

A detailed close-up photograph of a mechanical watch movement, showing various gears, jewels, and metal components. A semi-transparent black rectangular box is overlaid on the right side of the image, containing white and gold text.

# MTSA講座

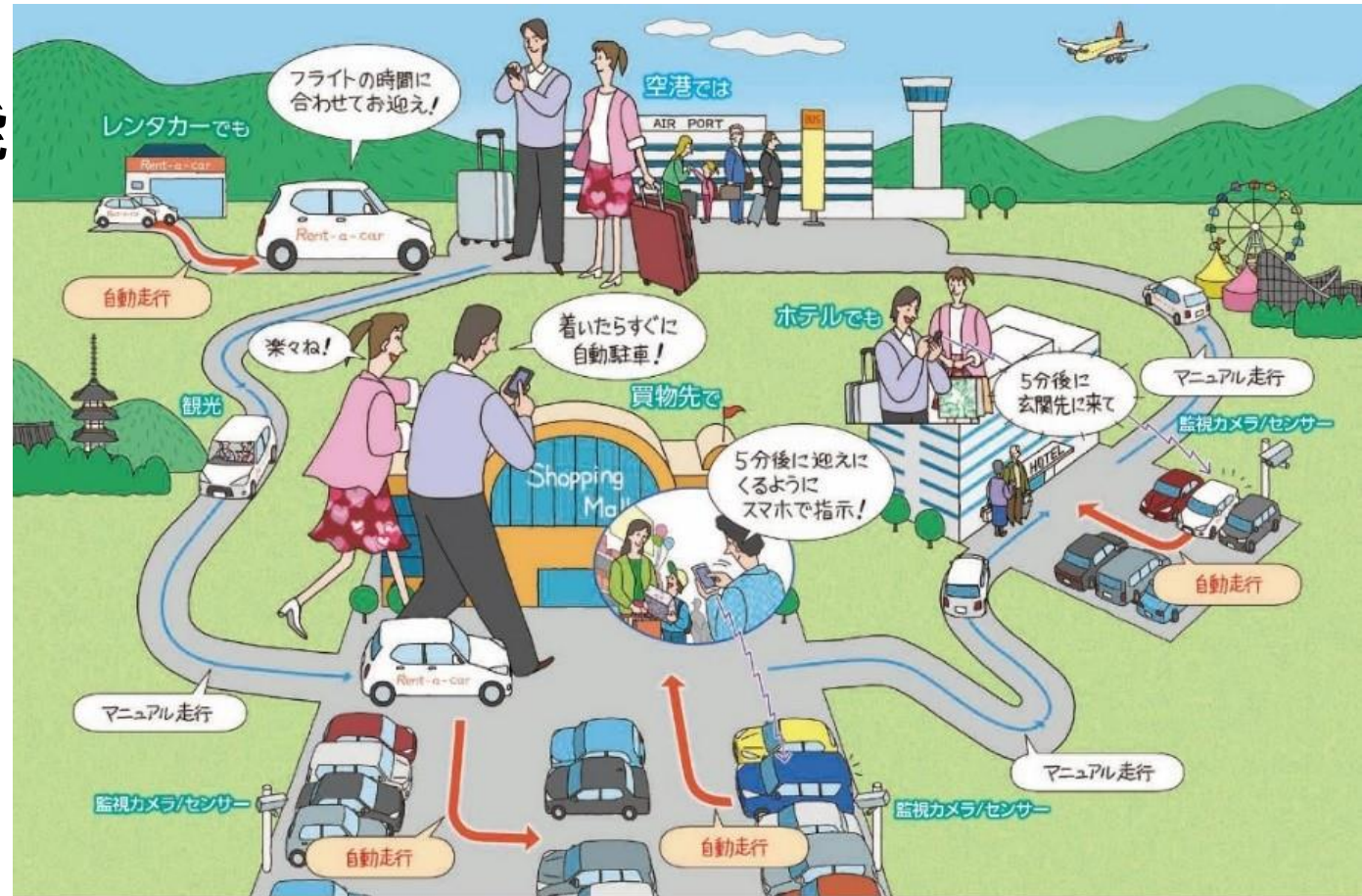
第2回

# 目的

- ・ 実際に存在する(あるいはしそうな)問題にMTSAを適用してみよう
- ・ 適用する過程で必要な次のことを一例を通して感じてみよう
  - ・ モデリングの仕方・注意点
  - ・ 洗い出した要求を論理式にする方法
- ・ 適用するに当たってぶつかりがちな問題点を見てみよう

# 自動バレーパーキング

- 自動運転が進めば  
自動バレーパーキングが可能
  - 入り口で降りて自動駐車
  - 出口で呼べば自動で来る
- 駐車場内の車の走行を管制するコントローラが必要
  - これをMTSAで作ろう

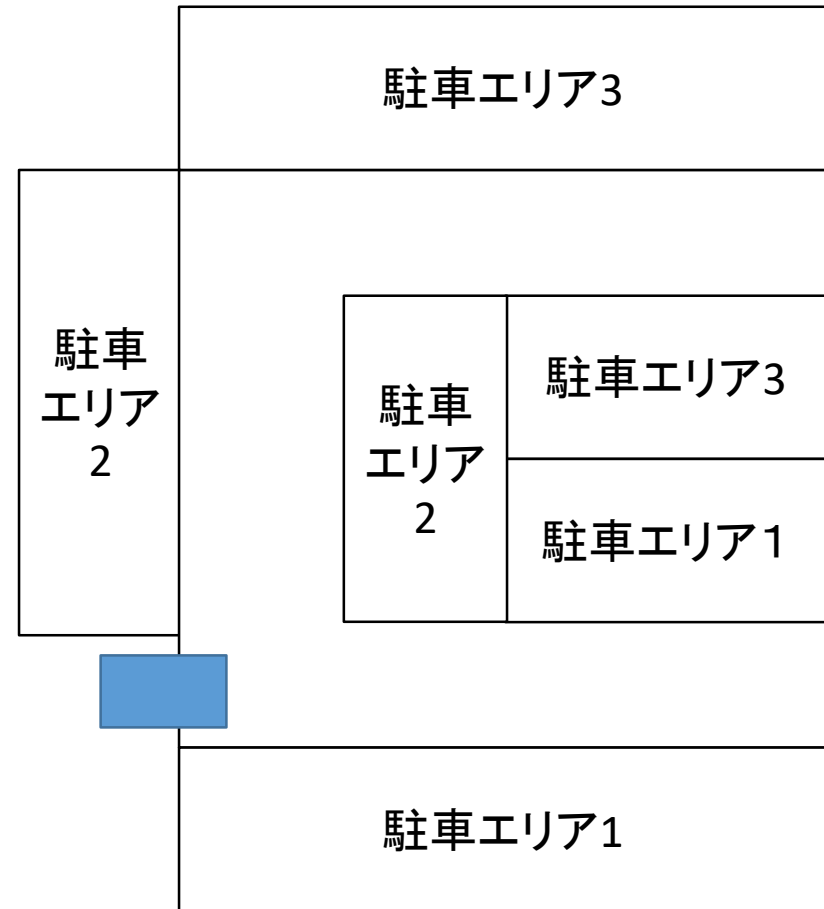




# 駐車場をモデル化しよう

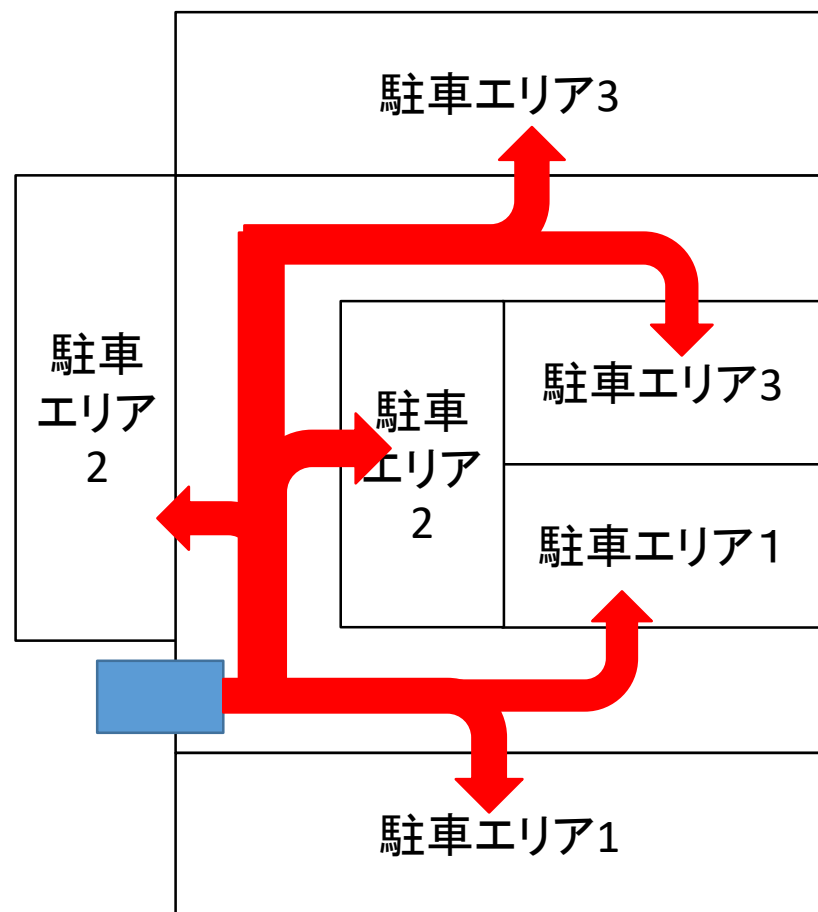
## [本年度卒論テーマ(予定)]

- 例えば右のような駐車場を考える
  - 入口と出口は同じ場所
- コの字型の通路を囲んで駐車エリアが存在
- 1通路には(とりえあえず)最大1台が入る想定

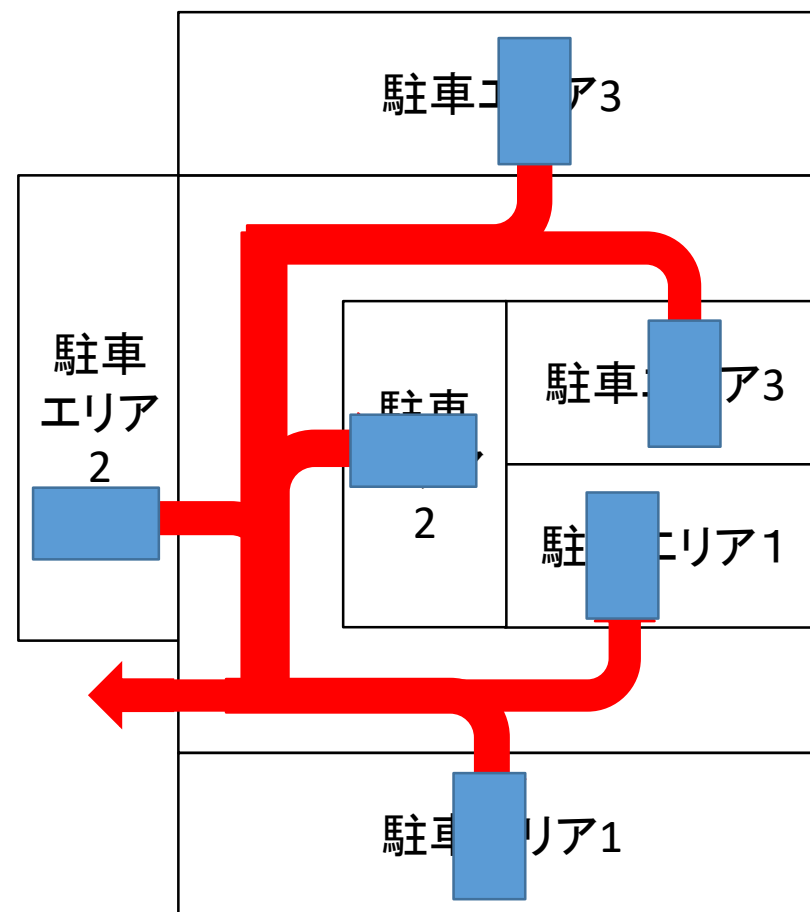


# 導線

・ゲートから入る時



・出る時



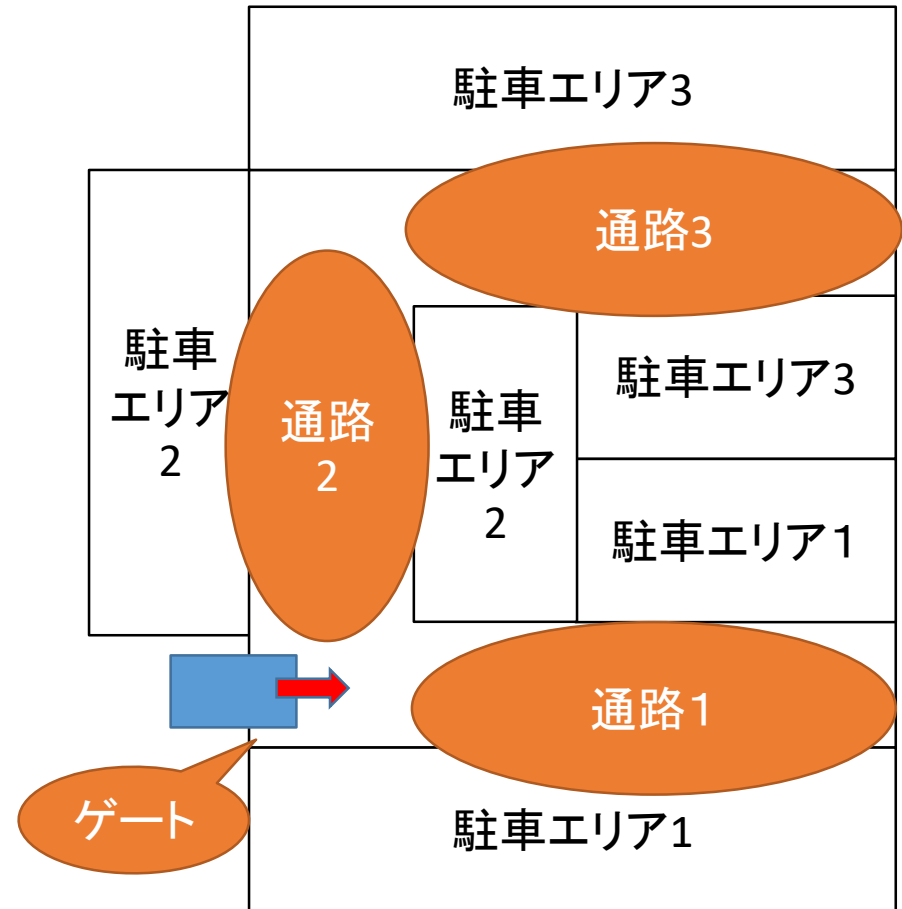
# どうモデリングするか？

- ・ パーツに分ける

- ・ 駐車スペース ...駐車台数分
- ・ 通路 ...3つ
- ・ ゲート ...1つ

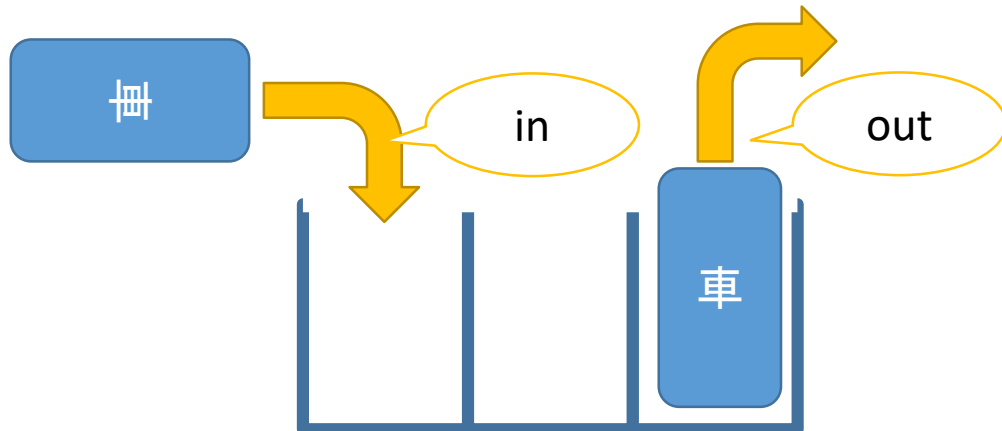
- ・ パーツ同士の繋がりを確認

- ・ ゲート ↔ 通路1, 通路2
- ・ 通路1 ↔ 通路2
- ・ 通路2 ↔ 通路3
- ・ 各通路 ↔ 各駐車場



# パーツのモデリング①

- ・ 駐車スペース
  - ・ 1か所ずつモデル化
  - ・ 状態:
    - ・ EMPTY/FULL
  - ・ アクション:
    - ・ 入庫(in)/出庫(out)  
/出庫依頼(request)  
(細かい制御は考えない)



**PARK=EMPTY,**  
**EMPTY=(in->FULL),**  
**FULL=(request->out->EMPTY).**

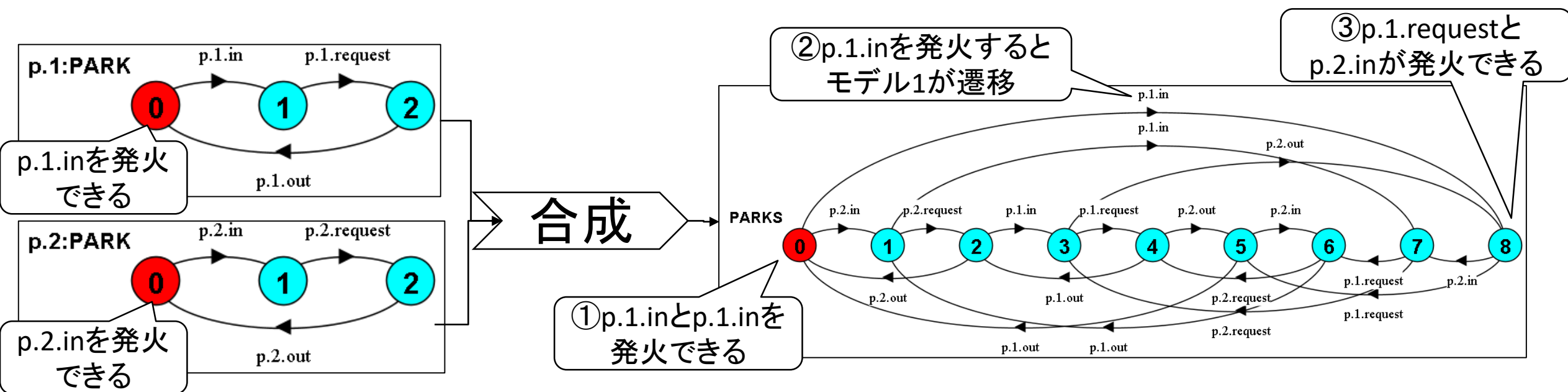
駐車スペース1台分のモデル

**|| PARKS=(p[**NUM\_OF\_SLOT**]:PARK).**

駐車台数分を並列合成

# 並列合成(その1)

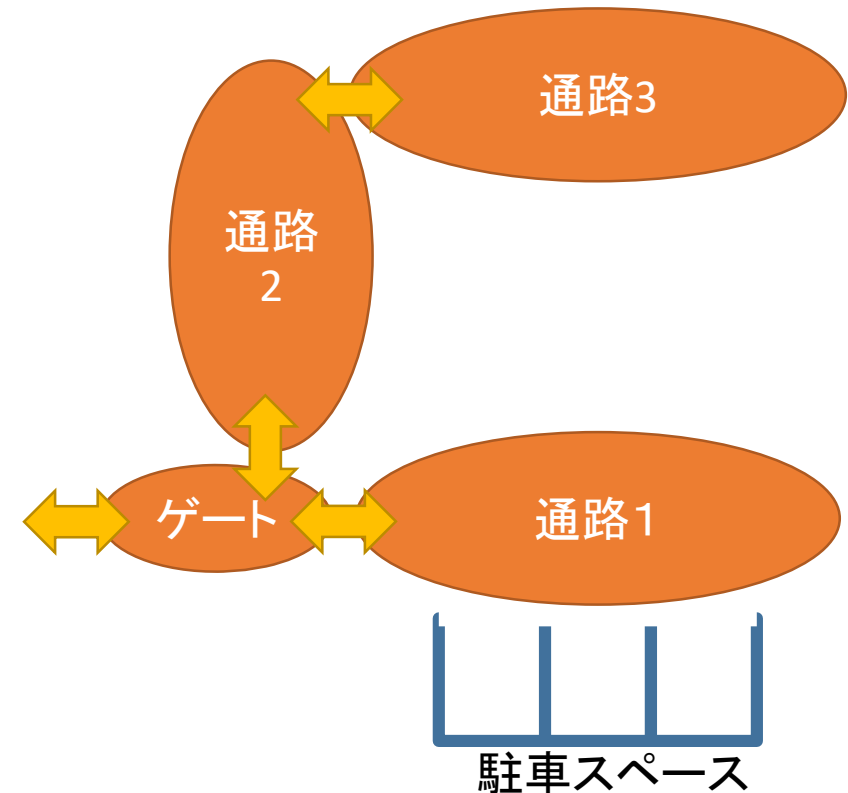
- 複数のモデルを1つのモデルにまとめる技術
  - 非同期アクションは任意の順番で発火できる
  - 同期アクションは全てのモデル上で発火可能になったときだけ発火できる(後述)





# パーツのモデリング②

- 通路とゲート
  - それぞれには最大で1台入れる
  - 各通路から対応する駐車場に入れる
  - 状態:
    - EMPTY/FULL
  - アクション
    - それぞれの通路に入るアクション
    - 各駐車スペースへの入庫アクション



# パーツのモデリング②

- 取り敢えず簡単な例を作ろう
  - ゲート × 1, 通路 × 1, 駐車スペース × 2

```
const N=2 range R=1..N
```

```
PARK=EMPTY,
```

```
EMPTY=(in->FULL), FULL=(request->out->EMPTY).
```

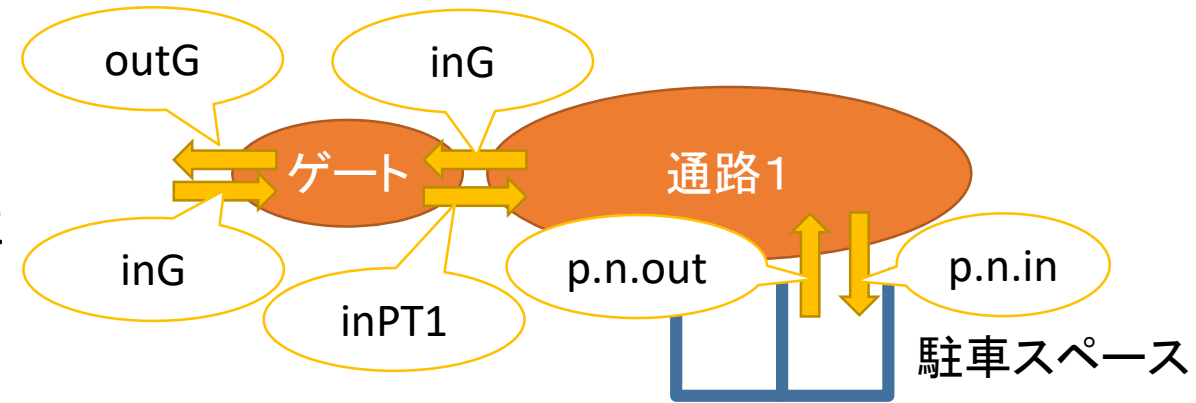
```
GATE=EMPTY,
```

```
EMPTY=(inG->FULL), FULL=({outG, inPT1} -> EMPTY).
```

```
Path=EMPTY,
```

```
EMPTY=({p[R]. outPK} -> FULL), FULL=({p[R]. inPK, inG} -> EMPTY).
```

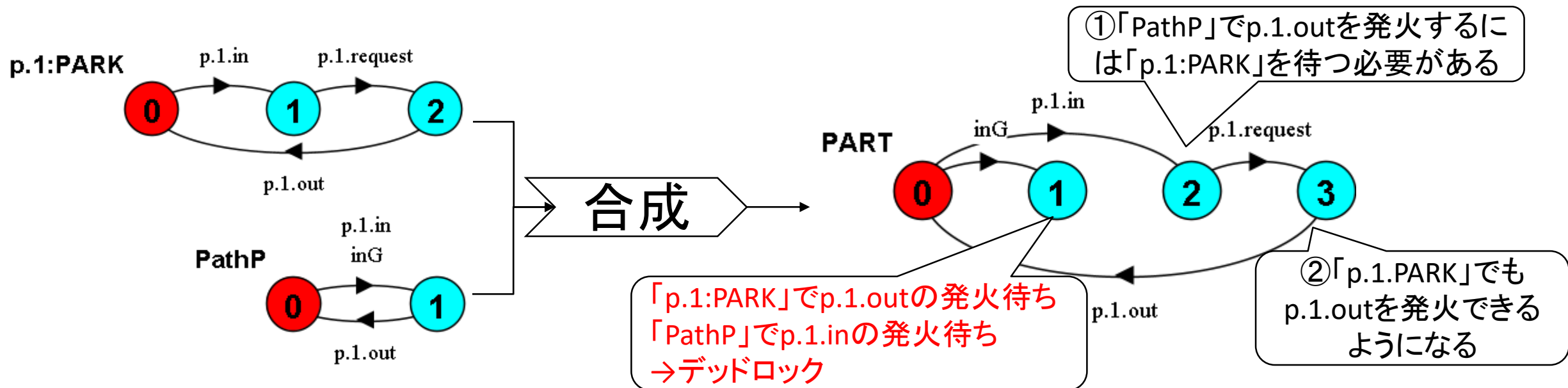
```
||ENV=(p[R]: PARK || GATE || Path).
```



# 並列合成(その2)

- 同期アクションとその合成

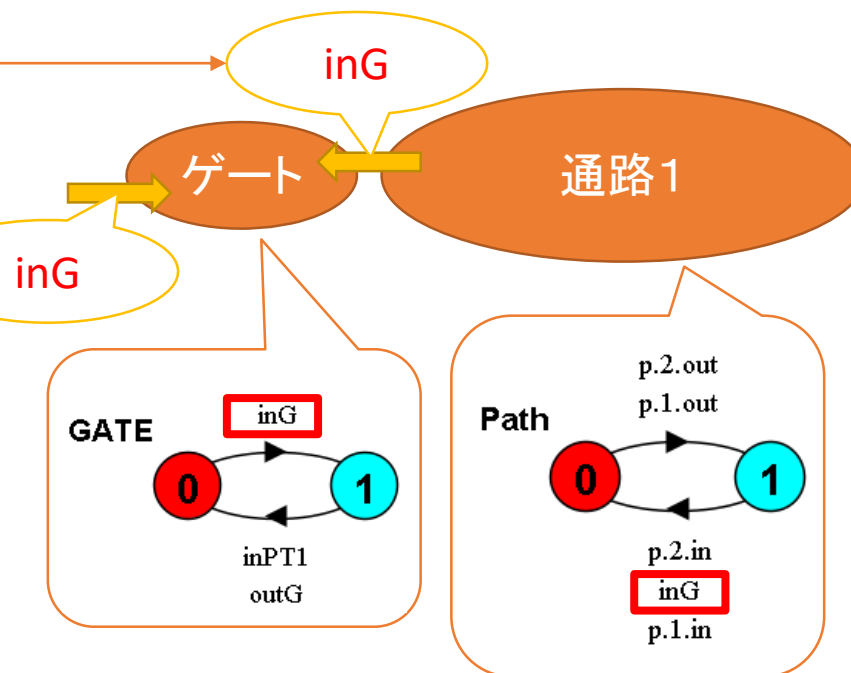
- 同期アクション : 二つのモデル間で共有されるアクション
- 並列合成時の扱い : 二つのモデル上で発火可能になったとき発火可能
- デッドロック : あるモデルで発火



# 同期アクションのワナ

- さっき作った駐車場のモデルを並列合成するとデッドロックする

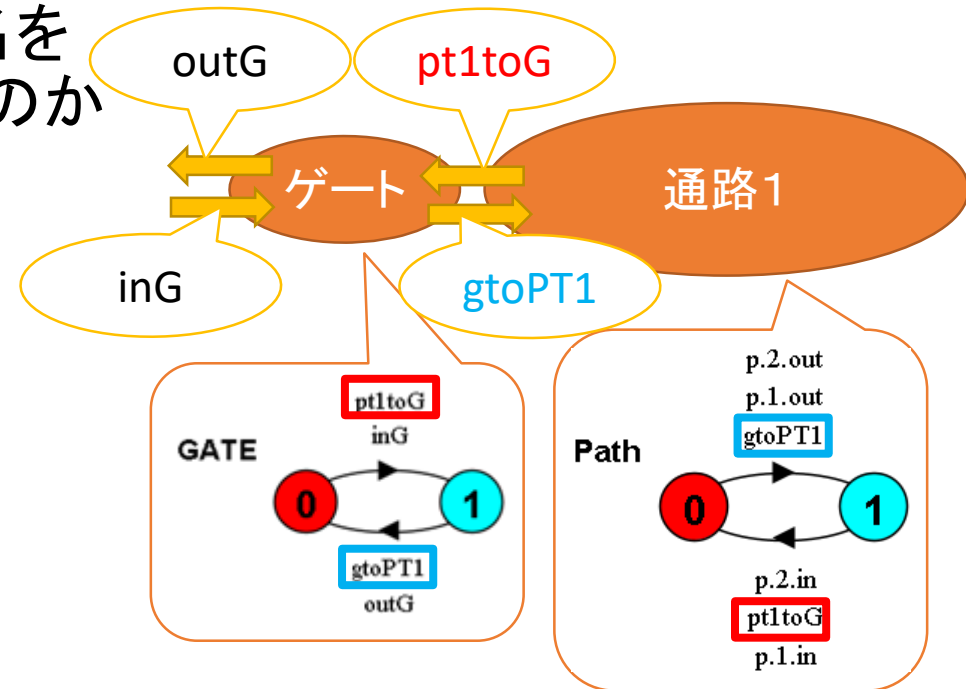
どちらもゲートに入るアクション  
だがどこから入るかが異なる  
→同期されてはいけない



- 同期されてはいけないアクションは  
違う名前にしなければならない！

# 同期すべきでないアクションの名前分け

- 同期すべきでないアクションの名前は分ける
  - 複数のモデル間で同じアクション名を使う時はそれが同期されて良いものか十分に留意する



# 並列合成による環境モデルの生成 (簡単な例のまま続けます)

- ・ 駐車スペース(PARK)、ゲート(GATE)、通路(PATH)を並列合成

```
const N=2 range R=1..N
```

```
PARK=EMPTY,
```

```
EMPTY=(in->FULL), FULL=(request->out->EMPTY).
```

```
GATE=EMPTY,
```

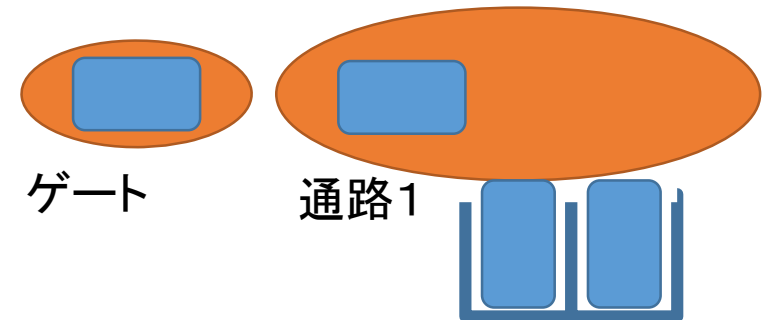
```
EMPTY=({inG, pt1toG}->FULL), FULL=({outG, gtoPT1} -> EMPTY).
```

```
PATH=EMPTY,
```

```
EMPTY=({p[R]. out, gtoPT1}->FULL), FULL=({p[R]. in, pt1toG}->EMPTY).
```

```
||ENV=(p[R]:PARK||GATE||PATH).
```

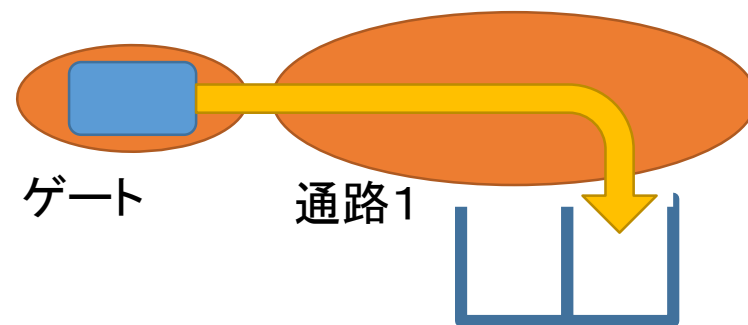
これは飽くまで環境モデル  
なので下のような状況に陥る



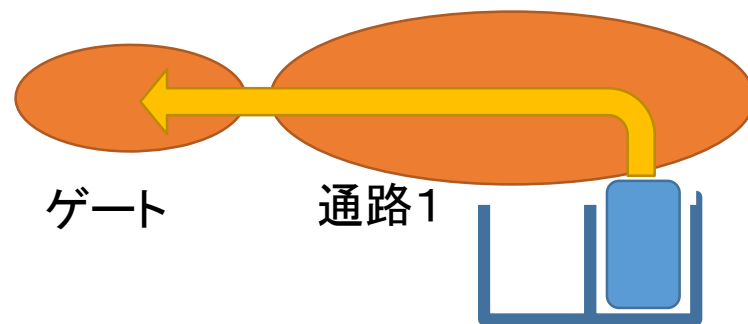


# 駐車場の要求を考える (起こって欲しいことを考える)

1. ゲートから入って来た車は必ず  
いつかは駐車スペースに辿り着く



2. 駐車スペースから出た車は必ず  
いつかはゲートに辿り着く

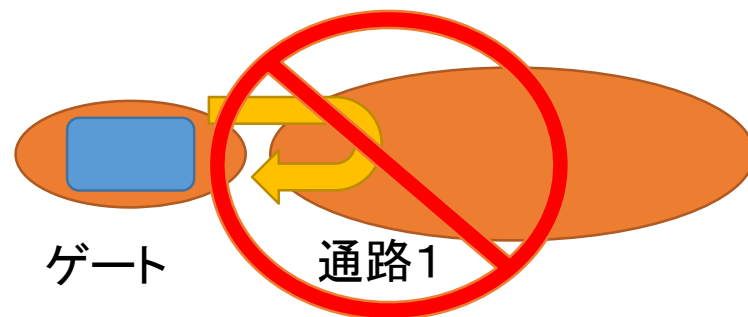


# 駐車場の要求を考える (起きてはいけないことを考える)

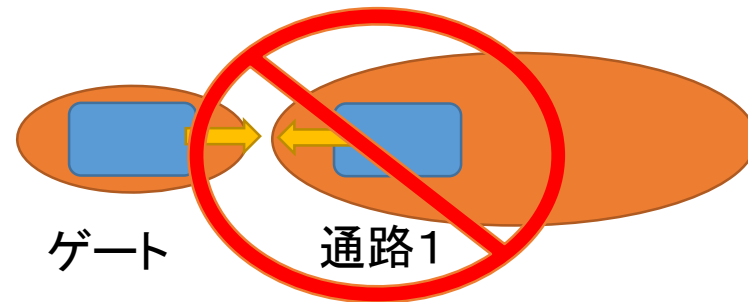
3. 少なくとも右の図のような状態は  
起きてはいけない



4. 同じ空間を何度も行き来する  
というのもしるべきではない

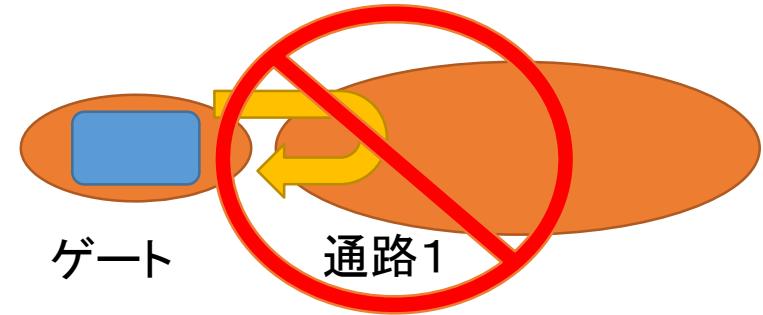


5. 入る車と出る車がかち合う  
ということも起きてはいけない



# 要求を論理式に落とし込む (実はこの例では5さえ満たせば良い)

5. 同じ空間を何度も行き来する  
というのも起きるべきではない



- ・ 時相論理で使える演算子を意識して翻訳しよう

- ・ 「ゲートから通路1に入ったならば、その後は駐車スペースに入るまでゲートに戻らない」

$\rightarrow$  (implicit)

$X$  (next)

$W$  (weak until)



$!$  (negation)

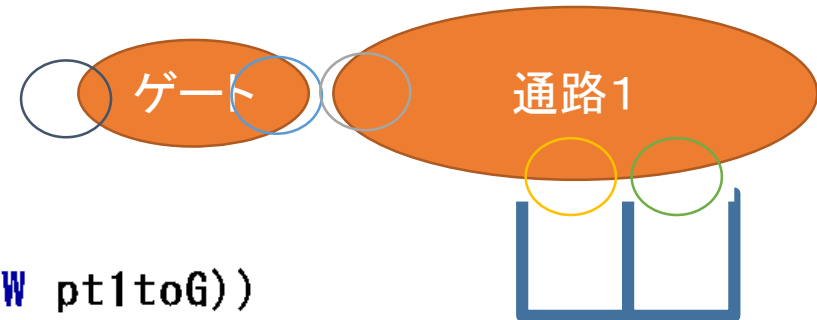
```
ltl_property NotReturnPT1=[] (gtoPT1 $\rightarrow$  $X$ (!pt1toG  $W$  (p[1]. in || p[2]. in)))
```

他の場所も考えてみよう！

# 要求を論理式に落とし込む

- パーツ間に着目して要求を考える

`set Requests={p[R]. request}`



○ `ltl_property NotReturnPK1=[] (p[1]. out->X(!p[1]. in W pt1toG))`

○ `ltl_property NotReturnPK2=[] (p[2]. out->X(!p[2]. in W pt1toG))`

○ `ltl_property NotReturnTheGate=[] (inG->X(!outG W gtoPT1))`

○ `ltl_property NotReturnPT1=[] (gtoPT1->X(!pt1toG W (p[1]. in||p[2]. in)))`

○ `ltl_property NotReturnG=[] (pt1toG->X(!gtoPT1 W (outG)))`

- 出庫リクエストが来るまで出庫しない

`ltl_property OutWithRequest=((!outG W {Requests})&&[] (outG-> X(!outG W {Requests})))`

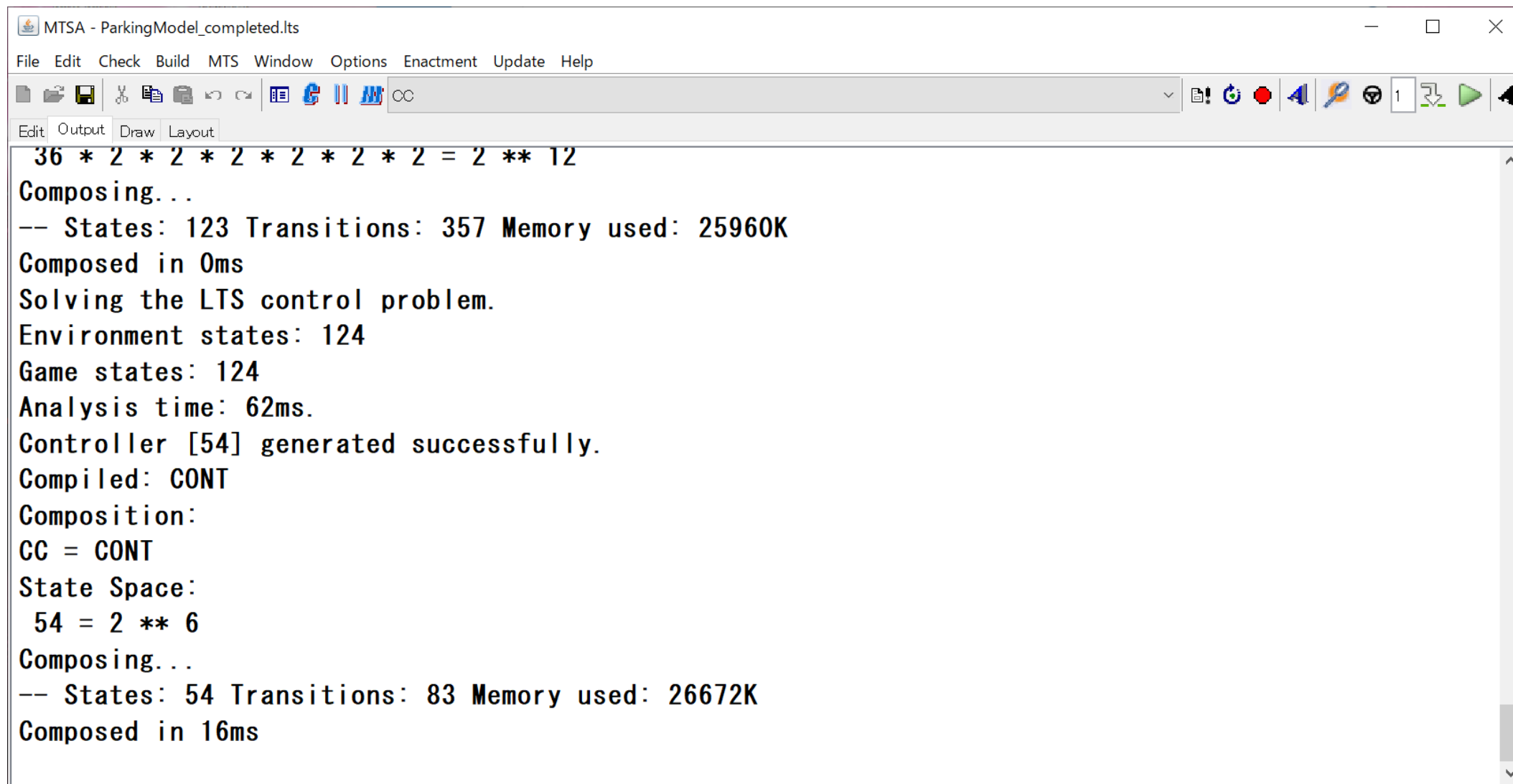
# コントローラを合成しよう

- 制御可能なアクションの集合を準備しよう
  - request以外は制御可能
- 要求と制御可能なアクションでcontrollerSpecを定義
- controllerSpecと環境モデルからコントローラを合成

```
set ControllableAction = {p[R]. in, p[R]. out, gtoPT1, pt1toG, inG, outG}
controllerSpec C={
  safety={NotReturnPK1, NotReturnPK2,
          OutWithRequest, NotReturnTheGate, NotReturnPT1, NotReturnG/**/}
  controllable =ControllableAction
}

controller ||CONT=(ENV) ~ {C}.
```

# できた？



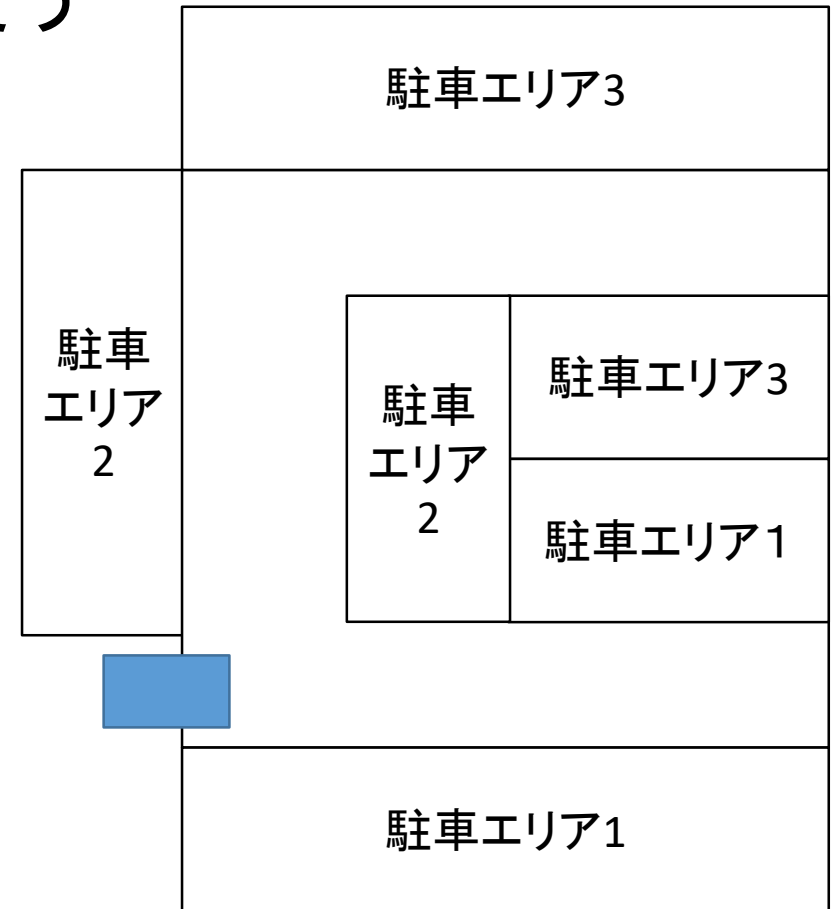
The screenshot shows a software window titled "MTSA - ParkingModel\_completed.lts". The window has a menu bar with "File", "Edit", "Check", "Build", "MTS", "Window", "Options", "Enactment", "Update", and "Help". Below the menu bar is a toolbar with various icons. The main area of the window displays the following text:

```
36 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = 2 ** 12
Composing...
-- States: 123 Transitions: 357 Memory used: 25960K
Composed in 0ms
Solving the LTS control problem.
Environment states: 124
Game states: 124
Analysis time: 62ms.
Controller [54] generated successfully.
Compiled: CONT
Composition:
CC = CONT
State Space:
  54 = 2 ** 6
Composing...
-- States: 54 Transitions: 83 Memory used: 26672K
Composed in 16ms
```



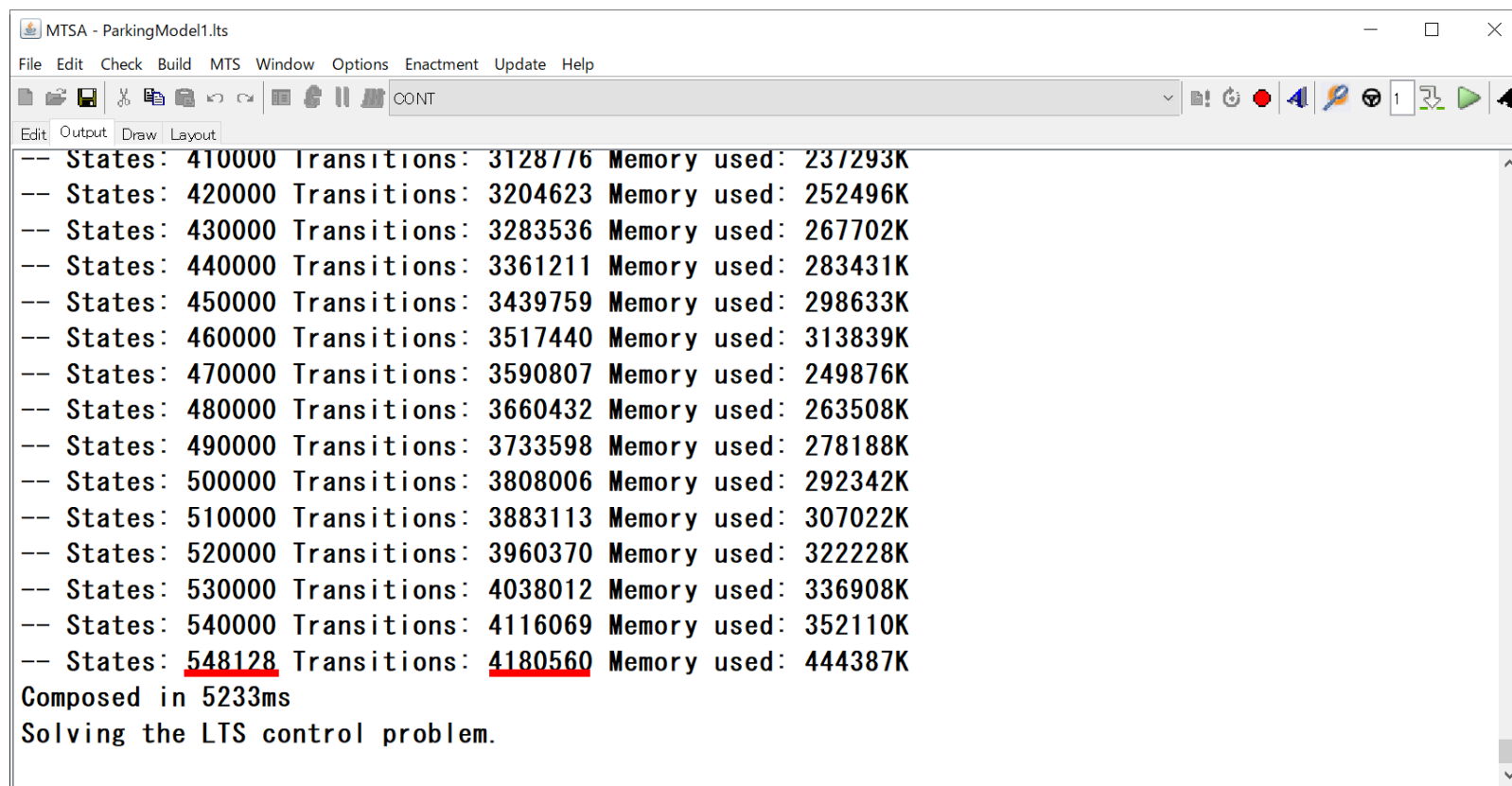
# 本題の駐車場もモデル化してみよう

- ・ 駐車エリアに停まれる数は徐々に増やそう
  - ・ 先ずは1台ずつ(合計3台)
  - ・ そこから2台...3台と増やす
- ・ 通路を増やそう
  - ・ 通路と駐車エリアとの繋がり方は同じ
  - ・ 駐車エリア2との繋げ方にだけ注意
- ・ 要求も考えてみよう



# やってみると...

- 簡単な例では状態数が124だったのにこっちは54万(遷移は418万)
- スペックの高いPCじゃないと合成出来ない



The screenshot shows a software window titled "MTSA - ParkingModel1.lts". The window has a menu bar with "File", "Edit", "Check", "Build", "MTS", "Window", "Options", "Enactment", "Update", and "Help". Below the menu bar is a toolbar with various icons. The main area of the window displays a list of states and transitions, with the last line highlighted in red. The text in the window is as follows:

```
-- States: 410000 Transitions: 3128776 Memory used: 237293K
-- States: 420000 Transitions: 3204623 Memory used: 252496K
-- States: 430000 Transitions: 3283536 Memory used: 267702K
-- States: 440000 Transitions: 3361211 Memory used: 283431K
-- States: 450000 Transitions: 3439759 Memory used: 298633K
-- States: 460000 Transitions: 3517440 Memory used: 313839K
-- States: 470000 Transitions: 3590807 Memory used: 249876K
-- States: 480000 Transitions: 3660432 Memory used: 263508K
-- States: 490000 Transitions: 3733598 Memory used: 278188K
-- States: 500000 Transitions: 3808006 Memory used: 292342K
-- States: 510000 Transitions: 3883113 Memory used: 307022K
-- States: 520000 Transitions: 3960370 Memory used: 322228K
-- States: 530000 Transitions: 4038012 Memory used: 336908K
-- States: 540000 Transitions: 4116069 Memory used: 352110K
-- States: 548128 Transitions: 4180560 Memory used: 444387K
Composed in 5233ms
Solving the LTS control problem.
```

# MTSAの天敵：状態爆発

- 今回のようなモデリング方法の欠点（問題）
  - 駐車エリア内の空き状況を1つ1つモデル化
  - どこが空いていてどこが埋まっているのかの組み合わせは $2^{\text{駐車台数}}$ 通り
  - 駐車スペースを1つ増やすごとに状態数は何倍にも膨らんでいく（状態爆発）
- この問題をどう解決したらいいだろう？（研究の足掛かり）
  - 駐車スペースを区別する必要は本当にあるのか？（抽象化）
  - 駐車スペースの管理と通路の管理でコントローラを分けられないか？（分割）

# 今回扱わなかった話

- ・ 今回のモデルはとてもシンプル
- ・ もっと特殊なケースも考えられる
  - ・ 車の出庫が途中でキャンセルされたときは？
  - ・ 車が指示した場所と異なる場所に辿り着いたときは？
  - ・ 自動運転だけでなくマニュアル運転車も共存するとどうなる？

ただでさえ状態爆発が起きるのにこんな特殊ケースはどう扱う？

# 次回（やるとしたらどっち？）

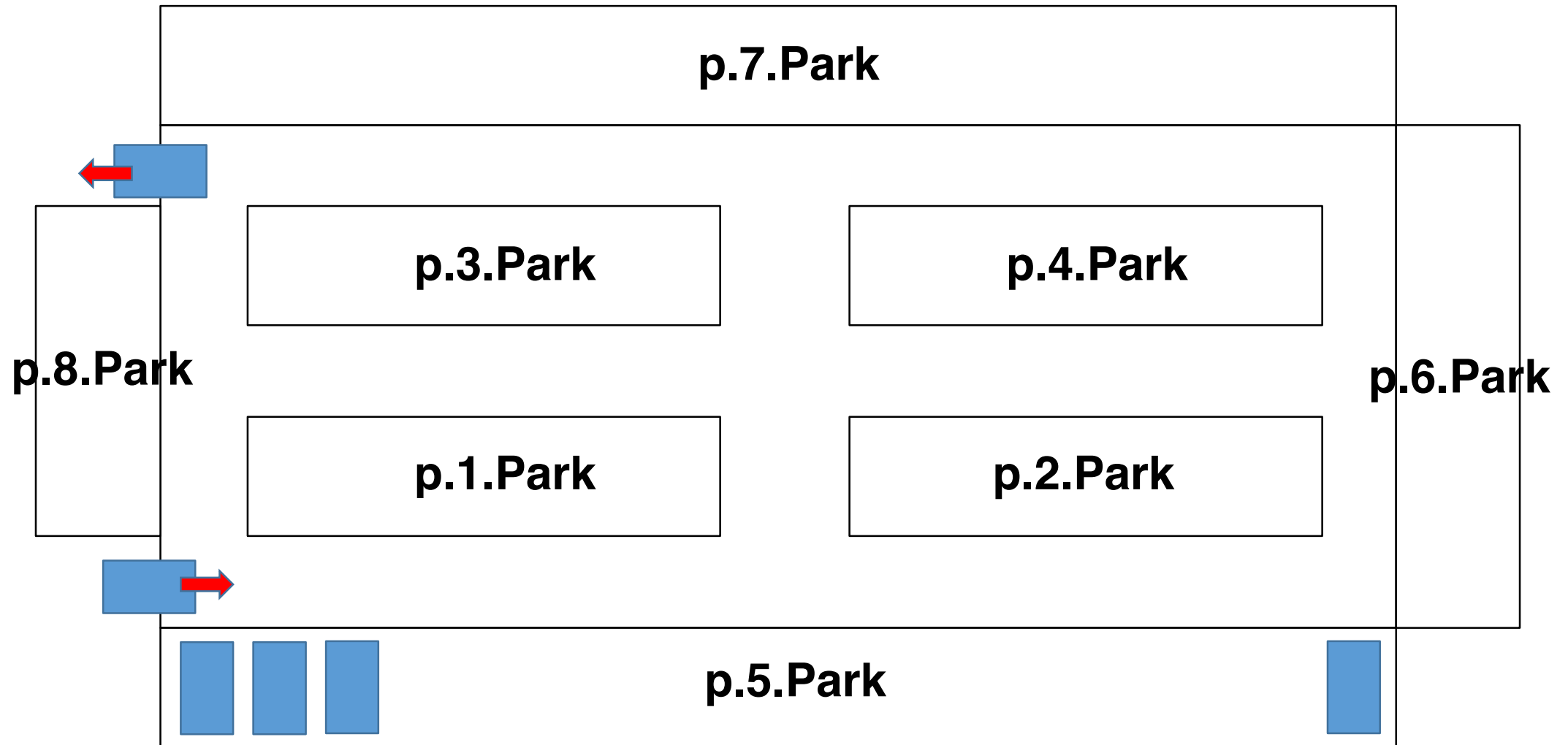
## 1. MTSAの中の理論とその研究例を見てみよう（研究）

- ゲームを作る・ゲームを解く
  - 到達可能性ゲーム
  - Buchiゲーム
  - Generalized Buchiゲーム
  - Generalized Reactivity(1)
- 目的に合わせてゲームを改造する、改造したゲームの解き方を考える

## 2. MTSAで実際のシステムを動かしてみよう（開発）

- オープンハウスの例（自動走行ロボット）
- みんなが作ったシステムを繋げるにはどうすれば良いか

# 本日の改修：以下のように修正





# 改修内容

- 駐車スペースを 8 個に増やす
  - pathは12個
  - どのパスからどの駐車場に止めれるようにするかよく考える
  - （余裕があれば）各駐車場に複数台止めれるようにする
- Gateを 2 つに増やす
  - 両方から入退室できる
  - （余裕があれば）片方は入室専用，片方は退室専用