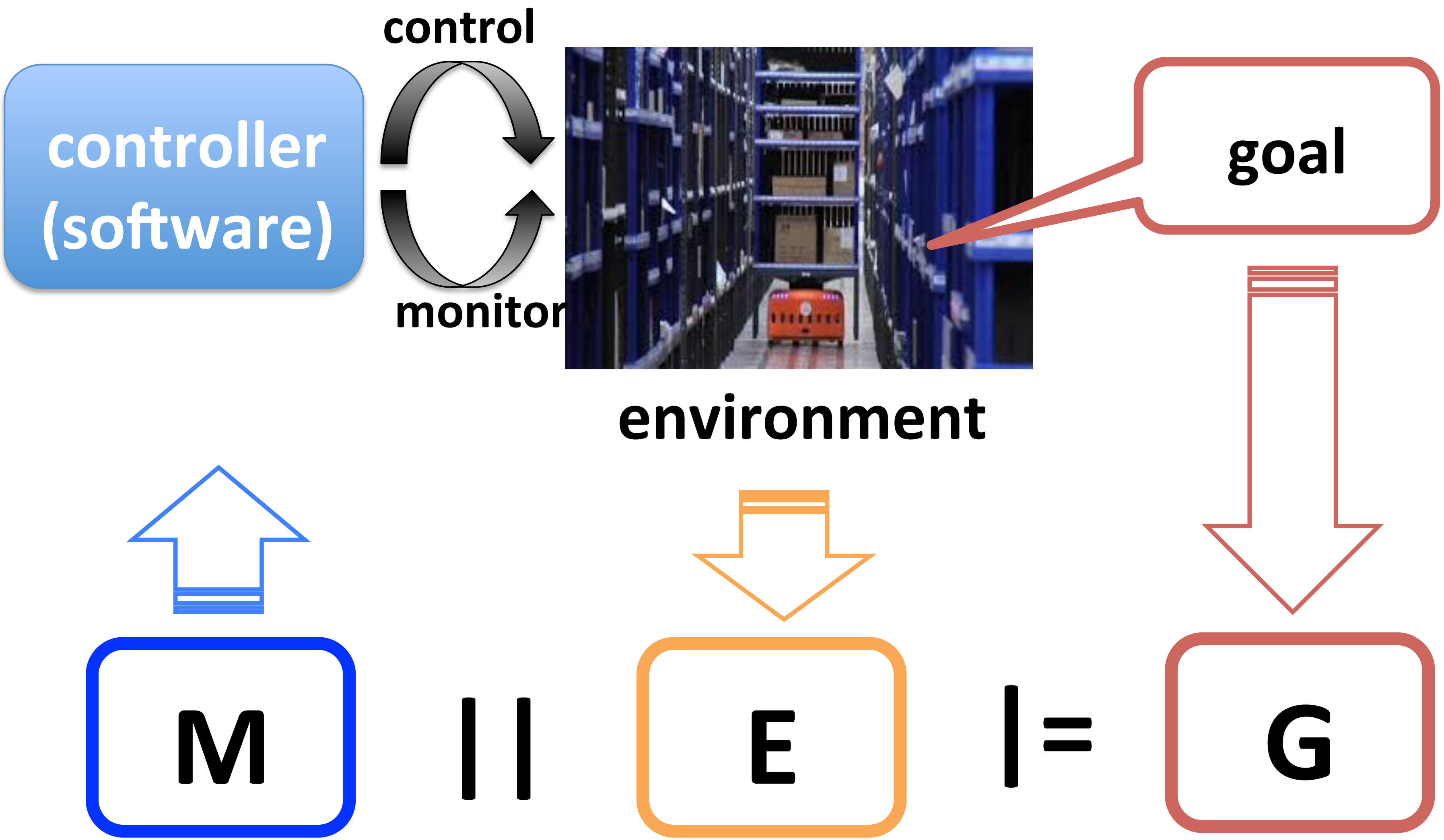
Assured Graceful Degradation with Discrete Controller Synthesis

Kenji Tei
National Institute of Informatics

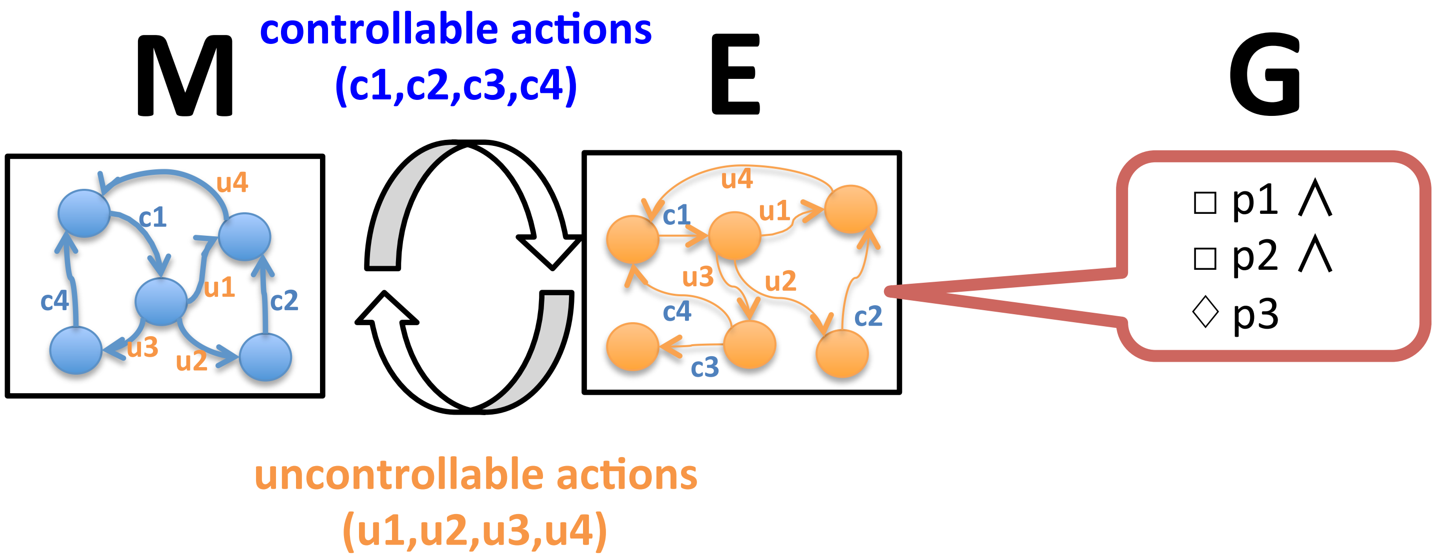
Joint work with
Kazuya Aizawa, Waseda University
Nicolas D'Ippolito, Universidad de Buenos Aires

Assurance at development time

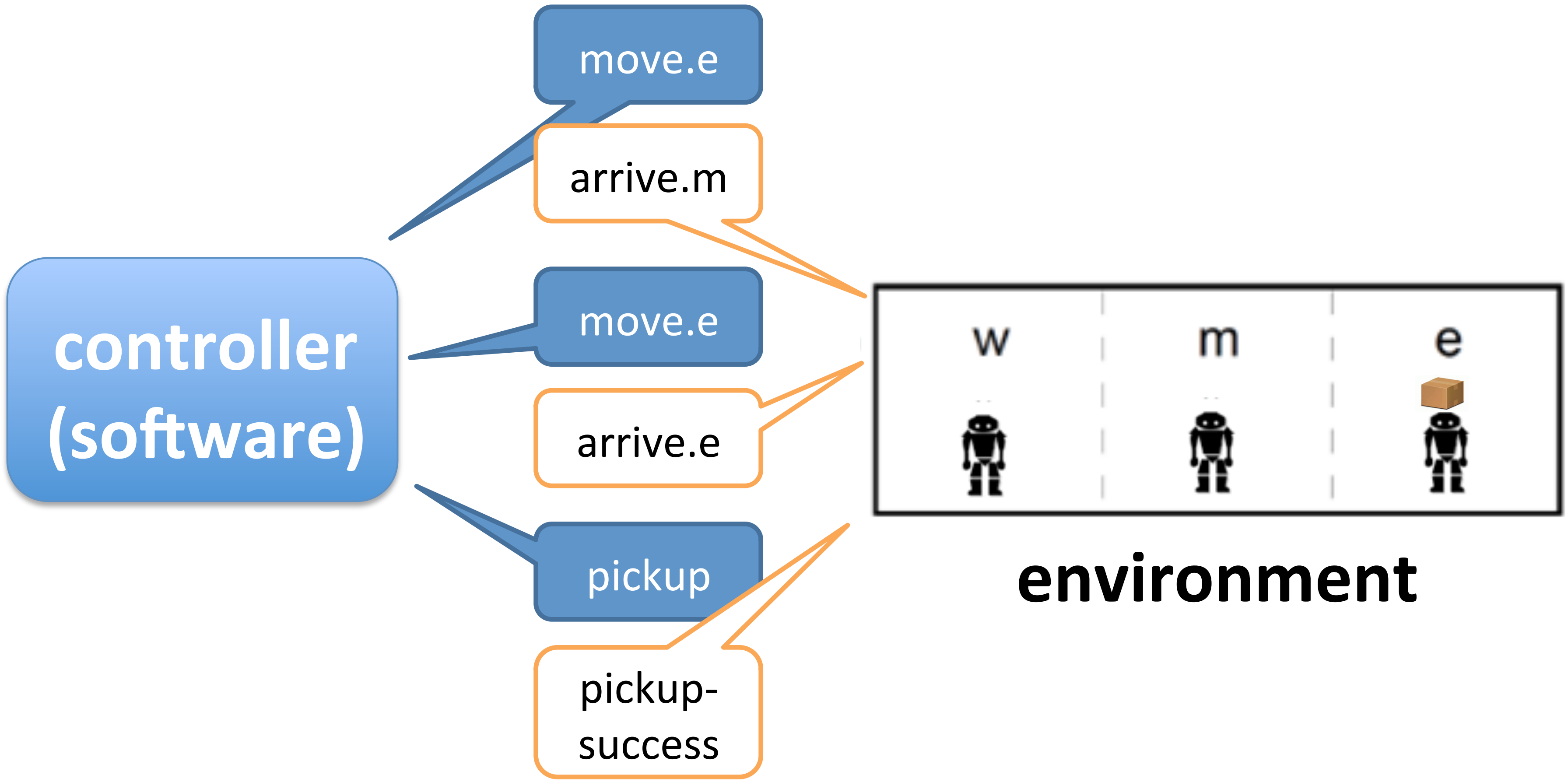


Modeling approach : LTS and FLTL

- **M** and **E** : **labeled transition system (LTS)**
- **G** : **fluent linear temporal logic (FLTL)**



Example : Automated Warehouse

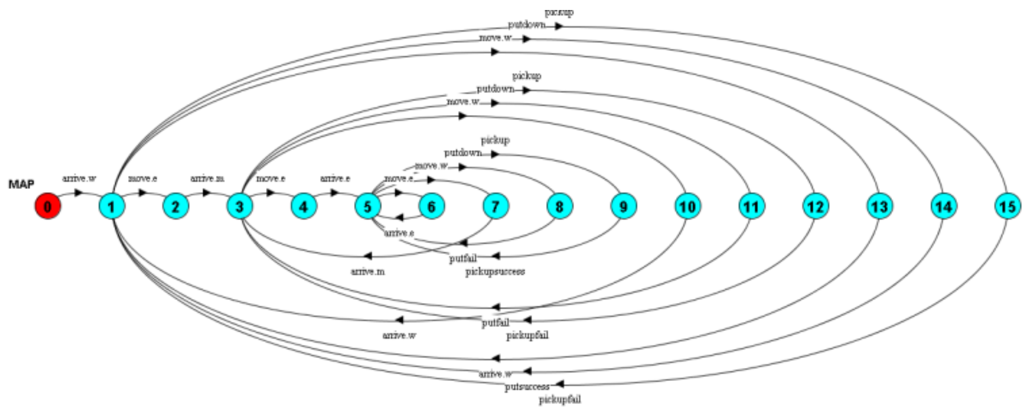


Modeling an environment by LTS

||E = (MAP||W_ROBOT).

```
MAP=(arrive['w] -> MAP['w]),
MAP['w]=( move['e] -> arrive['m] -> MAP['m]
 | move['w] -> arrive['w] -> MAP['w]
 | putdown -> putsuccess -> MAP['w]
 | pickup -> pickupfail -> MAP['w]),
MAP['m]=( move['e] -> arrive['e] -> MAP['e]
 | move['w] -> arrive['w] -> MAP['w]
 | putdown -> putfail -> MAP['m]
 | pickup -> pickupfail -> MAP['m]),
MAP['e]=( move['e] -> arrive['e] -> MAP['e]
 | move['w] -> arrive['m] -> MAP['m]
 | putdown -> putfail -> MAP['e]
 | pickup -> pickupsuccess -> MAP['e]).
```

```
W_ROBOT=(arrive['w] -> ROBOT),
ROBOT= (move[Direction] -> arrive[Locations] -> ROBOT
 | pickup -> (pickupsuccess -> ROBOT | pickupfail -> ROBOT)
 | putdown -> (putsuccess -> ROBOT | putfail -> ROBOT)
 | ended -> reset -> ROBOT).
```



Modeling how the state of the env. is changed
and how the env. will react

Specifying Goals by FLTL

[]((AT['w] && X(move['e])) -> X(!arrive['w] W pickupsuccess))

[]((AT['e] && X(move['w])) -> X(!arrive['e] W putsuccess))

[](putdown->AT['w])

[]!(!<pickupsuccess,putsuccess> && putdown)

[](pickup->AT['e])

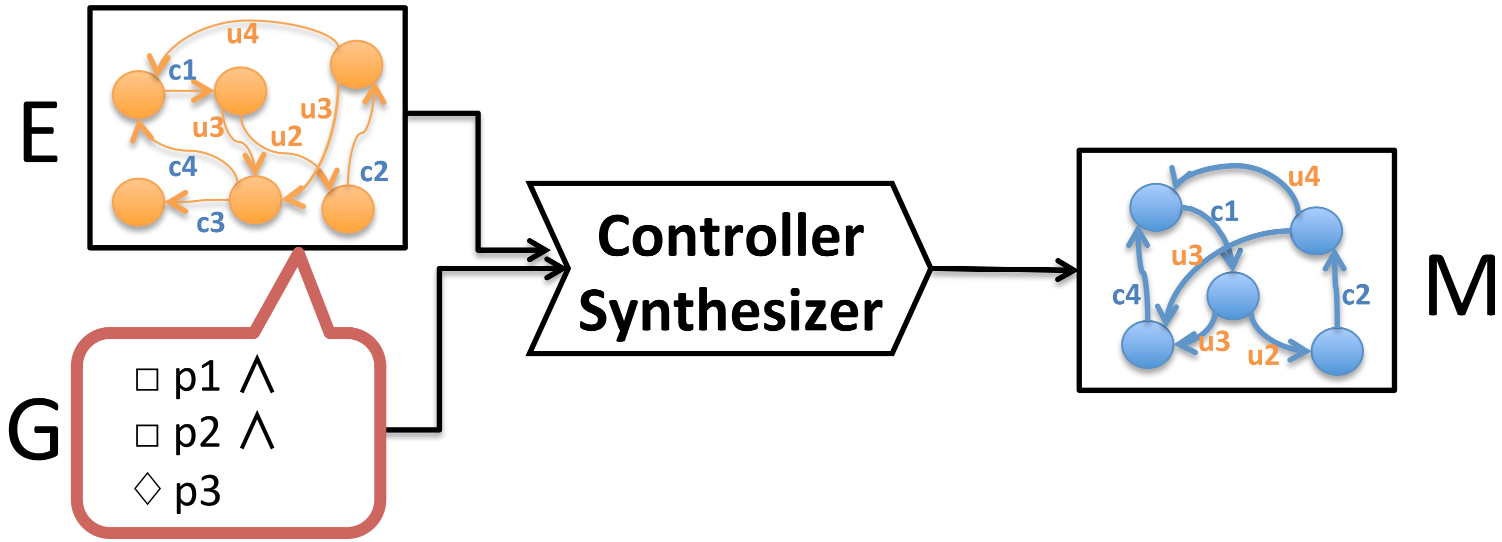
[]!(<pickupsuccess,putsuccess> && pickup)

[](<ended,reset> -> (<pickupsuccess,{reset}> && <putsuccess,{reset}>))

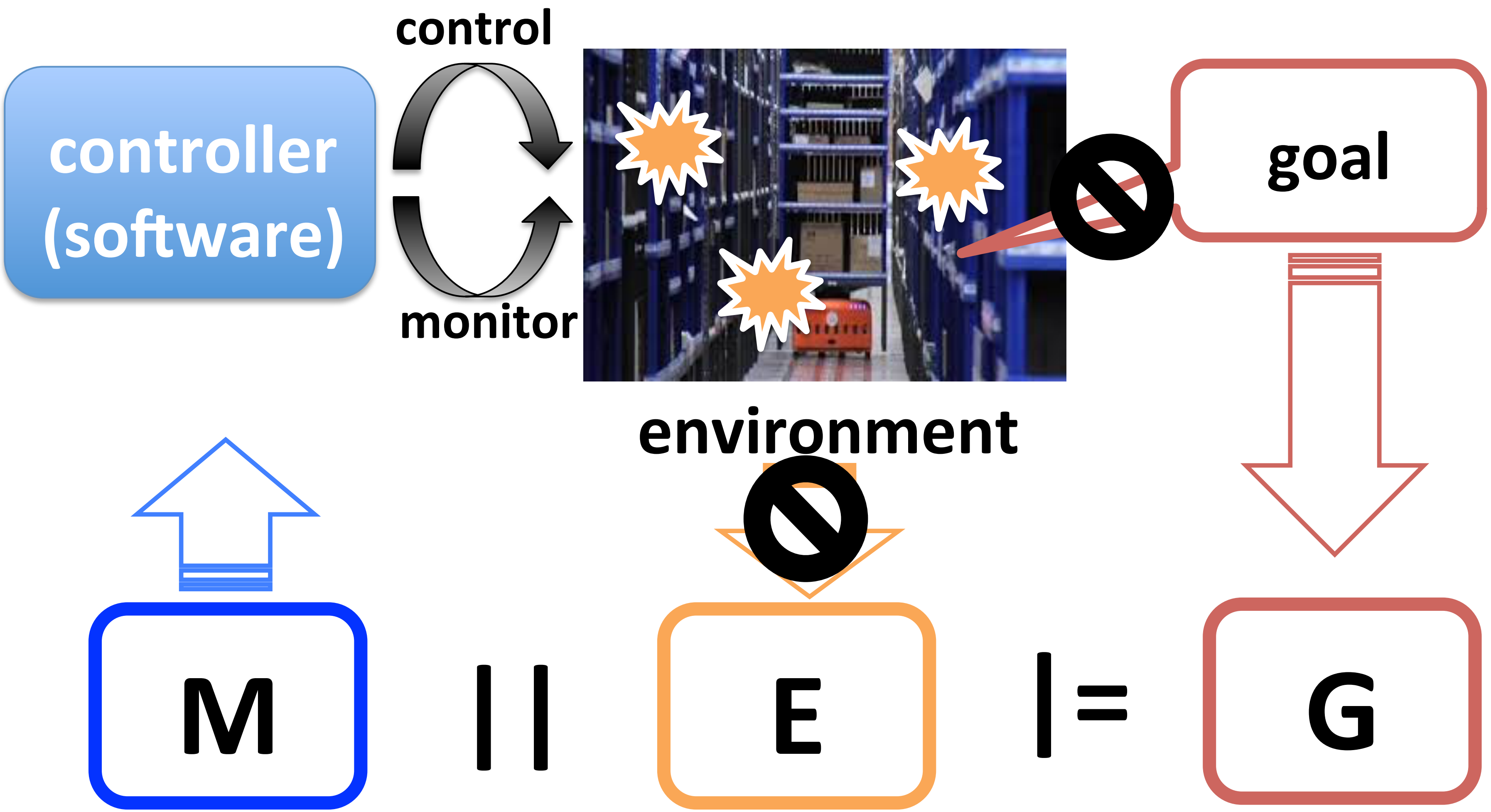
※ fluent AT[x:Locations] = <arrive[x],{move[Direction]}\{move[x]}>

A way to generate M with assurance

- discrete controller synthesis
  [D'Ippolito, 2010] [D'Ippolito, 2011]
  - solve a control problem ***<E,G>*** to find an LTS ***M***



E may be invalid at runtime



System ***may no longer work***,
or ***may continue***, but ***without any assurances***

Assuming More Realistic

```
...
MAP['w]=( move['e] -> arrive['m] -> MAP['m]
 | move['w] -> arrive['w] -> MAP['w]
 | putdown -> putsuccess -> MAP['w]
 | pickup -> pickupfail -> MAP['w]),
...
```

[]((AT['w] && X(move['e])) -> X(!arrive['w] W pickupsuccess))

[]((AT['e] && X(move['w])) -> X(!arrive['e] W putsuccess))

[](putdown->AT['w])

[]!(!<pickupsuccess,putsuccess> && putdown)

[](pickup->AT['e])

[]!(<pickupsuccess,putsuccess> && pickup)

[](<ended,reset> -> (<pickupsuccess,{reset}> && <putsuccess,{reset}>))

```
...
MAP['w]=(move['e] -> (arrive['m] -> MAP['m]
                     | (arrive['w] -> MAP['w]))
 | move['w] -> arrive['w] -> MAP['w]
 | putdown -> putsuccess -> MAP['w]
 | pickup -> pickupfail -> MAP['w]),
...
```

~~[]((AT['w] && X(move['e])) -> X(!arrive['w] W pickupsuccess))~~

[]((AT['e] && X(move['w])) -> X(!arrive['e] W putsuccess))

[](putdown->AT['w])

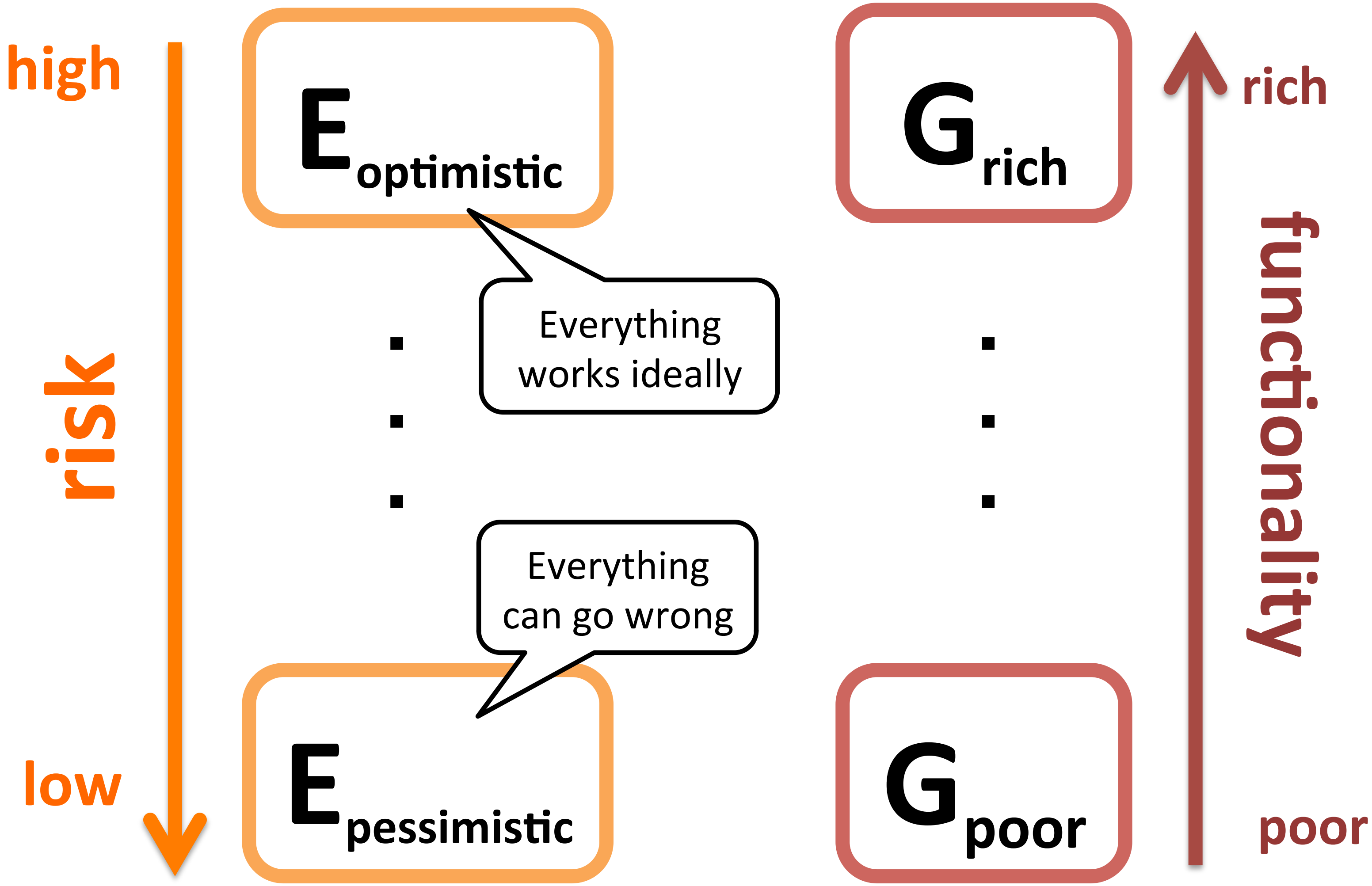
[]!(!<pickupsuccess,putsuccess> && putdown)

[](pickup->AT['e])

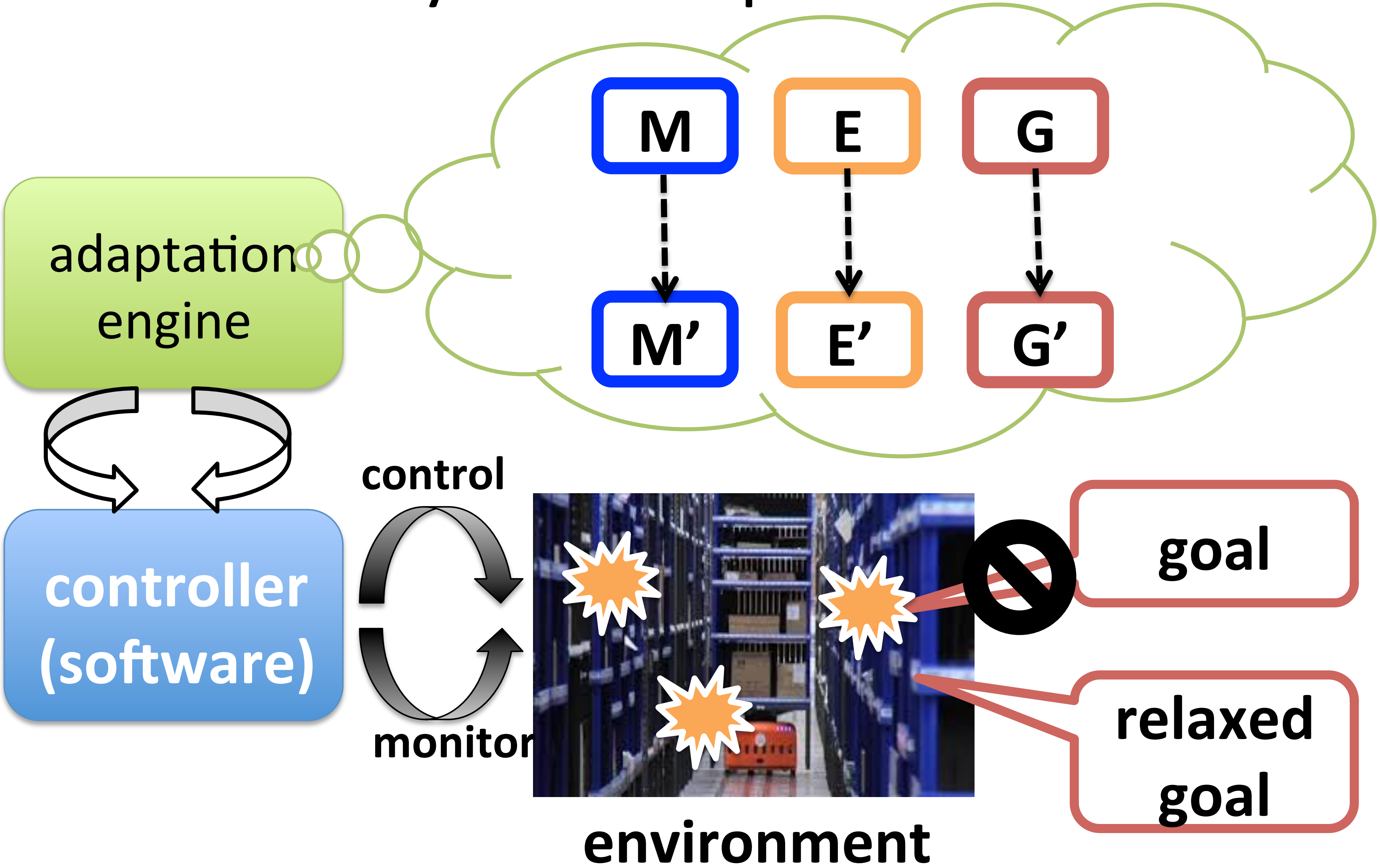
[]!(<pickupsuccess,putsuccess> && pickup)

[](<ended,reset> -> (<pickupsuccess,{reset}> && <putsuccess,{reset}>))

How much should we assume?



Graceful Degradation by Self-adaptation



Questions

How can the system be made to degrade gracefully with assurance?

How can the system determine how much it should degrade?

Objective

We propose a framework for adaptation engine enabling graceful degradation

G ---> G' **Avoiding too much degradation**

– should not degrade the system too much

M' || E' |= G' **Providing assurance**

– should assure that the system after degradation satisfies a selected level of goals

M' simulate M **Performing degradation seamlessly**

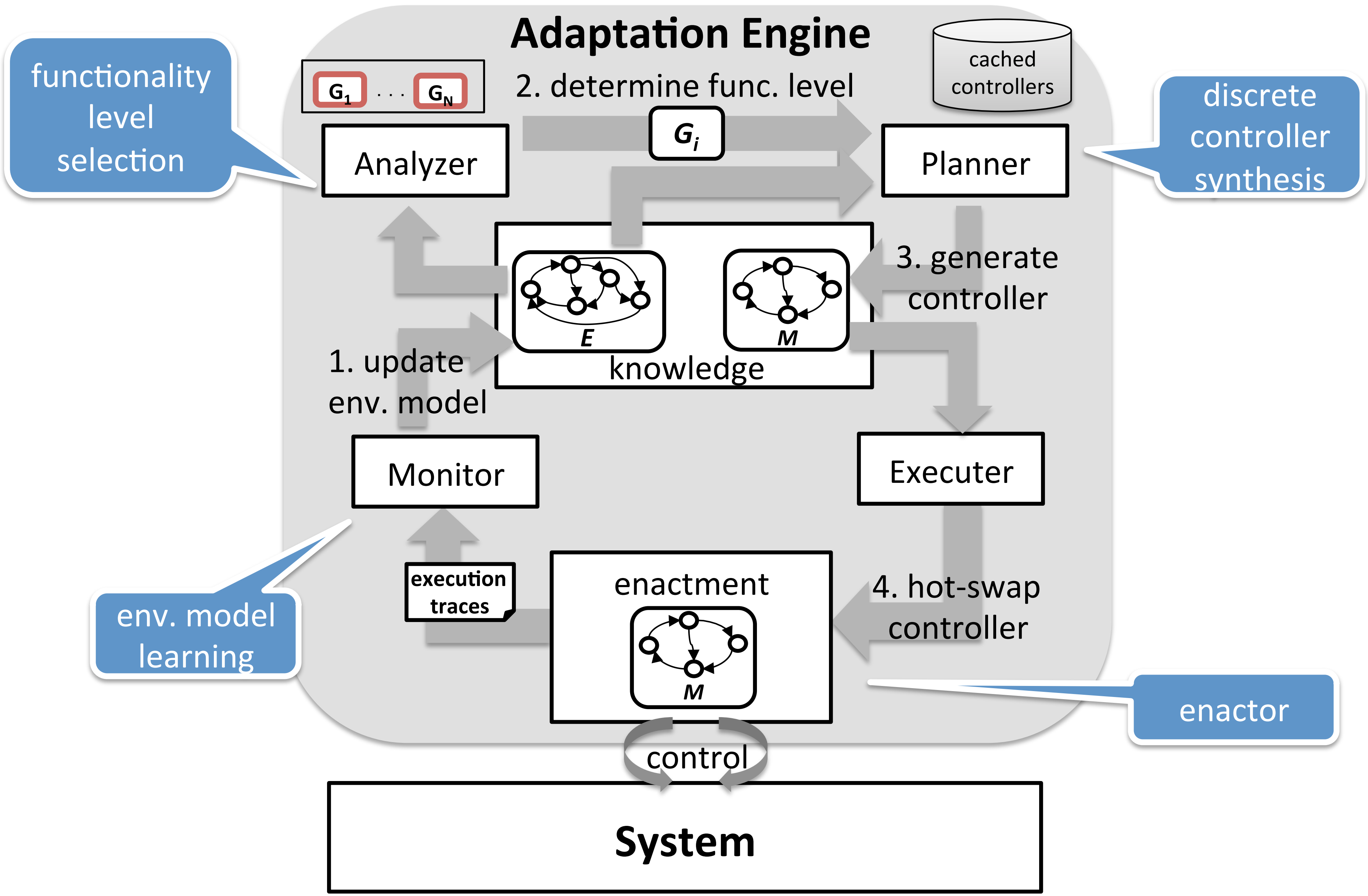
– should not stop or restart the system

– M' should simulate M

Approach : Models@Run.time

- Revising environment model at runtime
  - to fit the environment
- Generating behavior specification with assurance at runtime
  - by using algorithmic techniques, in particular ***discrete controller synthesis***
- Change behavior of the system in accordance with the generated model

Overall architecture



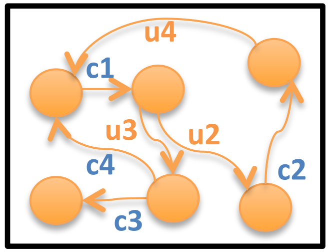
0. Initialization

1. Specify levels of functionalities

G_1 . . . G_N

2. Describe the initial environment model

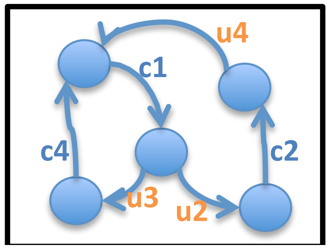
E_0



3. Select a func. level and construct the initial controller

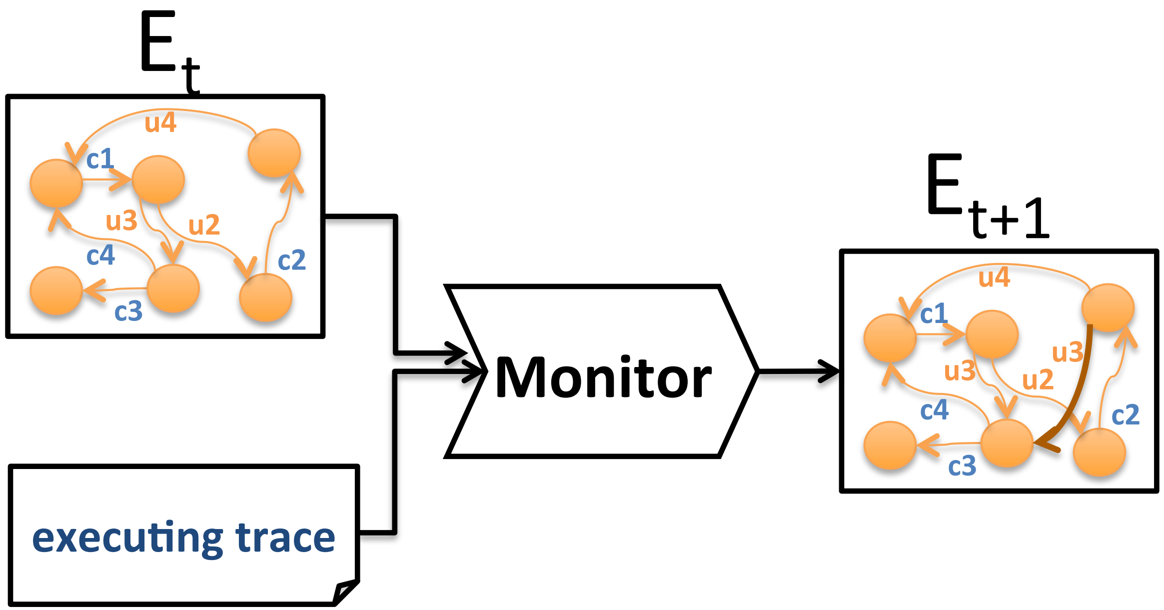
G_i

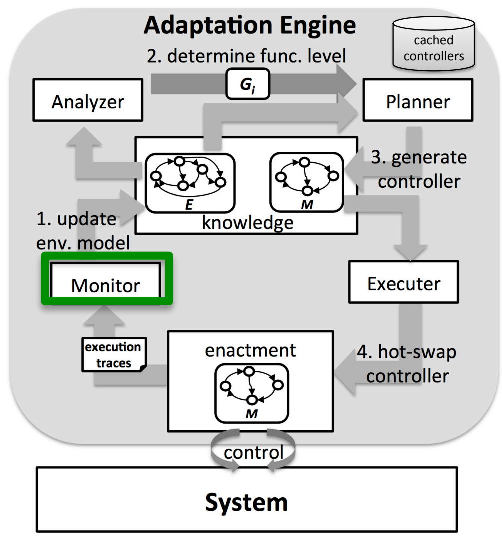
M_0



1. Monitor: Environment Model Updates

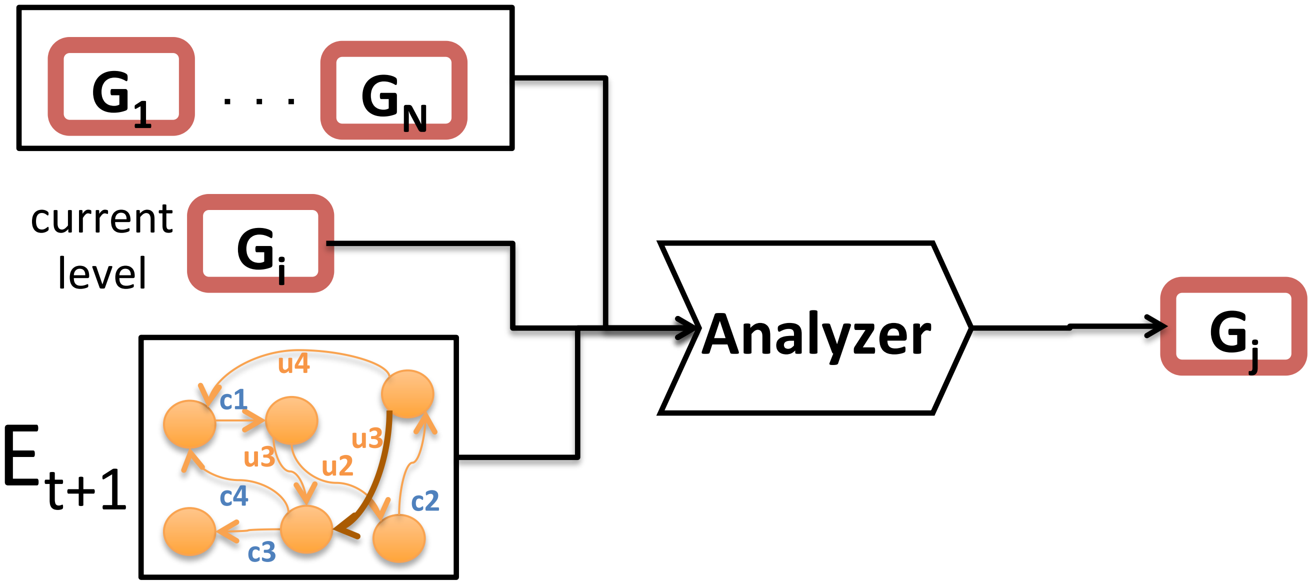
- update E_t to generate E_{t+1} so that E_{t+1} can explain execution traces of the system
  - find and add unmodeled uncontrollable transitions Δ_{t+1}
    - When a robot performed "move.w" action at "e", the environment will respond "arrive.m" or ***"arrive.e"***
  - ***rule learning for environment model update***[Sykes,2013]

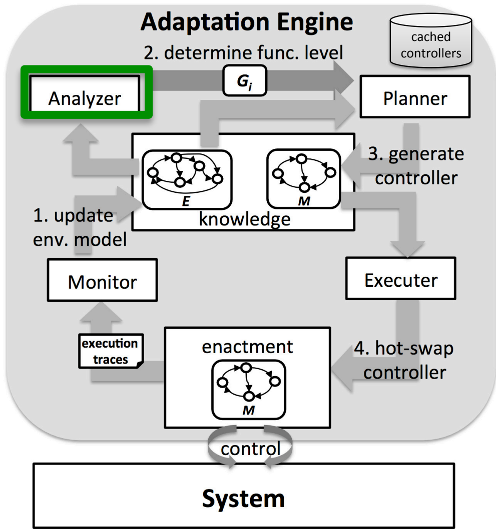




2. Analysis: Functionality-level Selection

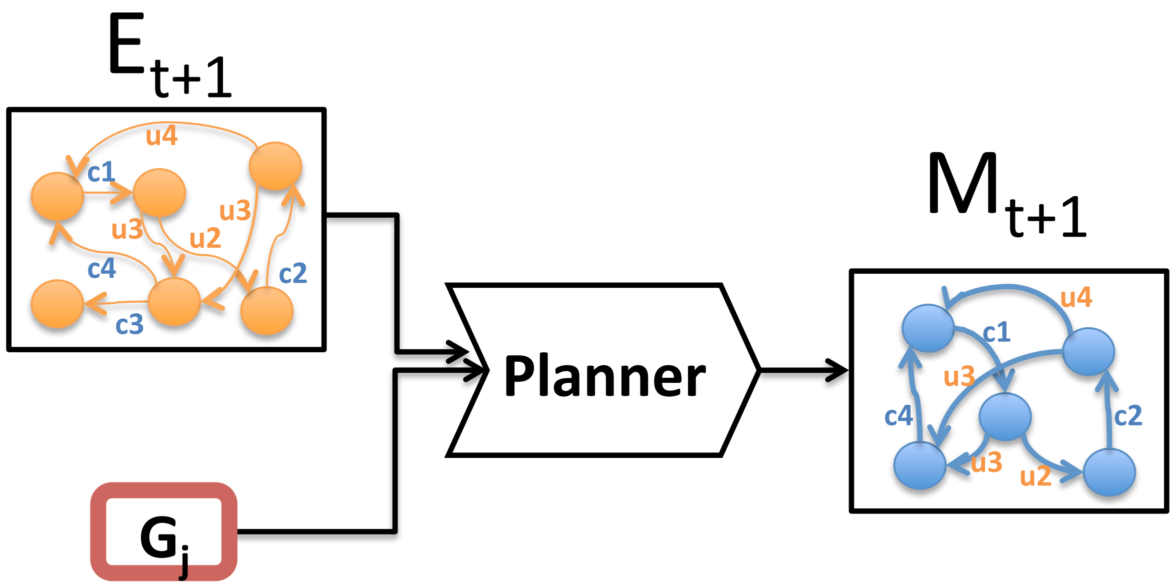
- Determine a functionality level $\boldsymbol{G_j}$ from $\{\boldsymbol{G_1},\ldots\boldsymbol{G_N}\}$
  - $\boldsymbol{G_j}$ can be satisfied in $\boldsymbol{E_{t+1}}$
  - The system can degrade to $\boldsymbol{G_j}$ without stopping or restarting itself
- Functionality level selection
  - (will be explained later)

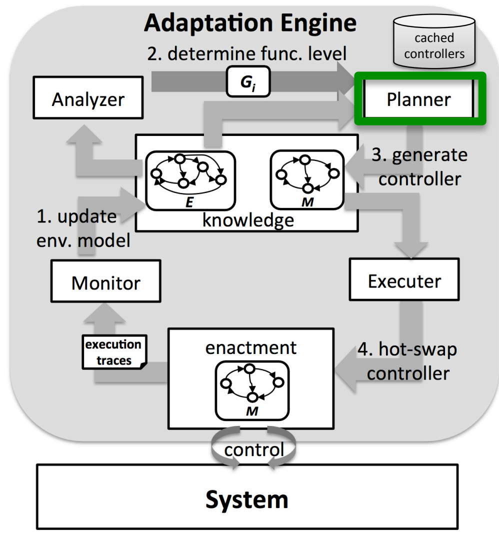




3. Plan: Discrete Controller Synthesis

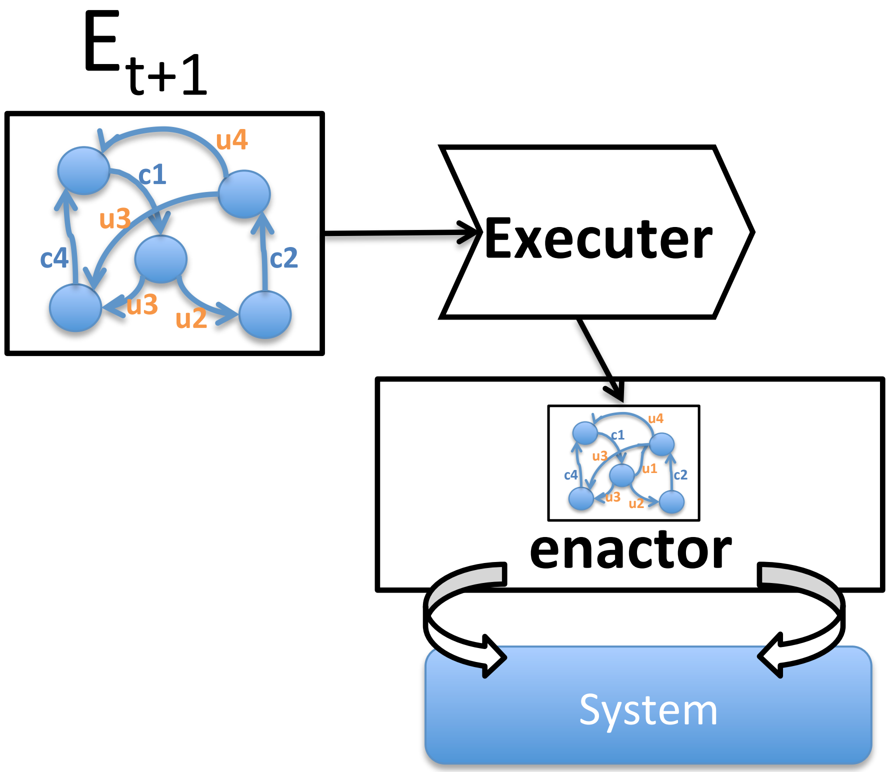
- Generate an LTS $\boldsymbol{M_{t+1}}$ guaranteeing satisfaction of $\boldsymbol{G_j}$ in $\boldsymbol{E_{t+1}}$
- ***Discrete controller synthesis*** [D'Ippolito, 2010] [D'Ippolito, 2011]
  - solve a control problem ***<E,G>*** to find an LTS ***M***

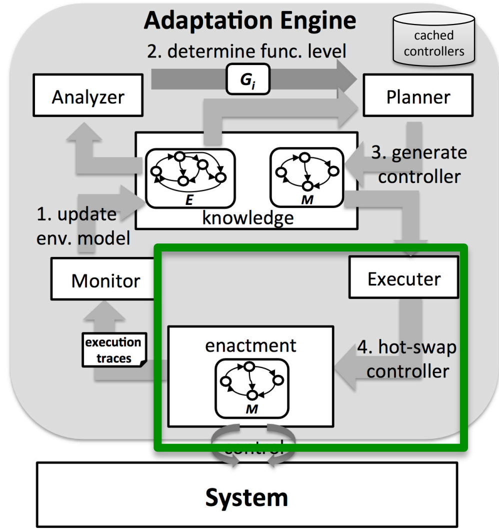




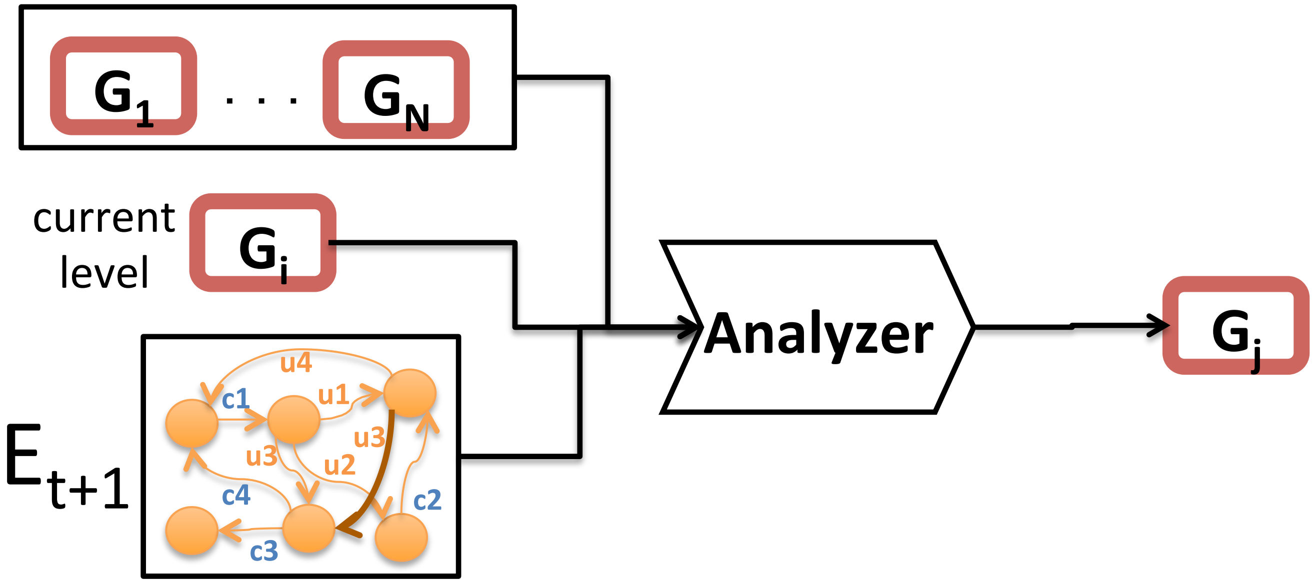
4. Execute: Enactor

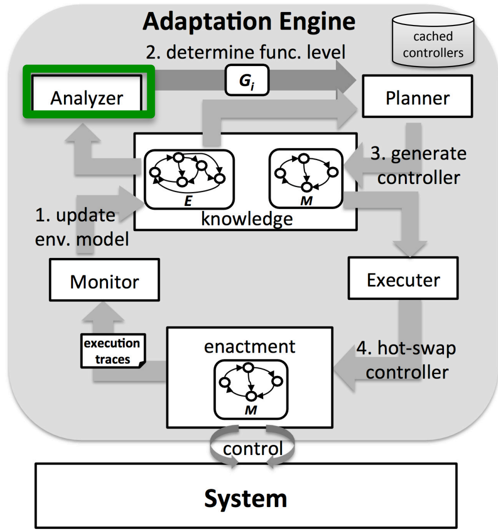
- Hot-swap controller model from M_t to M_{t+1}
  - It can be done without stopping the system because M_{t+1} simulates M_t
- ***Enactment framework*** [Braberman, 2013]
  - interpret LTS and orchestrate high-level operations provided by the system





Q: How can the system determine how much it should degrade?





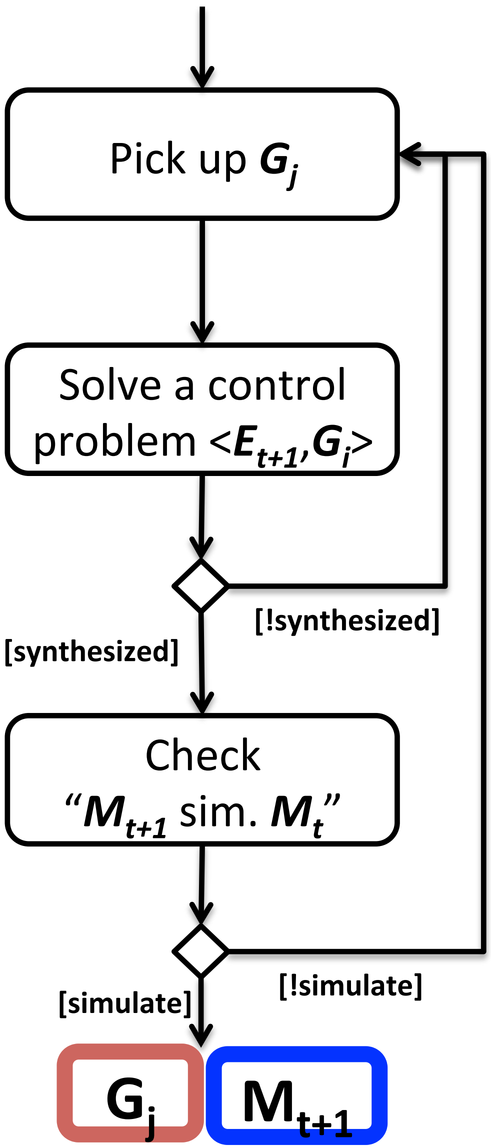
How does the system find the highest level G_j ?

M_{t+1} exists such that

(1) $M_{t+1} \,||\, E_{t+1} \models G_j$ **(provide assurance)**

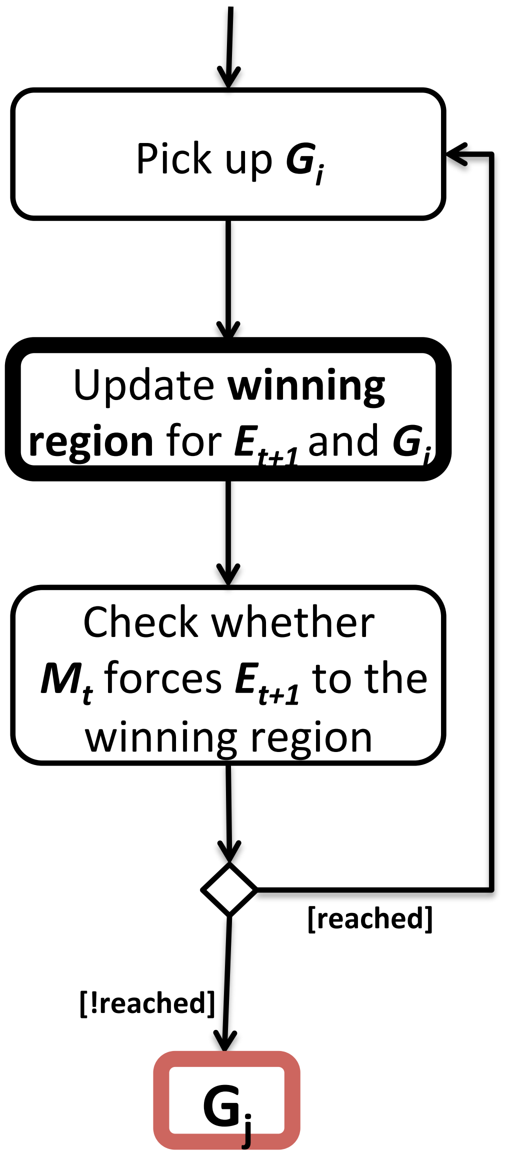
(2) M_{t+1} simulate M_t **(perform degradation seamlessly)**

Naive Strategy : Synthesize, then check



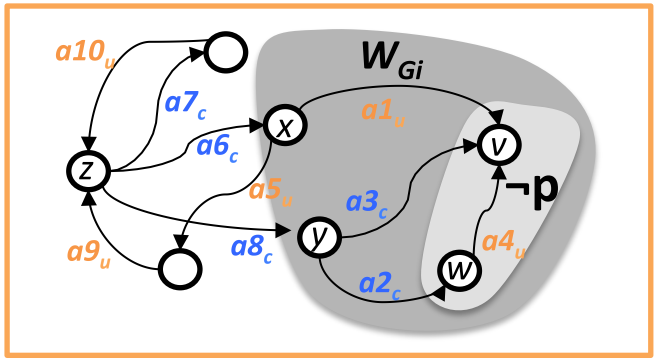
- Simple and straightforward
  - synthesized controller M_{t+1} can be used for the next controller
- Computationally inefficient
  - ***N*** control problems should be solved at worst

Advanced Strategy: Check without Synthesis



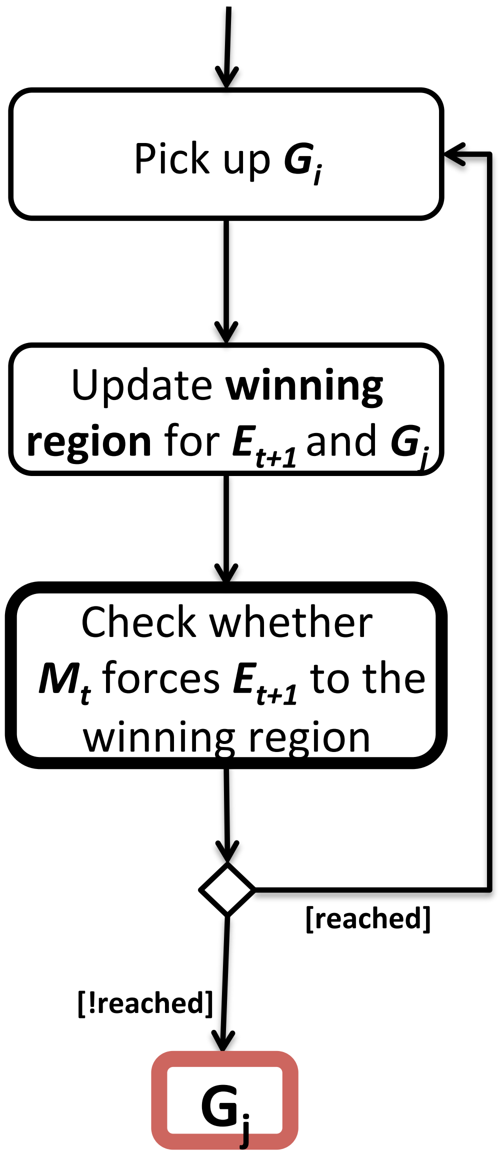
- **winning region $W_{Gi,Et}$**
  - the set of all states s such that no system forces E_t to satisfy G_i from s
  - controller strategy should avoid the winning region

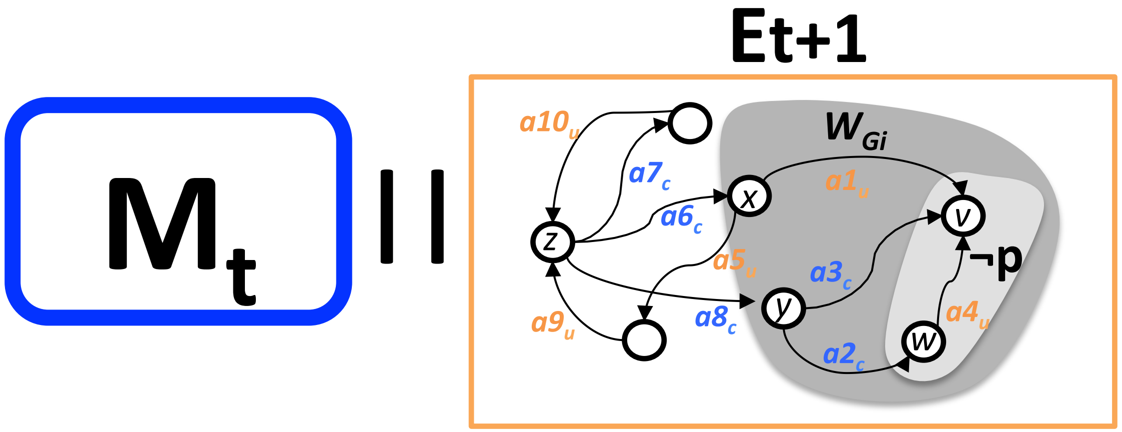
e.g. a winning region for $G_i = \square p$



- update **winning region**
  - $W_{Gi,Et+1}$ is obtained from $W_{Gi,Et}$ and Δ_{t+1}
  - determine states newly added in the region by checking updated part in env. model

Advanced Strategy: Check without Synthesis





- Does $M_t || E_{t+1}$ reach to $W_{Gi,Et+1}$?
  - if yes, M_{t+1} does not exist such that

$M_{t+1} || E_{t+1} \models G_j$

M_{t+1} simulate M_t

Case studies

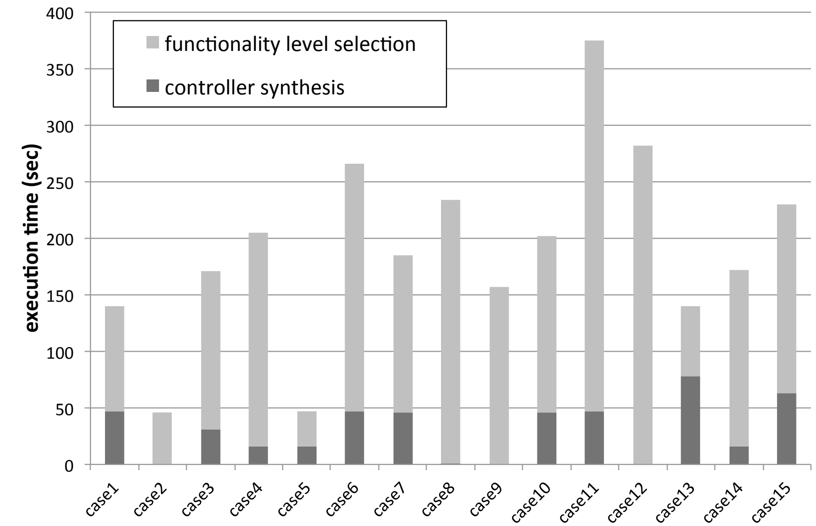
automated warehouse

production cell

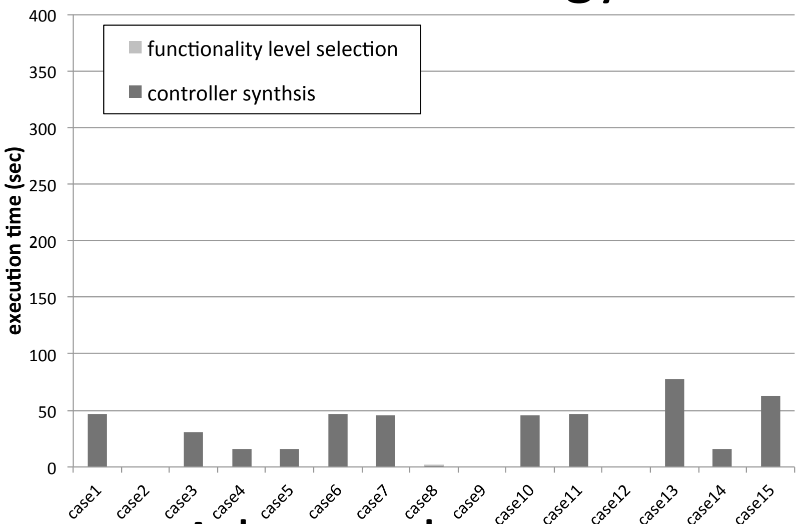
Automated Warehouse

Table 1: Case studies in the automated warehouse scenario

Case	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
before degradation	G_5	G_5	G_5	G_5	G_5	G_4	G_3	G_4	G_2	G_5	G_5	G_5	G_3	G_2	G_2
after degradation	G_5	G_4	G_3	G_2	G_4	G_3	G_1	G_2	G_1	G_4	G_2	G_3	G_3	G_2	G_1
# of levels checked	1	2	3	4	2	2	3	3	2	2	4	3	1	1	2

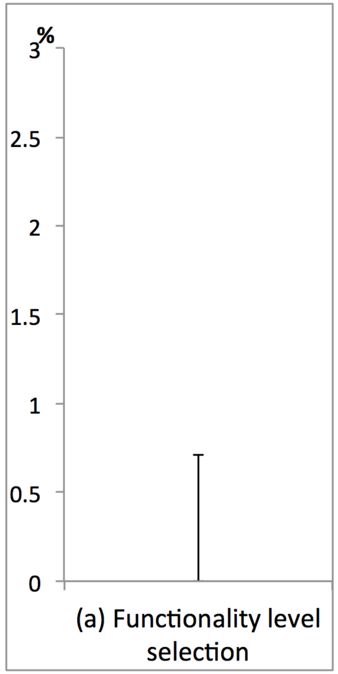


Naive strategy

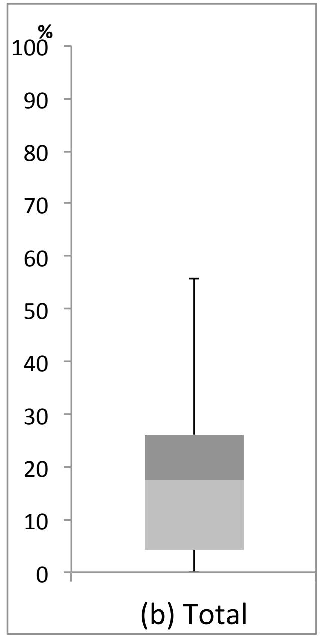


Advanced strategy

For func. selection



For func. selection + controller synthesis



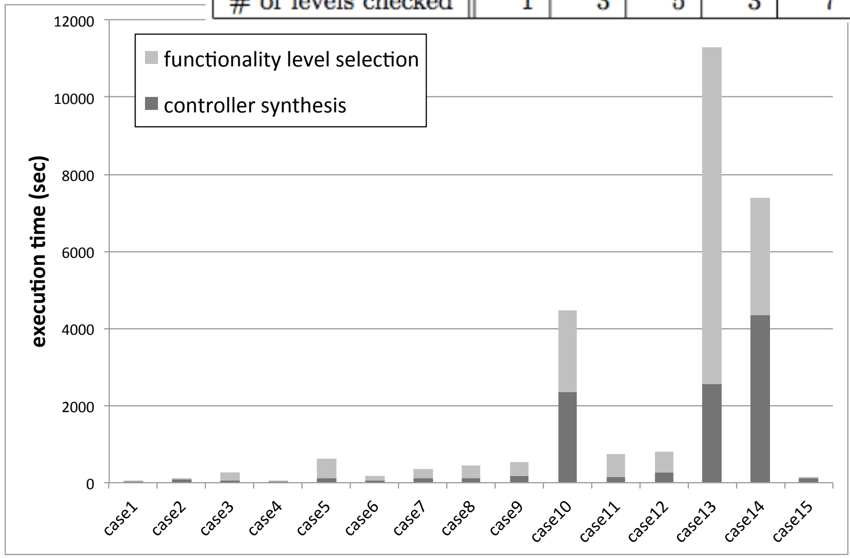
0.7% in the worst
0.00002% on average

35.8% in the worst
13.6% on average

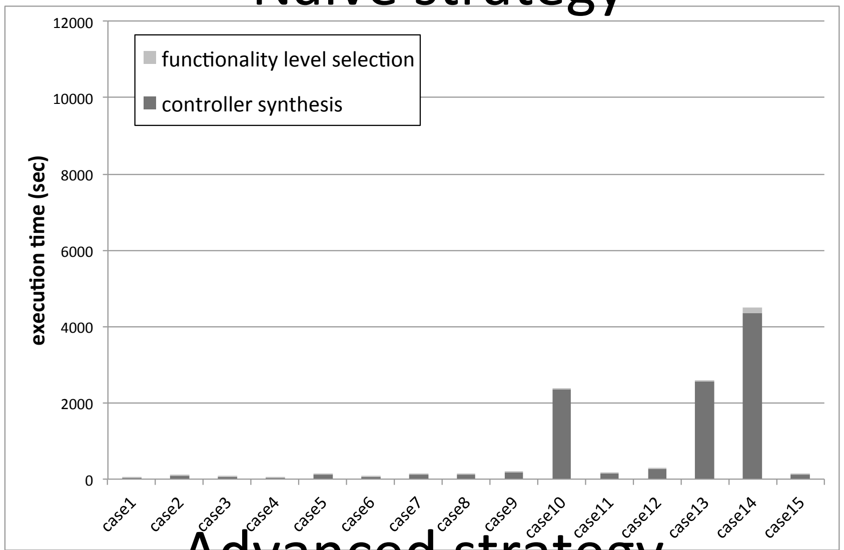
Production Cell

Table 2: Case studies in the production cell scenario

Case	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
before degradation	G_{12}	G_{12}	G_{12}	G_{12}	G_{11}	G_{12}	G_{10}	G_{11}	G_7	G_5	G_6	G_4	G_{11}	G_6	G_{11}
after degradation	G_{12}	G_{10}	G_8	G_{10}	G_5	G_9	G_7	G_6	G_4	G_3	G_2	G_1	G_6	G_4	G_{11}
# of levels checked	1	3	5	3	7	4	4	6	4	3	5	4	6	3	1

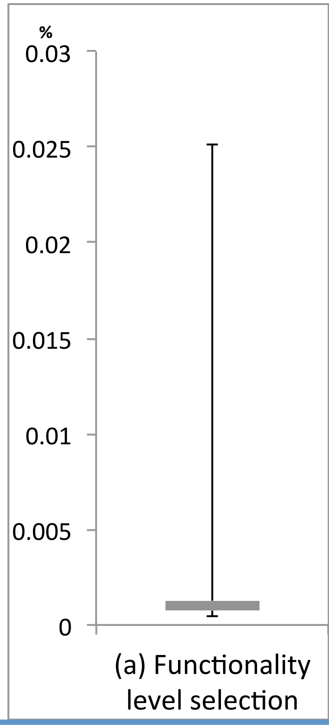


Naive strategy



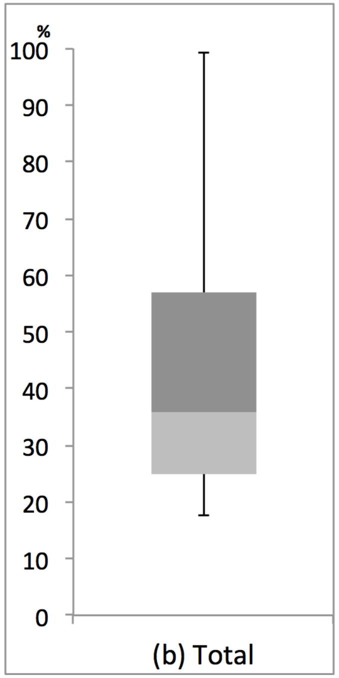
Advanced strategy

For func. selection



2.5% in the worst
0.176% on average

For func. selection + controller synthesis



99.2% in the worst
44.9% on average

Conclusion

- How does the system cope with development time uncertainty?
  - How do we select appropriate level of functionality considering risks and functionality?
- We propose a framework enabling graceful degradation
  - revise environment model @ runtime
  - generate behavior specification with assurance @ runtime
  - change behavior of the system @ runtime
- We introduce two strategies to find the highest level of functionality that can be guarantee and to which the system can seamlessly degrade