# CS4211 Formal Methods of Software Engineering Project

## Parking Lot System

**Name of Student**
Ma Zhencai Jayden

**Matriculation Number**
U087139J

**Date**
19/11/2011

## 1. Introduction

The main aim of this project is to model a real life parking lot system using both Z notation and PAT. Comparisons will be made between the 2 models to allow a better understanding of the limitations of the 2 specification languages.

## 2. Problem Description

**Parking Lot System**
The parking lot system has various components with different functions. But in general, it consists of an entry, an exit, the cars and the parking lots.

The entry only allows cars to enter the carpark when the carpark is not full.

The exit only allows cars to exit after it has received payment. Payment is calculated in accordance to the length of stay.

The cars can only enter the carpark when the entry allows it to enter. Likewise, it can only exit the carpark only when the exit allows it to leave.

The parking lots could be modelled to allow cars to park into them, but this does not in any ways affect the basic features of the parking lot system.

Further modelling could be done to make the system closer to the real world situation, but the above are the minimal requirements of a parking lot system.

## 3. System Modelling

### Z notation

**Carpark**

carcount : N
lots : ParkingLot
cars : Car
parkAt : Car ↔ ParkingLot
currentTime: N
entryTime: Car ↔ N
parkedForDays : Car ↔ N

# lot = 4

**Carpark $_{INIT}$**

carcount = 0
lots = {lot1, lot2, lot3, lot4}
cars = {}
currentTime = 0

**Entering**

Δ Carpark
c? : Car

carcount < 4
carcount′ = carcount + 1
entryTime′ = entryTine U {c? ↔ currentTime}
parkedForDays′ = parkedForDays U {c? ↔ 0}
cars′ = car U {c?}

**Parking**

Δ Carpark
c? : Car
p? : ParkingLot

c? ∈ cars
{ _↔p?} ∉ parkAt
parkAt′ = parkAt U {c? ↔ p?}

**UnParking**

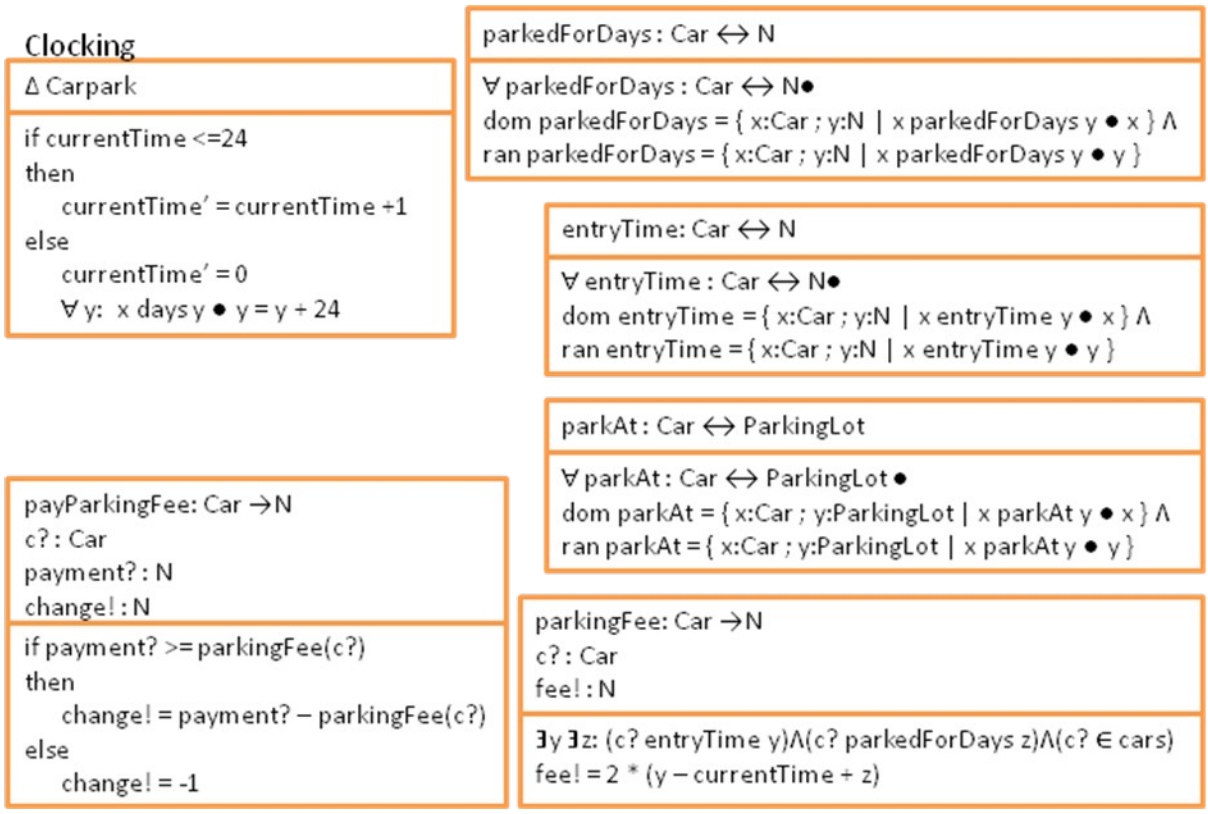
Δ Carpark
c? : Car
p? : ParkingLot

c? ∈ cars
{ c?↔p?} ∈ parkAt
parkAt′ = parkAt − {c? ↔ p?}

**Exiting**

Δ Carpark
c? : Car
payment? : N
change! : N

c? ∈ cars
{c? ↔ _ } ∉ parkAt
if payParkingFee(payment?) == -1
then
    change! = payment?
else
    change! = payParkingFee(payment?)
    carcount ′ = carcount − 1
    cars′ = car − c?




In this model, cars can only enter the carpark via Entering. Only after the car has entered, then rest of model allow the car to do anything else. Upon entry, the car count is increased and time of entry is initialised to the current time. After the car enters, it can either park itself or choose to leave the carpark. If the car chooses to park, it can only take the next action of unparking itself before it can choose to exit, or change to another parking lot. On exiting, the car drive will have to make payment to the gantry and the parking fee is calculated based on the entry time. After exiting, the car count is decreased back. Time is modelled as an operation and it arbitrary increases it by 1 hr each time.

## PAT

```
// Done by  : Ma Zhencai Jayden
// Matic no : U087139J

#define NO_OF_CARS 2; // Specifys number of cars that are in this world.
#define MAX_LOTS 2; // Specifys maximum lots in the carpark.

channel car_at_entry 0;
channel car_enter 0;
channel car_at_exit 0;
channel car_exit 0;
channel make_payment 0;

var car_count = 0;

// Car is modelled to the actions it can take.
Car(i) = Car_Enter(i);
Car_Enter(i) = car_at_entry!i -> car_enter!i -> Car_Parked(i);
Car_Parked(i) = car_parked.i -> Car_Exit(i);

// Upon reaching the exit, the gantry will detect and the payment engine will see if the car has enough
money.
// The car can then make leave if payment is made, or do other actions if payment is not made.
Car_Exit(i) = car_at_exit!i -> make_payment!i -> (
                                        payment_made -> car_exit!i -> Car(i)
// Car exits if theres enough payment.
                                        [] insufficient_funds -> (
                                                Car_Parked(i)
// The car goes back to parking and decides what to do before trying to exit again.
                                                [] Car_Exit(i)
// The car tries to exit again with maybe a different cashcard or the same one.
                                                )
                                        );

// The Entry Gantry uses ERP, so no button is pushed.
Entry_Gantry() = [car_count<MAX_LOTS] car_at_entry?i -> entry_open {car_count++} -> car_enter?i
-> entry_close -> Entry_Gantry();

// The Exit Gantry uses ERP, so no button is pushed.
// The vehicle is checked if it has enough money upon reaching the exit, after which it determines if
theres enough money and do the actions.
Exit_Gantry() = car_at_exit?i -> (
                        payment_made -> exit_open {car_count--;} -> car_exit?i -> exit_close ->
Exit_Gantry()
                        [] insufficient_funds -> Exit_Gantry()
                        );

// The Payment control which determines if the payment is made, or theres insufficient money.
// This synchronises the events in Exit_Gantry and Car_Exit.
Payment() = make_payment?i -> (
                        payment_made -> Payment()
                        [] insufficient_funds -> Payment()
                        );

SpawnCars() = ||| i:{0..NO_OF_CARS-1}@Car(i);
Carpark() = SpawnCars() || Entry_Gantry() || Exit_Gantry() || Payment();
```

```
// Assertions

#assert Carpark() deadlockfree;

// checks that car count cannot be lower than 0 and cannot go beyond maximum number of cars
#define carsCountOutOfRange (car_count>MAX_LOTS || car_count<0);
#assert Carpark() reaches carsCountOutOfRange;

// a car that enters will eventually exit
#assert Carpark() |= (car_enter.0 -> <>car_exit.0);

// a car that has paid will eventually exit
#assert Carpark() |= (payment_made.0 -> <>car_exit.0);

// car 0 and car 1 cannot enter consequetively immediately
ConsequetiveEntryRun() = car_at_entry.0 -> entry_open -> car_enter.0 -> car_enter.1 -> Skip()
 []car_at_entry.0 -> entry_open -> car_enter.1 -> car_enter.0 -> Skip()
 []car_at_entry.1 -> entry_open -> car_enter.0 -> car_enter.1 -> Skip()
 []car_at_entry.1 -> entry_open -> car_enter.1 -> car_enter.0 -> Skip();
ConsequetiveEntry() = ConsequetiveEntryRun() || Carpark();
#assert ConsequetiveEntry() refines Carpark();
```

## 4. Investigated Properties

The following properties are investigated in PAT.

```
#assert Carpark() deadlockfree;
```
It is supposed to be deadlock free as there is no scenario that it will cause a deadlock based on the model.

```
#define carsCountOutOfRange (car_count>MAX_LOTS || car_count<0);
#assert Carpark() reaches carsCountOutOfRange;
```
This checks that the car count goes lower than 0 or can go beyond maximum number of cars. This is supposed to validate as false since the system cannot allow less than 0 cars and more than the maximum number of cars to be in the carpark.

```
#assert Carpark() |= (car_enter.0 -> <>car_exit.0);
```
This checks that if a car enters the carpark, it will eventually be able to exit the carpark. This is supposed to validate as true.

```
#assert Carpark() |= (payment_made.0 -> <>car_exit.0);
```
This checks that if a car has made his payment, it will eventually exit the carpark. This is supposed to validate as true.

```
ConsequetiveEntryRun() = car_at_entry.0 -> entry_open -> car_enter.0 -> car_enter.1 -> Skip()
[]car_at_entry.0 -> entry_open -> car_enter.1 -> car_enter.0 -> Skip()
[]car_at_entry.1 -> entry_open -> car_enter.0 -> car_enter.1 -> Skip()
[]car_at_entry.1 -> entry_open -> car_enter.1 -> car_enter.0 -> Skip();
ConsequetiveEntry() = ConsequetiveEntryRun() || Carpark();
#assert ConsequetiveEntry() refines Carpark();
```
This checks that 2 cars can enter through the gantry at the same time. This is supposed to validate as false.

## 5. Experiments

#assert Carpark() deadlockfree;
********Verification Result********
The Assertion (Carpark() deadlockfree) is **VALID**.

********Verification Setting********
Admissible Behavior: All
Search Engine: First Witness Trace using Depth First Search
System Abstraction: False

********Verification Statistics********
Visited States:174
Total Transitions:429
Time Used:0.0266168s
Estimated Memory Used:10506.496KB

#define **carsCountOutOfRange** (car_count>MAX_LOTS || car_count<0);
#assert Carpark() reaches carsCountOutOfRange;
********Verification Result********
The Assertion (Carpark() reaches carsCountOutOfRange) is **NOT valid**.

********Verification Setting********
Admissible Behavior: All
Search Engine: First Witness Trace using Depth First Search
System Abstraction: False

********Verification Statistics********
Visited States:174
Total Transitions:429
Time Used:0.0281687s
Estimated Memory Used:10442.408KB

#assert Carpark() **|=** (car_enter.0 **-> <>**car_exit.0);
********Verification Result********
The Assertion (Carpark() |= ( car_enter.0-><> car_exit.0)) is **VALID**.

********Verification Setting********
Admissible Behavior: All
Search Engine: Loop Existence Checking - The negation of the LTL formula is a safety property!
System Abstraction: False
Fairness: no fairness

********Verification Statistics********
Visited States:0
Total Transitions:0
Time Used:0.0145988s
Estimated Memory Used:8545.2KB

#assert Carpark() **|=** (payment_made.0 **->** **<>**car_exit.0);
********Verification Result********
The Assertion (Carpark() |= ( payment_made.0-><> car_exit.0)) is **VALID**.

********Verification Setting********
Admissible Behavior: All
Search Engine: Loop Existence Checking - The negation of the LTL formula is a safety property!
System Abstraction: False
Fairness: no fairness

********Verification Statistics********
Visited States:0
Total Transitions:0
Time Used:0.0124726s
Estimated Memory Used:8544.296KB

**ConsequetiveEntryRun**() = car_at_entry.0 -> entry_open -> car_enter.0 -> car_enter.1 -> **Skip**()
[]car_at_entry.0 -> entry_open -> car_enter.1 -> car_enter.0 -> **Skip**()
[]car_at_entry.1 -> entry_open -> car_enter.0 -> car_enter.1 -> **Skip**()
[]car_at_entry.1 -> entry_open -> car_enter.1 -> car_enter.0 -> **Skip**();
**ConsequetiveEntry**() = **ConsequetiveEntryRun**() || **Carpark**();
#assert ConsequetiveEntry() refines Carpark();
********Verification Result********
The Assertion (ConsequetiveEntry() refines Carpark()) is **NOT valid**.
The following trace is allowed in ConsequetiveEntry(), but not in Carpark().
<init -> car_at_entry.1 -> car_at_entry.0>

********Verification Setting********
Admissible Behavior: All
Search Engine: On-the-fly Trace Refinement Checking using Depth First Search
System Abstraction: False

********Verification Statistics********
Visited States:7
Total Transitions:11
Time Used:0.0583697s
Estimated Memory Used:8711.184KB

## 6. Discussions

In the PAT model, time was not modelled in. Initial attempt at modelling time similar to that of the Z notation model caused a huge state explosion that resulted in the failure to do any verification. However, after numerous attempts at doing so, it is observed that all of the time, the number of states would grow too large for verifications to be done. Hence as a result, time was excluded from the PAT model.

For the PAT model, as time could not be modelled, payment was modelled in the way that either the sum was fully paid, or it was not paid at all. Even though this is a little different from the original specification requirements, however, it is still fairly similar in attaining the results of having the car not being able to leave if it did not pay.

The limitation of the Z model was such that highly descriptive events could not be described. Events such as the opening and closing of gates serve no purpose at all if specified in the Z notation. This however could be done in PAT by adding more states to the model and naming those states as the name of the events.

## 7. Feedback and Suggestion for PAT

There is only one bug discovered so far. When running PAT using windows 7. After pressing the simulation button and having the simulation window pop-up, if one was to minimise the simulation window and maximising it again, the display on the simulation window would become messed up and the right side panels cannot be accessed.

## 8. Conclusion

In conclusion, even though Z notation proved to be more versatile and good for modelling, it is harder to model as there is no model checker like PAT which can verify that the model is correct. PAT however has the checker and with the various verification capabilities showed more conclusive results to the models implemented even though it was disadvantaged by the limitations of computing power.