image map of P is $P^{-1} : pwr(\Sigma'^*) \rightarrow pwr(\Sigma^*)$ with

$$(\forall A \subseteq \Sigma'^*)\ \ P^{-1}(A) := \{s \in \Sigma^* \mid P(s) \in A\}$$

$$\text{a b c a c c d} \xrightarrow[P]{\Sigma = \{a, b, c, d\}\ \ \Sigma' = \{a, d\}} \text{a} \quad \text{a} \quad \text{d}$$

Synchronous Product over Languages

- Builds a more complex automaton



- with more complex language

$$L_m(A_1) \,\|\, L_m(A_2) \;=\; P_1^{-1}(L_m(A_1)) \;\cap\; P_2^{-1}(L_m(A_2))$$

expressed by natural projections

$$P_i: (\Sigma_1 \cup \Sigma_2)^* \rightarrow \Sigma_i^* \quad (i = 1,2)$$

The synchronous product is *commutative* and *associative* !

Implement Synchronous Product by Automaton Operation

- Let $G_1 = (X_1, \Sigma_1, \xi_1, x_{0,1}, X_{m,1})$ and $G_2 = (X_2, \Sigma_2, \xi_2, x_{0,2}, X_{m,2})$,
- Let

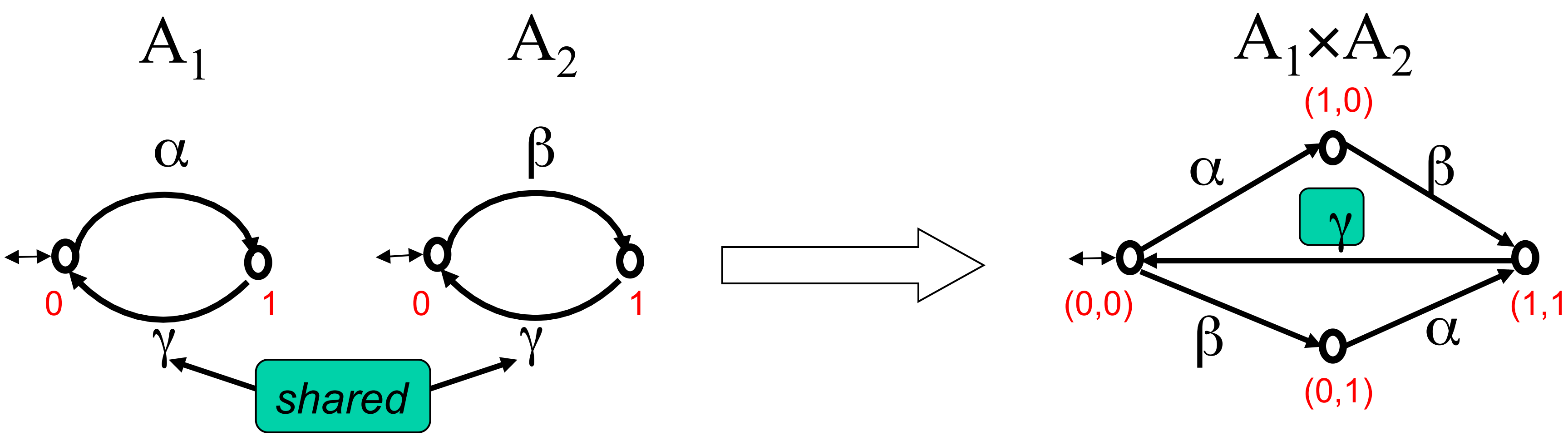
$$G_1 \times G_2 = (X_1 \times X_2, \Sigma_1 \cup \Sigma_2, \xi_1 \times \xi_2, (x_{0,1}, x_{0,2}), X_{m,1} \times X_{m,2})$$

where

$$\xi_1 \times \xi_2((x_1, x_2), \sigma) := \begin{cases} (\xi_1(x_1, \sigma), x_2) & \text{if } \sigma \in \Sigma_1 - \Sigma_2 \\ (x_1, \xi_2(x_2, \sigma)) & \text{if } \sigma \in \Sigma_2 - \Sigma_1 \\ (\xi_1(x_1, \sigma), \xi_2(x_2, \sigma)) & \text{if } \sigma \in \Sigma_1 \cap \Sigma_2 \end{cases}$$

- Result:
  - $L(G_1) \| L(G_2) = L(G_1 \times G_2)$
  - $L_m(G_1) \| L_m(G_2) = L_m(G_1 \times G_2)$

For Example



Automaton product implements synchronous product!

Properties of Projection and Synchronous Product

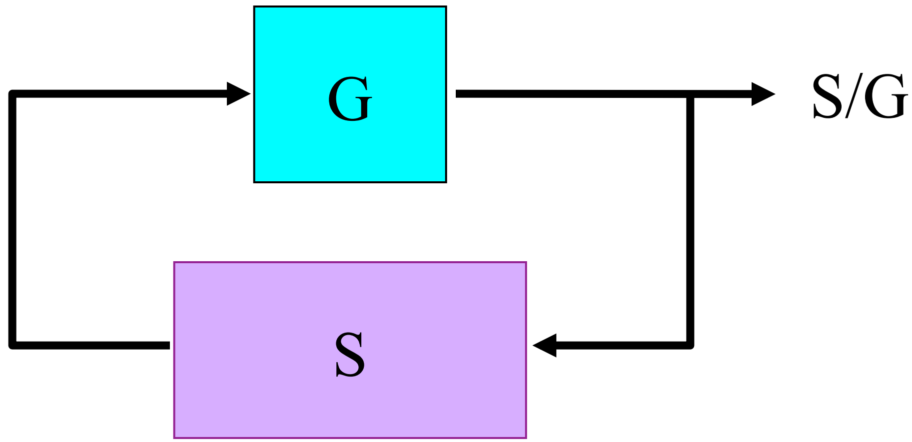
- [Chain Rule] Given Σ_1, Σ_2 and Σ_3, suppose $\Sigma_3 \subseteq \Sigma_2 \subseteq \Sigma_1$.
  - Let $P_{12}:\Sigma_1^* \rightarrow \Sigma_2^*$, $P_{23}:\Sigma_2^* \rightarrow \Sigma_3^*$ and $P_{13}:\Sigma_1^* \rightarrow \Sigma_3^*$ be natural projections
  - Then $P_{13} = P_{23}P_{12}$

- [Distribution Rule] Given $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, let $\Sigma' \subseteq \Sigma_1 \cup \Sigma_2$.
  - Let $P:(\Sigma_1 \cup \Sigma_2)^* \rightarrow \Sigma'^*$ be the natural projection. Then
    - $P(L_1 \,||\, L_2) \subseteq P(L_1) \,||\, P(L_2)$
    - $\Sigma_1 \cap \Sigma_2 \subseteq \Sigma' \Rightarrow P(L_1 \,||\, L_2) = P(L_1) \,||\, P(L_2)$

We now talk about control …

The Control Architecture

enable/disable events in Σ_c



$\Sigma = \Sigma_c \cup \Sigma_{uc}$

$\Sigma = \Sigma_o \cup \Sigma_{uo}$

Σ_c:=controllable alphabet

Σ_o:=observable alphabet

- Given a plant G and a requirement SPEC, compute a supervisor S
  - $L_m(S/G) := L_m(S) || L_m(G) \subseteq L_m(G) || L_m(SPEC)$
  - S should not disable the occurrence of any uncontrollable event
  - S should make a move only based on observable outputs of G
  - S/G is nonblocking

General Control Issues

Q1 : Is there a control that enforces both **safety**, and **liveness (nonblocking)**, and which is **maximally permissive** ?

Q2 : If so, can its design be **automated** ?

Q3 : If so, with **acceptable computing effort** ?

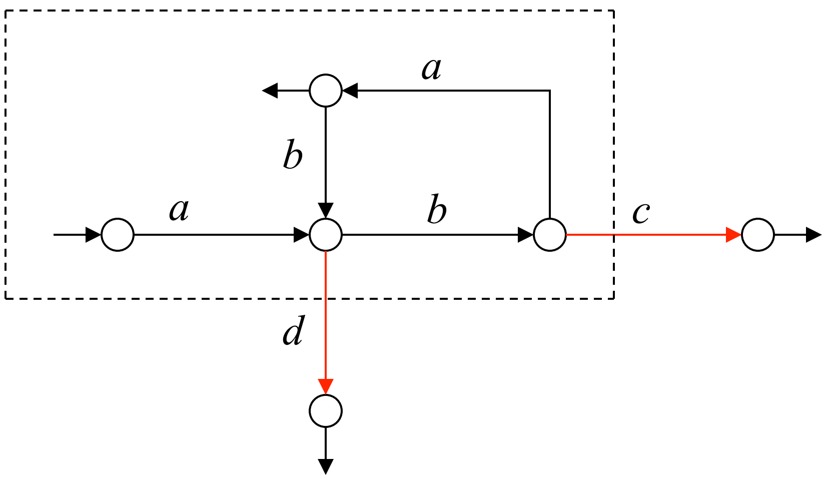
Solution to Question 1

- Fundamental **definition**

  A sublanguage $K \subseteq L_m(G)$ is ***controllable*** (w.r.t. G) if

$$\overline{K}\Sigma_{uc} \cap L(G) \subseteq \overline{K}$$

  – "Once in $\overline{K}$, you can't skid out on an uncontrollable event."



$\Sigma = \{a,b,c,d\}$

$\Sigma_c = \{a,c,d\}$

$\Sigma_{uc} = \{b\}$

Supremal Controllable Sublanguage

- Given a plant G and a specification SPEC (both over Σ), let

  $\mathcal{C}(G,SPEC):=\{K\subseteq L_m(G)\cap L_m(SPEC) \mid K \text{ is controllable w.r.t. } G\}$

- $\mathcal{C}(G,SPEC)$ is a poset under set inclusion and closed under arbitrary union
  - The largest element is called the *supremal* controllable sublanguage,

Fundamental Result

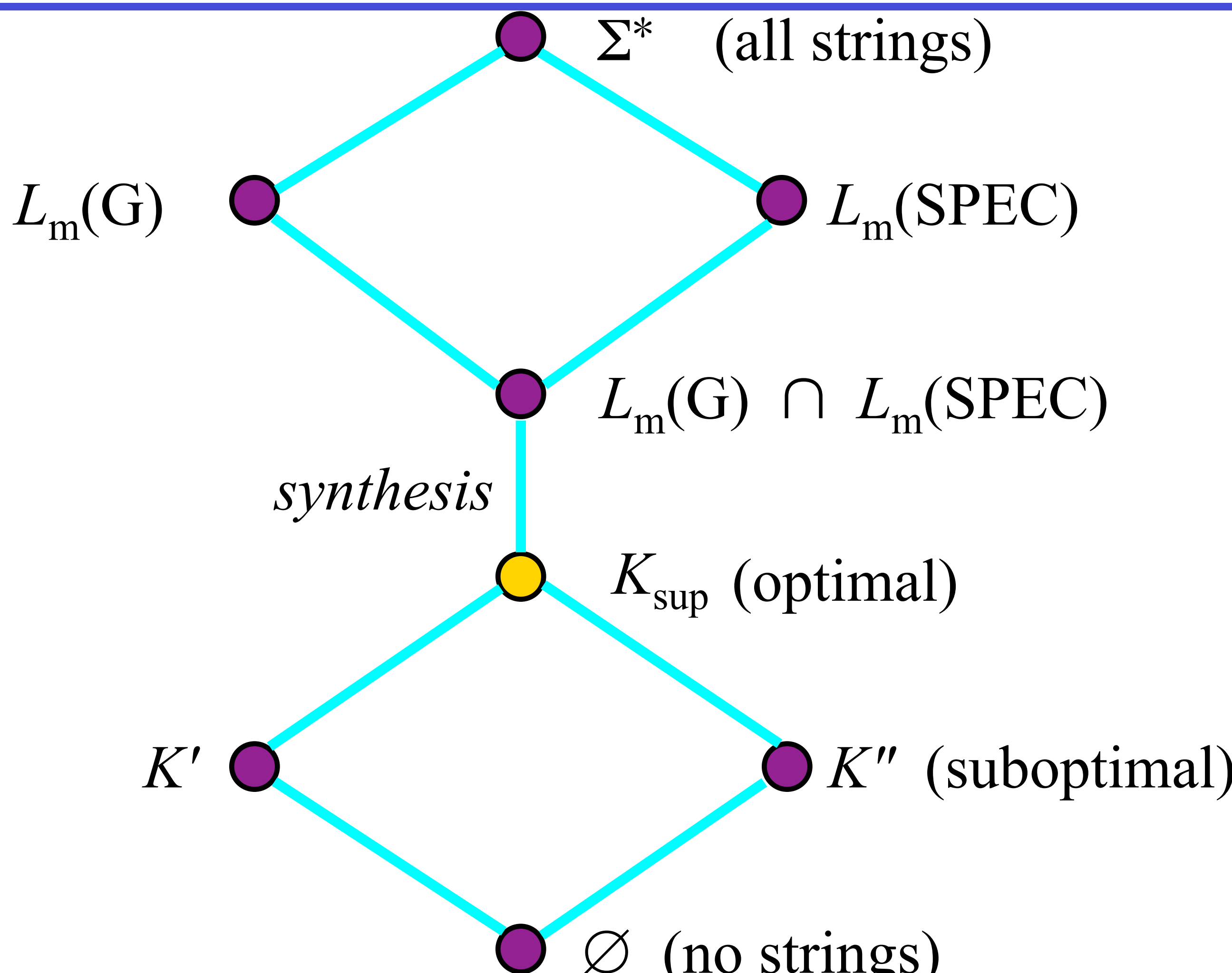
- There exists a (unique) ***supremal*** controllable sublanguage

$$K_{\text{sup}} \subseteq L_m(\mathrm{G}) \cap L_m(\mathrm{SPEC})$$

  – SPEC is an automaton model of a specification

- Furthermore K_{sup} can be effectively computed.

Lattice View of Solution to Question 1



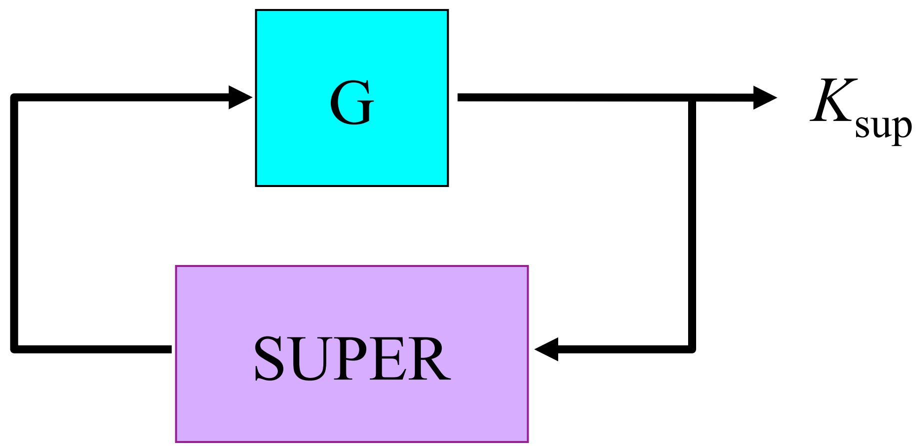
Solution to Question 2

- Given G and SPEC, compute K_{sup}

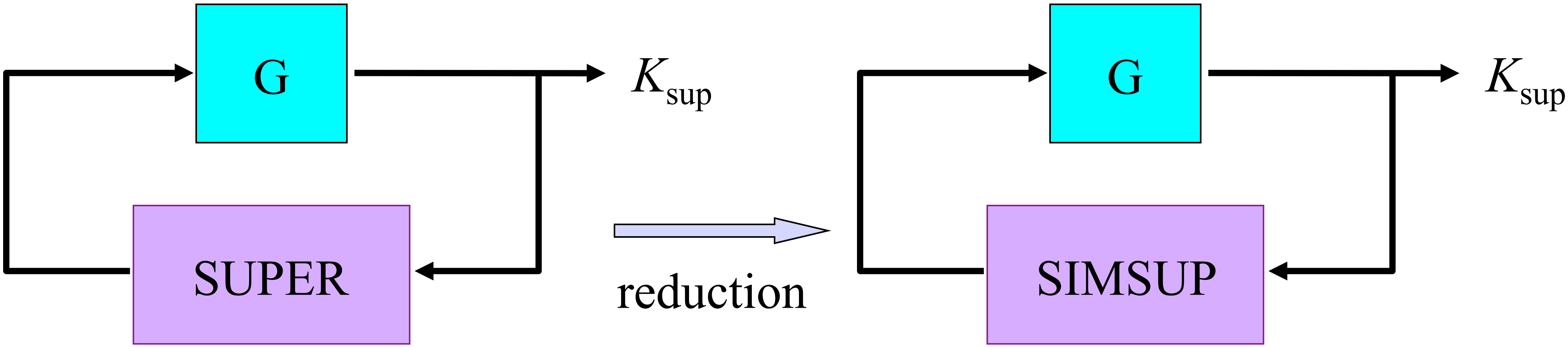
$$K_{\text{sup}} = L_{\text{m}}(\text{SUPER})$$
$$\text{SUPER} = \textbf{Supcon}\,(\text{G , SPEC})$$

- Given SUPER, implement K_{sup}

enable/disable events in Σ_c



Supervisor Reduction



SUPER and SIMSUP is *control equivalent* if

- L(G))∩L(SUPER) = L(G))∩L(SIMSUP)
- L_m(G))∩L_m(SUPER) = L_m(G))∩L_m(SIMSUP)

Supervisor Reduction

- Controlled behavior has *state size*

$$\|L_m(\text{SUPER})\| \leq \|L_m(G)\| \times \|L_m(\text{SPEC})\|$$

- Compute ***reduced, control- equivalent*** SIMSUP, often with

$$\|L_m(\text{SIMSUP})\| \ll \|L_m(\text{SUPER})\|$$

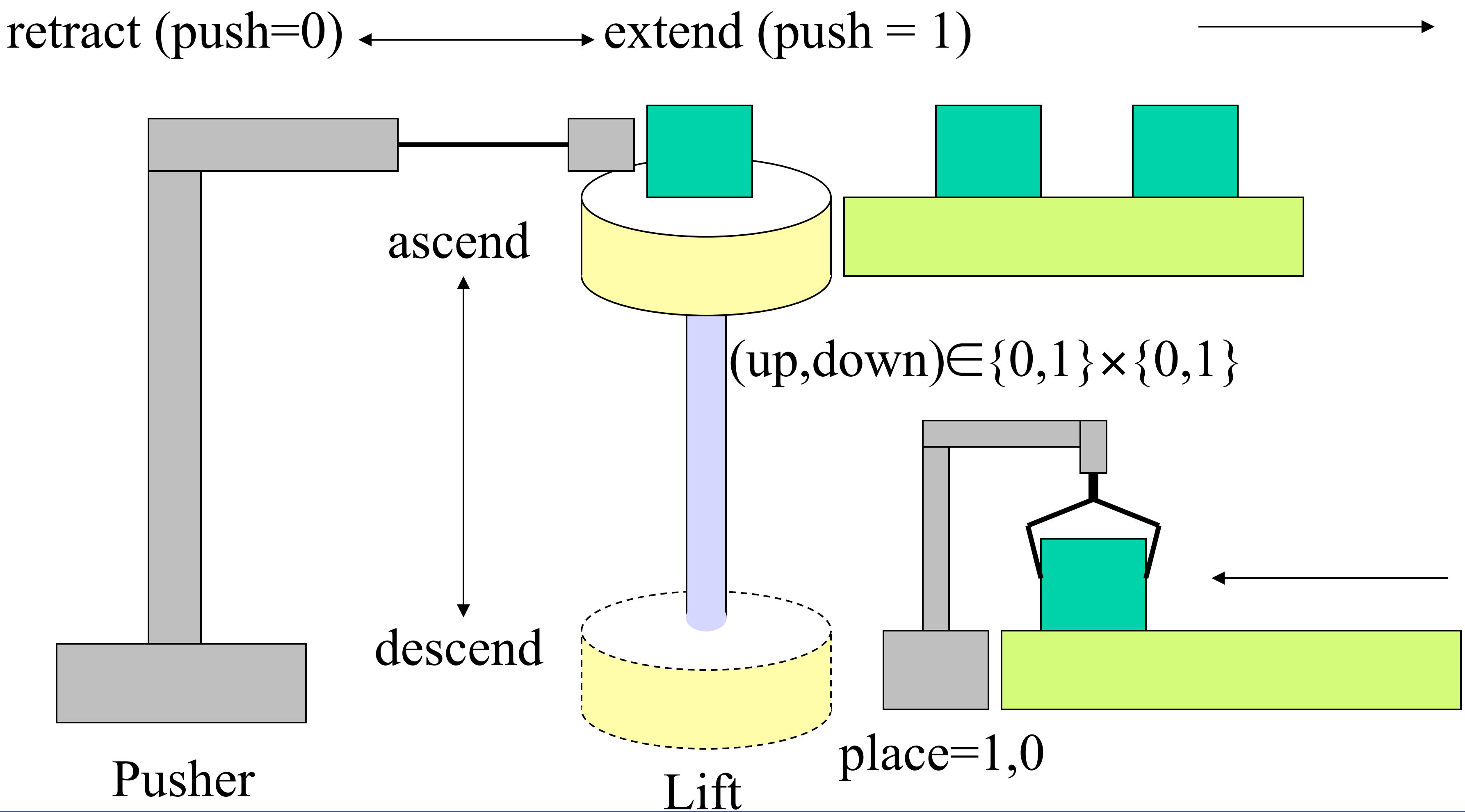
- In TCT:
  - CONSUPER = Condat(G,SUPER)
  - SIMSUP = Supreduce(G,SUPER,CONSUPER)

A solution to Question 3 is *modular*/*distributed*/*hierarchical* control

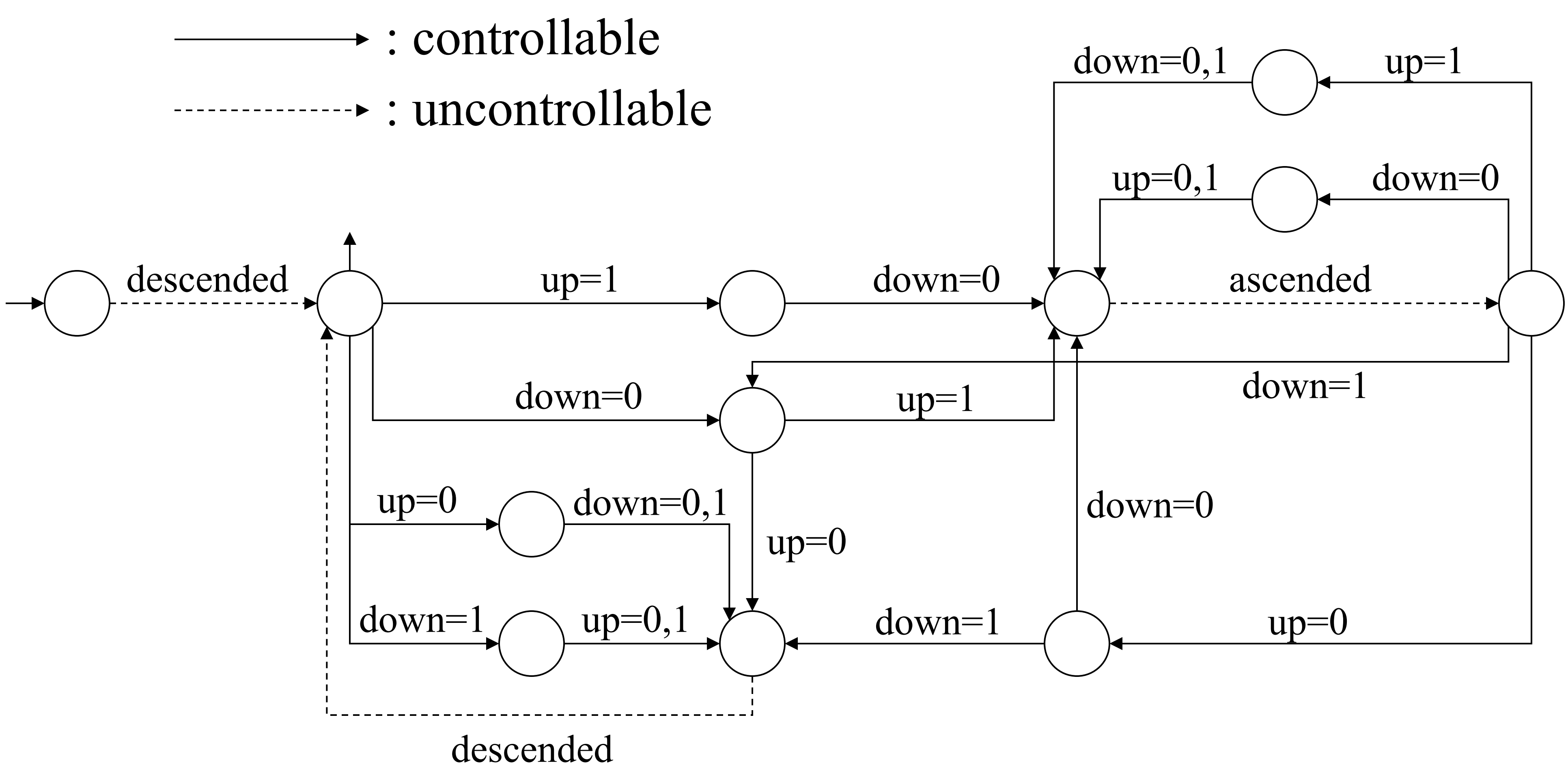
Outline

- Introduction to Supervisory Control
- Ramadge-Wonham Supervisory Control Theory
- Example – A Pusher-Lift System
- Primary Goals of EE6226

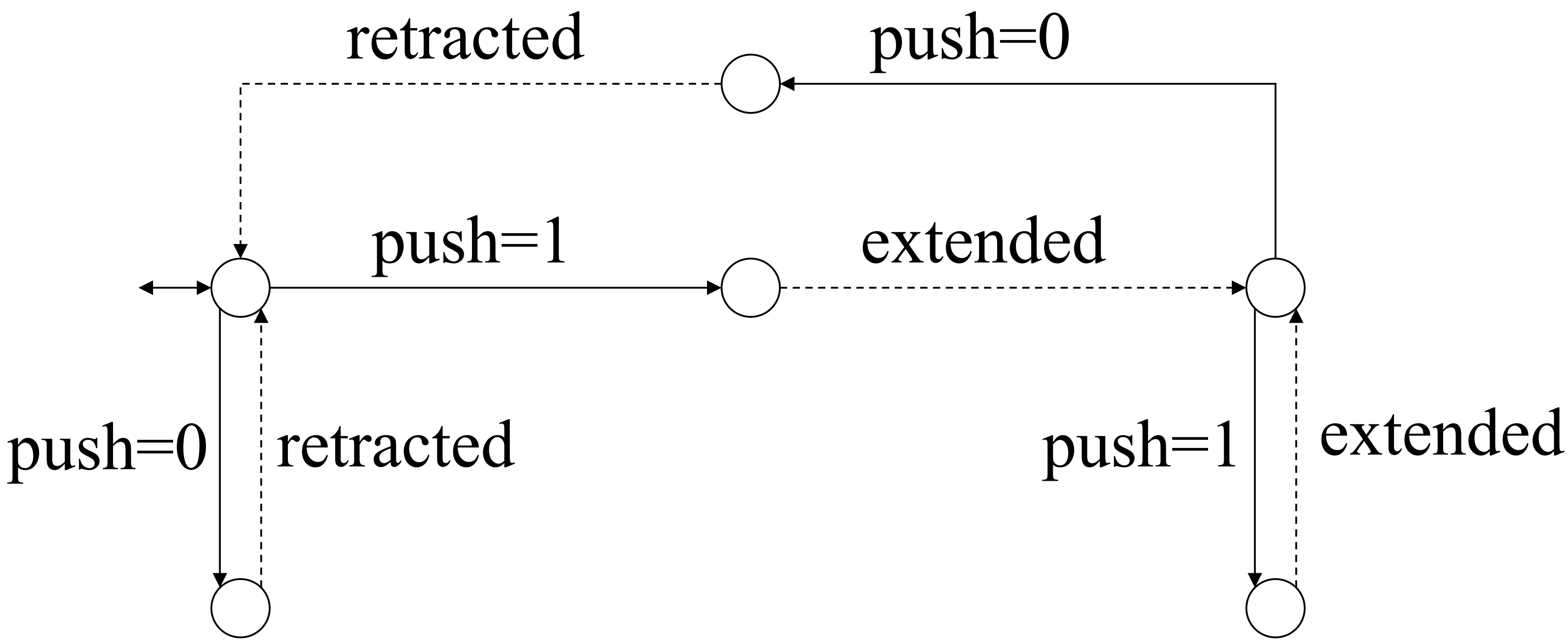
A Pusher-Lift System



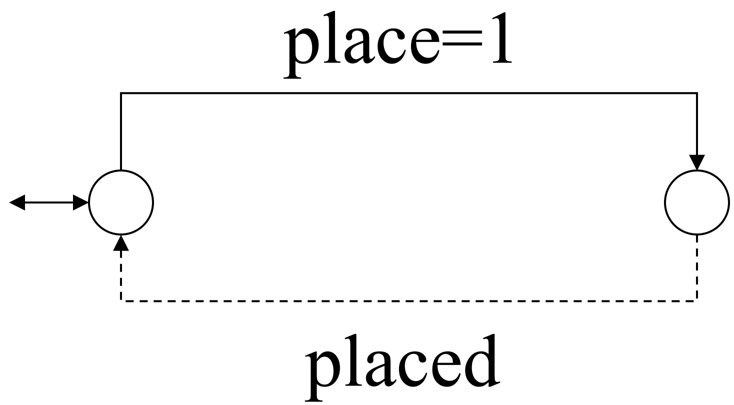
Lift Model G_{lift}



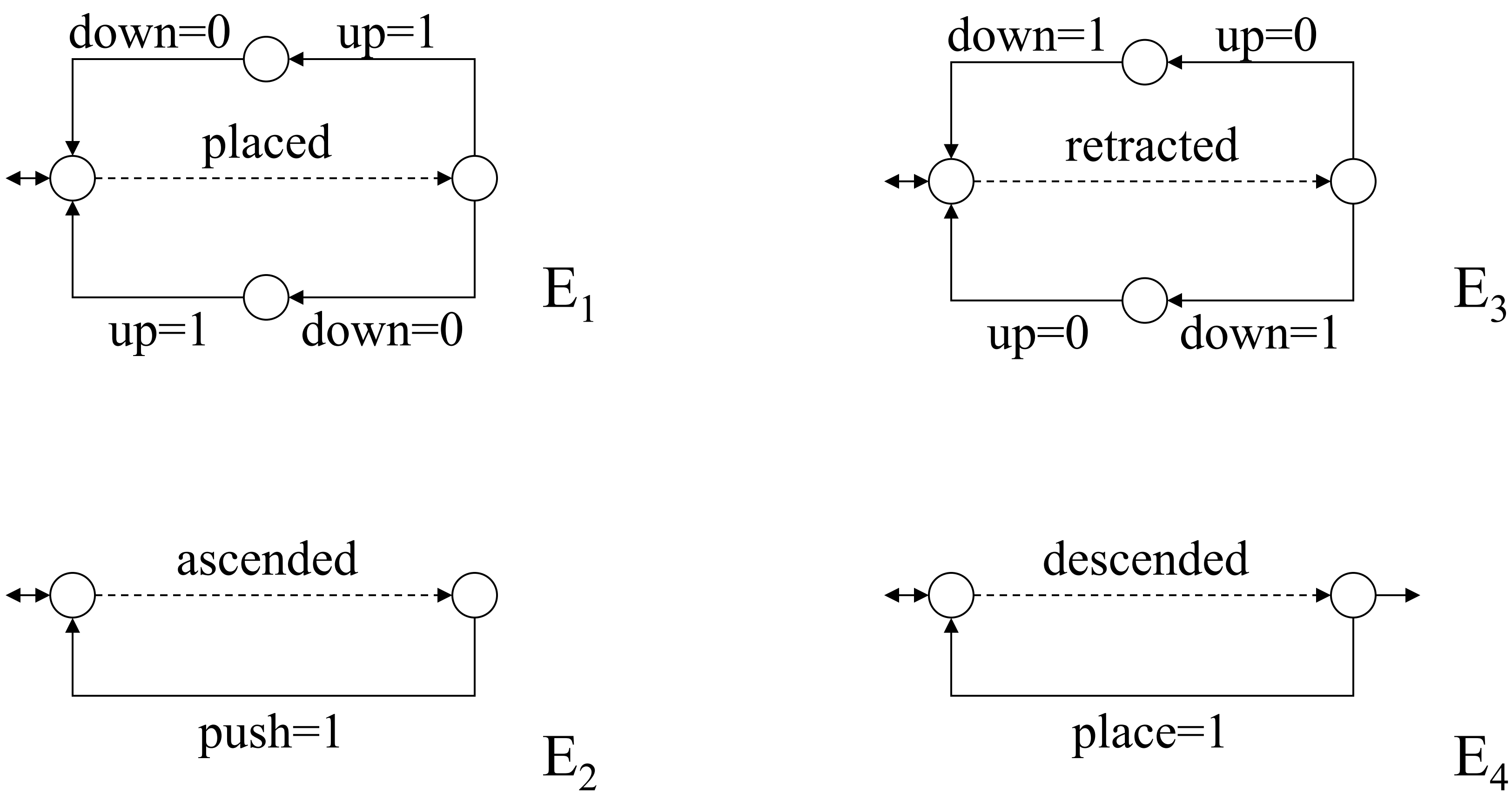
Pusher Model G_{pu}



Product Model G_{pro}



Specifications



Monolithic Method – Supervisor Synthesis

- Plant: $G = G_{lift,lo} \times G_{pu} \times G_{pro}$ (use Sync in TCT (240 , 956))
- Specification:
  - $E = E_1 \times E_2 \times E_3 \times E_4$ (64 , 288)
  - $E = \text{Selfloop}(E_1 \times E_2 \times E_3 \times E_4, \Sigma - (\Sigma_1 \cup \Sigma_2 \cup \Sigma_3 \cup \Sigma_4))$
- SUPER = Supcon(G , E) (636 , 1369)
- SUPER = Condat(G , SUPER) : controllable
- SIMSUPER = Supreduce(G,SUPER,SUPER) (99 , 476 ; slb=51)

Some Remarks

- Advantages of RW SCT
  - It is conceptually simple
  - Many real systems can be modeled in this framework
- Disadvantages of RW SCT
  - The computational complexity is very high for large systems
  - The implementation issues are not explicitly addressed
    - A procedure of signals→events (supervisory control)→signals is needed.
  - *Performance issues* are not well addressed
    - "Bad" behaviors are forbidden, but no specific "good" behavior is enforced.

Outline

- Introduction to Supervisory Control
- Ramadge-Wonham Supervisory Control Theory
- Example – A Pusher-Lift System
- Primary Goals of EE6226

Goals of EE6226

- To introduce several techniques that are aimed to handle the complexity issue involved in supervisor synthesis.
  - Modular control
  - Distributed control
  - Hierarchical control
  - State-feedback control
- To deal with supervisory control under partial observations.
- To address a certain type of performance.

Basic Functions of Supervisor Synthesis Package

Developed by R. Su

Nanyang Technological University

Create Automata

Automaton: B1.cfg

```
[automaton]
states = 0, 1, 2, 3, 4
alphabet = tau, R1-drop-B1, R1-pick-B1, R2-drop-B1, R2-pick-B1
controllable =  R1-drop-B1, R1-pick-B1, R2-drop-B1, R2-pick-B1
observable =  R1-drop-B1, R1-pick-B1, R2-drop-B1, R2-pick-B1
transitions = (0, 1, tau), (1, 2, R1-drop-B1), (2, 1, R2-pick-B1),
              (1, 3, R2-drop-B1), (3, 1, R1-pick-B1), (1, 4, R2-pick-B1),
              (1, 4, R1-pick-B1), (2, 4, R1-drop-B1), (3, 4, R2-drop-B1)
marker-states = 1
initial-state = 0
```

Check Size of Automaton

make_get_size.py

```
[user@host ~] $ make_get_size
Please input model (.cfg): B1.cfg
Number of states: 5
Number of transitions: 9
```

Automaton Product

make_product.py

```
[user@host ~]$ make_product
Please input list of your input automata (comma-seperated list of automata): B1.cfg, B2.cfg
Please input product automaton (.cfg): B1-B2.cfg
Mon Mar 16 10:33:51 2009: Must do 1 product computations.             (memory=9052160 bytes)
Mon Mar 16 10:33:51 2009: Product #1 done: 17 states, 65 transitions   (memory=9052160 bytes)
Mon Mar 16 10:33:51 2009: Computed product                             (memory=9052160 bytes)
                    Number of states: 17
                    Number of transitions: 65
Mon Mar 16 10:33:51 2009: Product is saved in B1-B2.cfg                (memory=9076736 bytes)
```

Automaton Abstraction

make_abstraction.py

```
[user@host ~]$ make_abstraction
Please input source automaton (.cfg): B1-B2.cfg
Please input list of preserved events (comma-seperated list of event names): tau, R1-drop-B1
Please input name of the abstraction (.cfg): B1-B2-abstraction.cfg
Mon Mar 16 10:40:54 2009: Computed abstraction     (memory=8364032 bytes)
                    Number of states: 5
                    Number of transitions: 14
Mon Mar 16 10:40:54 2009: Abstraction is saved in B1-B2-abstraction.cfg
          (memory=8409088 bytes)
```

Sequential Automaton Abstraction

make_sequential_abstraction.py

```
[user@host ~]$ make_sequnetial_abstraction
Please input list of your input automata (comma-seperated list of automata): B1.cfg, B2.cfg
Please input list of preserved events (comma-seperated list of event names): tau, R1-drop-B1
Please input abstraction (.cfg): B1-B2-sequential-abstraction.cfg
Mon Mar 16 13:01:23 2009: Started      (memory=8249344 bytes)
Mon Mar 16 13:01:23 2009: #states after adding 1 automata: 5          (memory=8257536 bytes)
Mon Mar 16 13:01:23 2009: #states and #transitions after abstraction: 4, 9(memory=8265728 bytes)
Mon Mar 16 13:01:23 2009: #states of 2 automata: 5; #states and #transitions of product: 13 51
            (memory=8278016 bytes)
Mon Mar 16 13:01:23 2009: #states and #transitions after abstraction: 5, 14(memory=8294400 bytes)
Mon Mar 16 13:01:23 2009: Abstraction is saved in B1-B2-sequential-abstraction.cfg
            (memory=8327168 bytes)
```

Natural Projection

make_natural_projection.py

```
[user@host ~]$ make_natural_projection
Please input source automaton (.cfg): B1-B2.cfg
Please input list of preserved events (comma-seperated list of event names): tau, R1-drop-B1
Please input name of the abstraction (.cfg): B1-B2-natural-projection.cfg
Mon Mar 16 10:46:04 2009: Computed projection      (memory=8376320 bytes)
                        Number of states: 3
                        Number of transitions: 3
Mon Mar 16 10:46:04 2009: Projected automaton is saved in B1-B2-natural-projection.cfg
             (memory=8417280 bytes)
```

Check Language Equivalence

Make_language_equivalence_test.py

```
[user@host ~]$ make_language_equivalence_test
Please input first model (.cfg): B1-B2-abstraction.cfg
Please input second model (.cfg): B1-B2-natural-projection.cfg
Language equivalence HOLDS
```

Supervisor Synthesis

make_supervisor.py

```
[user@host ~]$ make_supervisor
Please input plant model (.cfg): plant.cfg
Please input specification model (.cfg): spec.cfg
Please input supervisor (.cfg): supervisor.cfg
Mon Mar 16 12:49:59 2009: Computed supervisor     (memory=14548992 bytes)
                    Number of states: 140
                    Number of transitions: 288
Mon Mar 16 12:49:59 2009: Supervisor saved in supervisor.cfg    (memory=14536704 bytes)
```

Nonconflict Check

make_nonconflicting_check.py

```
[user@host ~]$ make_nonconflicting_check
Please input list of your input automata (comma-seperated list of automata): plant.cfg, supervisor.cfg
Mon Mar 16 12:56:21 2009: Started      (memory=14954496 bytes)
Mon Mar 16 12:56:21 2009: #states after adding 1 automata: 926       (memory=14954496 bytes)
Mon Mar 16 12:56:24 2009: #states and #transitions after abstraction: 926, 3919
            (memory=15073280 bytes)
Mon Mar 16 12:56:24 2009: #states of 2 automata: 139; #states and #transitions of product: 166 380
            (memory=15073280 bytes)
Mon Mar 16 12:56:24 2009: #states and #transitions after abstraction: 3, 6(memory=15036416 bytes)
ok
```

Check Controllability

make_controllability_check.py

```
[user@host ~]$ make_controllability_check
Please input plant model (.cfg): plant.cfg
Please input supervisor model (.cfg): supervisor.cfg
States with disabled controllable events:
    (1, 1): {R2-pick-B2, R3-pick-B2}
    (4, 2): {R2-drop-B2}
    (5, 3): {R3-drop-B2, R2-pick-B2, R3-drop-P33, R3-drop-B3}
    (10, 4): {R3-drop-B3, R2-drop-B2, R3-drop-P33}
    …………
    (799, 121): {R2-pick-B2, R3-pick-B2}

Supervisor is correct (no disabled uncontrollable events)
```

Compute Feasible Supervisor

make_feasible_supervisor.py

```
[user@host ~]$ make_feasible_supervisor
Please input plant model (.cfg): plant.cfg
Please input supervisor model (.cfg): supervisor.cfg
Please input feasible supervisor filename (.cfg): feasible_supervisor.cfg
Mon Mar 16 13:09:43 2009: Computed supervisor     (memory=10522624 bytes)
                          Number of states: 82
                          Number of transitions: 196
Mon Mar 16 13:09:43 2009: Supervisor saved in feasible_supervisor.cfg
    (memory=10547200 bytes)
```

Batch Operation

Batch_Operation.py

```
*******************************************************************************
#!/usr/bin/env python
from automata import frontend

#Compute product
frontend.make_product('B1.cfg, B2.cfg', 'B1-B2.cfg')

#Compute automaton abstraction
frontend.make_abstraction('B1-B2.cfg', 'tau,R1-drop-B1', 'B1-B2-abstraction.cfg')

#Compute supervisor
frontend.make_supervisor('plant.cfg', 'spec.cfg', 'supervisor.cfg')

#Check controllability
frontend.make_controllability_check('plant.cfg', 'supervisor.cfg')
```