Embedded Systems Specification and Design
Model-based Design and Verification

David Kendall

UPPAAL Language Extensions

Urgent and broadcast channels

Urgent channel

- `urgent chan c;`
- No time is allowed to pass if a synchronization on an urgent channel is enabled
- Clock guards are not allowed on edges with a synchronization on an urgent channel

Broadcast channel

- `broadcast chan c;`
- One sender, `c!`, synchronizes with an arbitrary number of receivers, `c?`.
- Any receiver that can synchronize, must do so
- If there are no receivers, the sender can still execute the `c!` action, i.e. broadcast sending is never blocking

UPPAAL Language Extensions

Urgent and committed locations

Urgent location

- Time is not allowed to pass if an urgent location is part of the current system state
- Equivalent to having an invariant `x <= 0` on the location, where the clock `x` is reset on every incoming edge

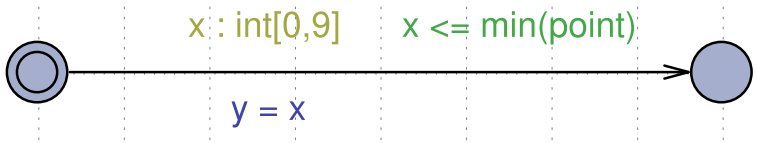
Committed location

- Like an urgent location in that time is not allowed to pass if a committed location is part of the current system state
- In addition, the next transition must involve an outgoing edge of at least one committed location

UPPAAL Language Extensions - Select

Select

- A select label contains a comma-separated list of *name : type* expressions, where *name* is a variable name and *type* is a defined type
- The value of *name* is chosen non-deterministically from the values of *type*
- The value of *name* is only available on the edge containing the select label



UPPAAL Language Extensions - Data

Constants

- Constants cannot be modified
- e.g. `const int LIMIT = 5`
- Can be used in guards, updates and invariants

Bounded integer variables

- `int[min, max]`
- `min` and `max` are the lower and upper bound, respectively
- e.g. `int[2, 5] x;` declares `x` to be a variable whose value is taken from the set `{2, 3, 4, 5}`
- If the bounds are omitted, a 16 bit signed integer is assumed
- Can be used in guards, updates and invariants

UPPAAL Language Extensions - Data (ctd)

Arrays

- Allowed for clocks, channels, constants and integer variables
- e.g. `clock c[3];` declares `c` to be an array of length 3, where each element is a clock. Indexing starts at 0.
- Can be initialised
  - `int[0,2] x[3] = {1, 0, 2};`

Structs

- Declared with the `struct` keyword, as in C
- `struct {int[0,9] x; int[0,9] y;} point = {3, 4};`
- `point` is a `struct` with fields `x` and `y`
- `point.x` has the value 3

UPPAAL Language Extensions - Data (ctd)

Typedefs

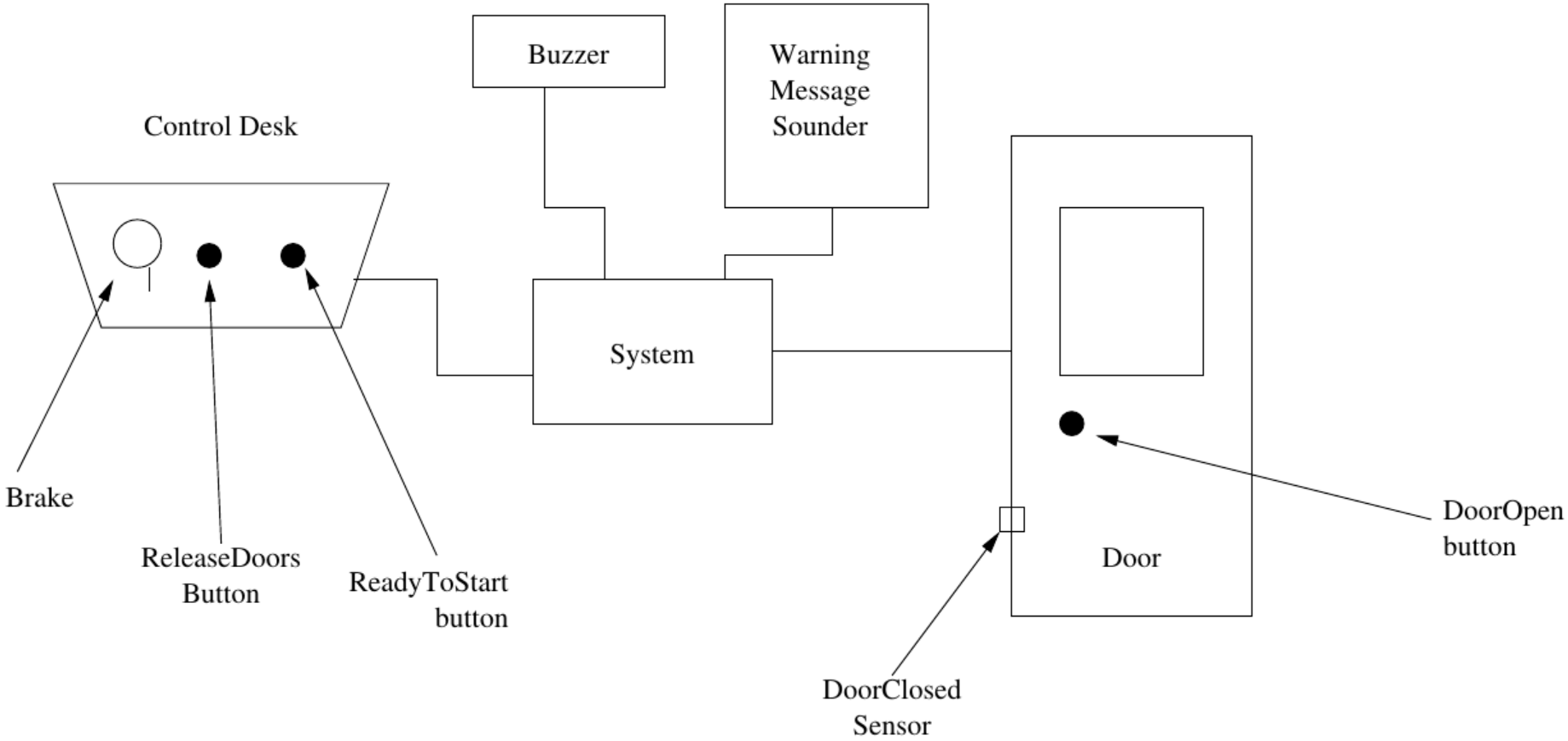
- User-defined types can be name using `typedef`, as in C
- `typedef struct {int[0,9] x; int[0,9] y;} point_t;`
- `point_t point = {3, 4};`

Functions

- Can be defined globally or locally
- Syntax is similar to C, except no pointer variables. C++ syntax for references is available for use with template parameters

```
int[0, 9] min(point_t p) {
  if (p.x <= p.y) {
    return p.x;
  }
  else {
    return p.y;
  }
}
```

Case study — Metro doors



Metro doors requirements

On the San Serif Metro System, each train has two sliding doors—one on each side. When a train arrives in a station, the driver applies the brake to bring the train to a stop. She then presses the *ReleaseDoors* button. Provided the brake is applied and the train is stationary, this enables the buttons beside the door that is next to the platform (there is a sensor which informs the system of this). When enabled, pressing the *DoorOpen* button opens the relevant door.

When the driver is ready to start, he presses another button—the *ReadyToStart* button. This causes a buzzer to sound, followed by a warning message “Stand clear of the doors, please”. The door is then closed after 5 seconds.

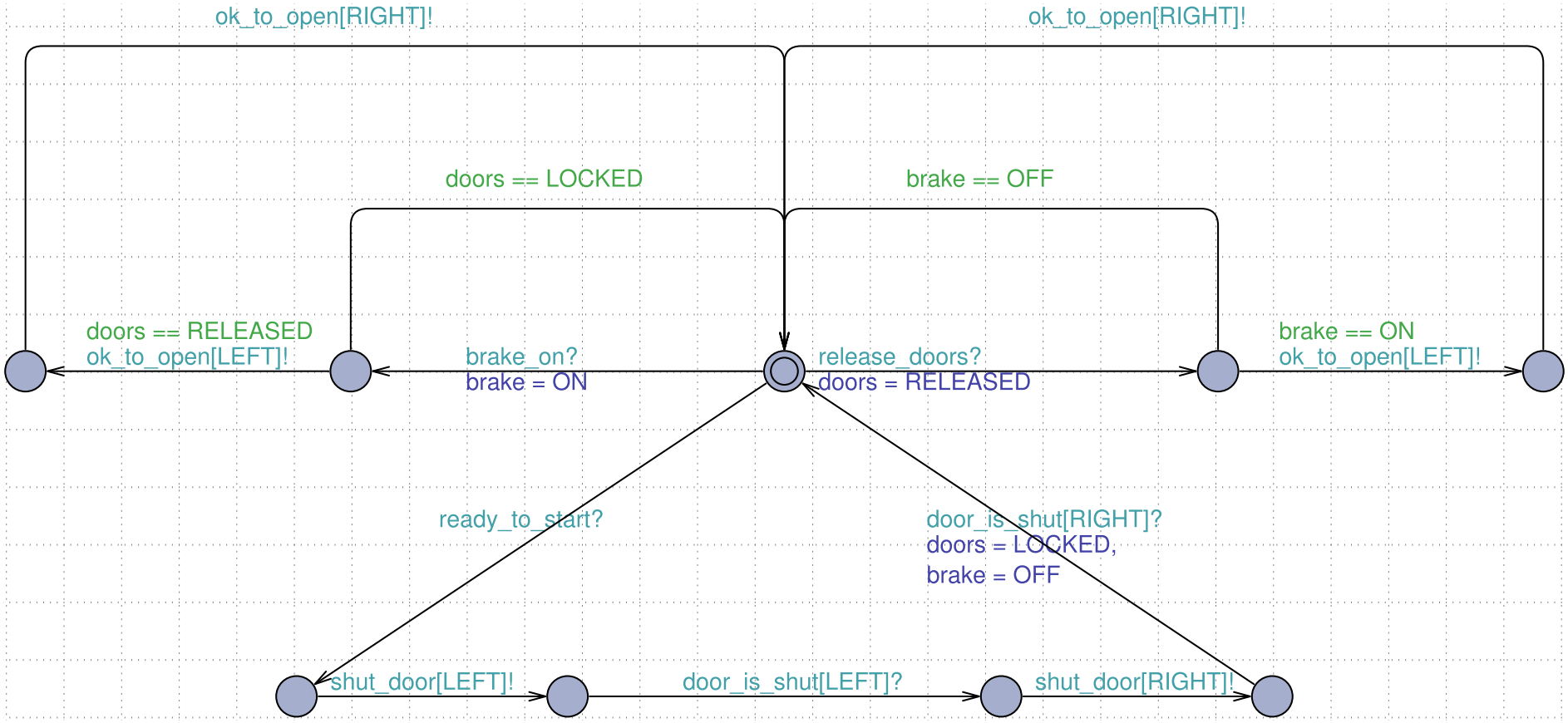
When it has been confimed that the doors are closed, the brake is released automatically and the train moves off.

A schematic diagram of the system is available.

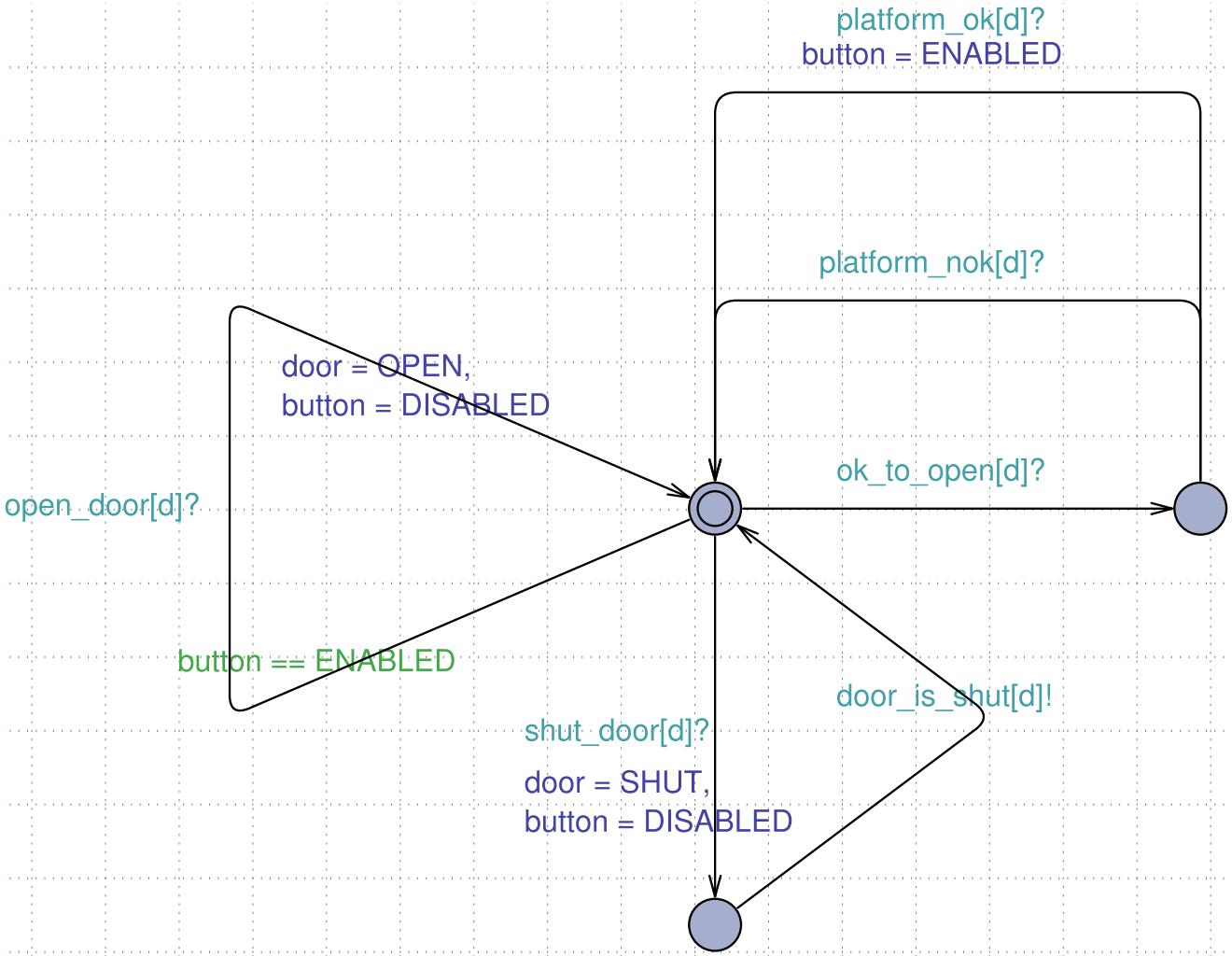
Constructing a model

- Identify the processes (system and environment)
- Identify the communication channels
- Identify the messages passed on each channel
- Gradually build up the model of the behaviour of the processes
  - Test early, test often
- Introduce variables as required – make each variable as small as possible(bit, byte , short, int, . . .)
- Abstract, abstract, abstract . . .

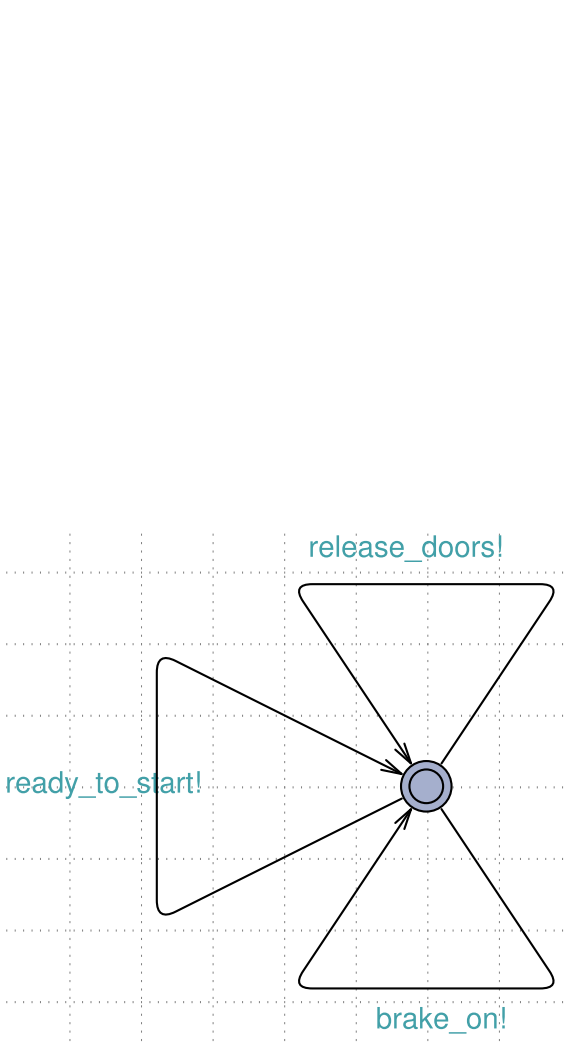
ControlDesk



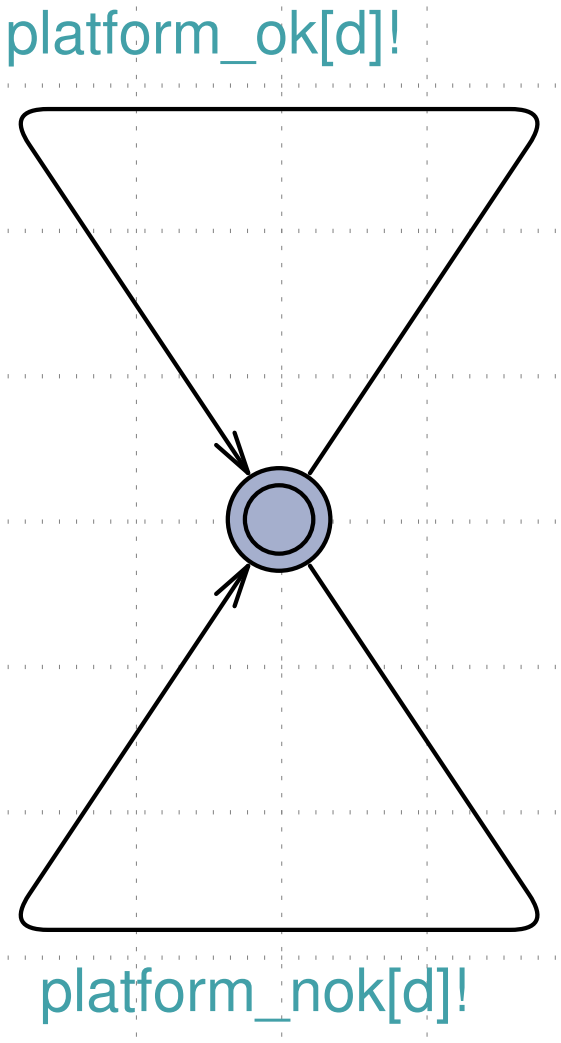
Door


platform_ok[d]?
button = ENABLED
platform_nok[d]?
door = OPEN,
button = DISABLED
ok_to_open[d]?
open_door[d]?
button == ENABLED
door_is_shut[d]!
shut_door[d]?
door = SHUT,
button = DISABLED

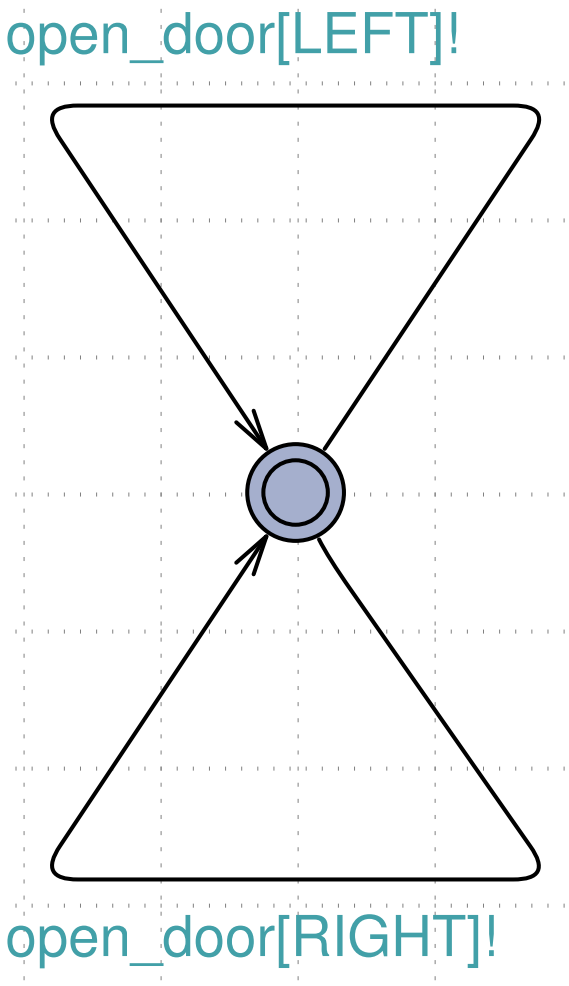
Environment agents



Driver



Platform



Passenger

Some door controller properties

- Invariant: whenever a door is open, the brake is on

```
A[] ((DoorLeft.door == OPEN or DoorRight.door == OPEN) imply ControlDesk0.brake == ON)
```

- Invariant: whenever the door buttons are enabled, the brake is on

```
A[] ((DoorLeft.button == ENABLED or DoorRight.button == ENABLED)
     imply ControlDesk0.brake == ON)
```

- There is some behaviour in which a door is open

```
E<> (DoorRight.door == OPEN or DoorLeft.door == OPEN)
```

- There is some behaviour in which a button is enabled

```
E<> (DoorRight.button == ENABLED or DoorLeft.button == ENABLED)
```

- Add your own properties
- Think primarily in terms of
  - Safety properties
  - Bounded Response properties
  - ‘Sanity’ checks

Reducing model complexity

When a verification cannot be completed because of computational complexity; here are some strategies that can be applied to combat this problem.

1. Try to make the model more general, more abstract. Remember that you are constructing a verification model and not an implementation.
2. Remove everything that is not directly related to the property you are trying to prove: redundant computations, redundant data. Avoid counters.

Reducing model complexity

When a verification cannot be completed because of computational complexity; here are some strategies that can be applied to combat this problem.

3. Asynchronous communication is a significant source of complexity in verification. Use synchronous channels where possible. If you need to use asynchronous communication, reduce the number of buffer slots to a minimum (use 2, or 3 slots to get started).
4. Look for processes that merely transfer messages. Consider if you can remove processes that only copy incoming messages from one channel into another, by letting the sender generate the final message right away. If the intermediate process makes choices (e.g., to delete or duplicate, etc.), let the sender make that choice, rather than the intermediate process.

Reducing model complexity

5. Combine local computations into sequences of committed locations
6. Be careful about choice of data types. Choose the smallest available type that you need for the data. Avoid leaving scratch data around in variables. Reset variables when possible.
7. If possible to do so: combine the behavior of two processes into a single one. Generalize behaviour; focus on coordination aspects (i.e., the interfaces between processes, rather than the local computation inside processes).
8. Try to exploit state space reductions techniques
   - Store fewer states
   - Store smaller states