National UniversityA logo of a university

Description automatically generated

of Sciences and Technology (NUST)

Formal Methods

SE 320 Project Report

BESE 12-B  
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Instructor : Dr. Sidra Sultana**

|  |  |
| --- | --- |
| **NAME** | **CMS** |
| Abdullah Tahir | 393056 |
| Laiba Atiq | 372187 |
| Imaan Ibrar | 373982 |

Contents

[**NUST SmartPark: Optimizing Parking Efficiency and User Experience 4**](#_ojfn4gjm32s6)

[**Current challenges 4**](#_wh6blk1065u2)

[**Proposed Solution 4**](#_v6cfkyakwxj4)

[**Objectives 4**](#_8n273x3krs3r)

[**WORKING 4**](#_ttnsimy7gbx2)

[Real-time Occupancy Monitoring: 5](#_ucmdqtz9rx6l)

[Dynamic Parking Allocation: 5](#_ptefzlj5x48u)

[User Interface and Mobile App: 5](#_idkhujlnul7m)

[Automated Entry and Exit: 5](#_hyu5x2ozzx08)

[Payment Integration: 5](#_h8meg6nmpmvn)

[Data Analytics and Reporting: 5](#_8o87wnjvfqsp)

[Alerts and Notifications: 5](#_m6783lg1q8sx)

[Efficient Staff Management: 5](#_27hvyk4foi6p)

[**Benefits: 5**](#_f3ccyb5rpag9)

[System Overview 6](#_30j0zll)

[**System Specification 6**](#_8hqo2qq0qdzp)

[Data Types: 7](#_t2qcb3uit2x8)

[Variables: 7](#_c8kpwgi7xnbc)

[Functions/Actions: 7](#_rg3tp5qbu57x)

[enterParkingLot(vehicle: Vehicle): 7](#_ejcbohh2qmte)

[exitParkingLot(ticket: Ticket): 7](#_g1ahr0hkje57)

[makePayment(ticket: Ticket, paymentType: PaymentType, amount: Money): 8](#_idsogv13ejde)

[calculateFine(ticket: Ticket, parkedDuration: Time): 8](#_32u38sl6n5sx)

[payFine(vehicle: Vehicle, amount: Money): 8](#_caxsi3j24p4s)

[updateGateStatus(gate: Gate, newStatus: GateStatus): 8](#_ulam1ez7f61r)

[System Constraints: 8](#_6ow5enirorsp)

[Verification Properties: 9](#_9rantjbjqz4g)

[Logical Constraints: 9](#_gm3wzmkhfz15)

[Behavioral Model: 9](#_gtmzzrds8eqq)

[Class Heirarchy 10](#_lssi0gnjh1rl)

[OntoGraff 10](#_j0hsmj85blk6)

[Timed Automata Modeling 11](#_3znysh7)

[PARKING LOT ENTRY GATE TIMED AUTOMATA 11](#_2et92p0)

[Locations: 11](#_tyjcwt)

[Clocks: 12](#_3dy6vkm)

[Channels: 12](#_szqhy88bx9fp)

[Transitions: 12](#_1t3h5sf)

[Invariants: 12](#_4d34og8)

[PARKING LOT EXIT GATE TIMED AUTOMATA 12](#_2s8eyo1)

[Locations: 13](#_17dp8vu)

[Clocks: 13](#_3rdcrjn)

[Channels: 13](#_y7lqz4k5jzv8)

[Transitions: 13](#_26in1rg)

[Invariants: 14](#_lnxbz9)

[CAR PARKING SPACE FINE TIMED AUTOMATA 14](#_35nkun2)

[Locations: 14](#_1ksv4uv)

[Clocks: 14](#_44sinio)

[Channels: 14](#_adswz45yig4g)

[Transitions: 15](#_2jxsxqh)

[Invariants: 15](#_z337ya)

[TICKET TIMED AUTOMATA 15](#_3j2qqm3)

[Locations: 15](#_1y810tw)

[Clocks: 16](#_4i7ojhp)

[Channels: 16](#_kjyd2a4xfpak)

[Transitions: 16](#_2xcytpi)

[Invariants: 17](#_1ci93xb)

[VEHICLE TIMED AUTOMATA 17](#_3whwml4)

[Locations: 18](#_2bn6wsx)

[Clocks: 18](#_qsh70q)

[Channels: 18](#_7aelybz16sbc)

[Transitions: 19](#_3as4poj)

[PAYMENT TIMED AUTOMATA 20](#_s1oshpb1kx3)

[Locations: 20](#_7inv2o4f9yhj)

[Clocks: 20](#_ft61meh7vg8e)

[Channels: 21](#_abiv1l6z65nv)

[Transitions: 21](#_9rpjov471p5i)

[Invariants: 22](#_ycea0xo2w248)

[Declarations 22](#_2ocaib4jtxd6)

[System Declarations 22](#_1bhtnutt8wwv)

[UPPAAL Simulation 23](#_5k6tgjvuhid4)

[Swimlanes 32](#_nkxfldvvbrqy)

[**Verification: 33**](#_uyjzwn9k87f3)

[Deadlock: 33](#_e5sywiz2k8lh)

[SAFETY 33](#_y3drw0knndau)

[Utility: 34](#_2zpipsdz11wa)

[Reachability: 34](#_kbua8r4z43aq)

[Liveness: 35](#_skwwhkjf9cga)

[Fairness 35](#_hrvk6gk3mp6f)

[Verification in uppal: 35](#_1qovcf3b1ix)

[CONCLUSION 36](#_1pxezwc)

# NUST SmartPark: Optimizing Parking Efficiency and User Experience

Introducing the innovative SmartPark system at NUST: revolutionizing parking management with advanced technology for optimized efficiency, enhanced user experience, and sustainable practices.

# Current challenges

During peak hours, NUST's parking lots face significant challenges due to high vehicle traffic, leading to congestion and limited parking availability. This results in frustration among drivers who struggle to find parking spaces, leading to inefficiencies and delays in parking. Additionally, the manual management of parking spaces can lead to errors and inaccuracies in tracking occupancy, further complicating the parking experience for users.

# Proposed Solution

Implementing NUST SmartPark using formal methods can address these challenges by optimizing parking space allocation, providing real-time availability updates, and streamlining entry and exit procedures. This would not only improve the overall efficiency of the parking system but also enhance the user experience by reducing wait times and minimizing the risk of parking-related issues.

# Objectives

The smart parking system at NUST aims to achieve several key objectives:

● Optimize parking efficiency by maximizing space utilization.

● Improve user experience through real-time availability updates and convenient payment options.

● Minimize congestion in parking lots.

● Enhance accessibility for all users, including those with disabilities.

● Increase operational efficiency through automation and data analytics.

● Enhance security measures for vehicles and parking areas.

● Promote sustainable transportation practices.

● Generate valuable insights for future planning and policy-making related to parking management.

# WORKING

NUST SmartPark will utilize advanced technologies and formal methods to optimize parking efficiency and improve user experience. Here's how it will work:

## Real-time Occupancy Monitoring:

The system will be equipped with sensors installed in each parking space to detect vehicle presence. These sensors will continuously monitor parking space occupancy in real time.

## Dynamic Parking Allocation:

Using the data from the occupancy sensors, the system will dynamically allocate parking spaces to incoming vehicles. It will prioritize available spaces based on proximity, vehicle size, and any specific requirements such as handicap-accessible spaces.

## User Interface and Mobile App:

Users will have access to a user-friendly interface and a mobile app where they can view real-time parking availability, reserve parking spaces in advance, and make payments seamlessly.

## Automated Entry and Exit:

The system will automate the entry and exit processes using license plate recognition technology or RFID tags. This will enable smooth and quick access for authorized vehicles without the need for manual intervention.

## Payment Integration:

Payment for parking will be integrated into the system, allowing users to pay via mobile wallets or credit cards. The system will generate digital parking tickets and receipts for each transaction.

## Data Analytics and Reporting:

The system will gather data on parking usage, peak hours, and trends. This data will be analyzed to optimize parking lot layout, improve traffic flow, and provide insights for future planning and decision-making.

## Alerts and Notifications:

Users will receive alerts and notifications regarding parking availability, reservation confirmations, and expiration reminders, enhancing communication and convenience.

## Efficient Staff Management:

The system will facilitate efficient management of parking staff by providing real-time data on parking lot occupancy and streamlining staff allocation based on demand.

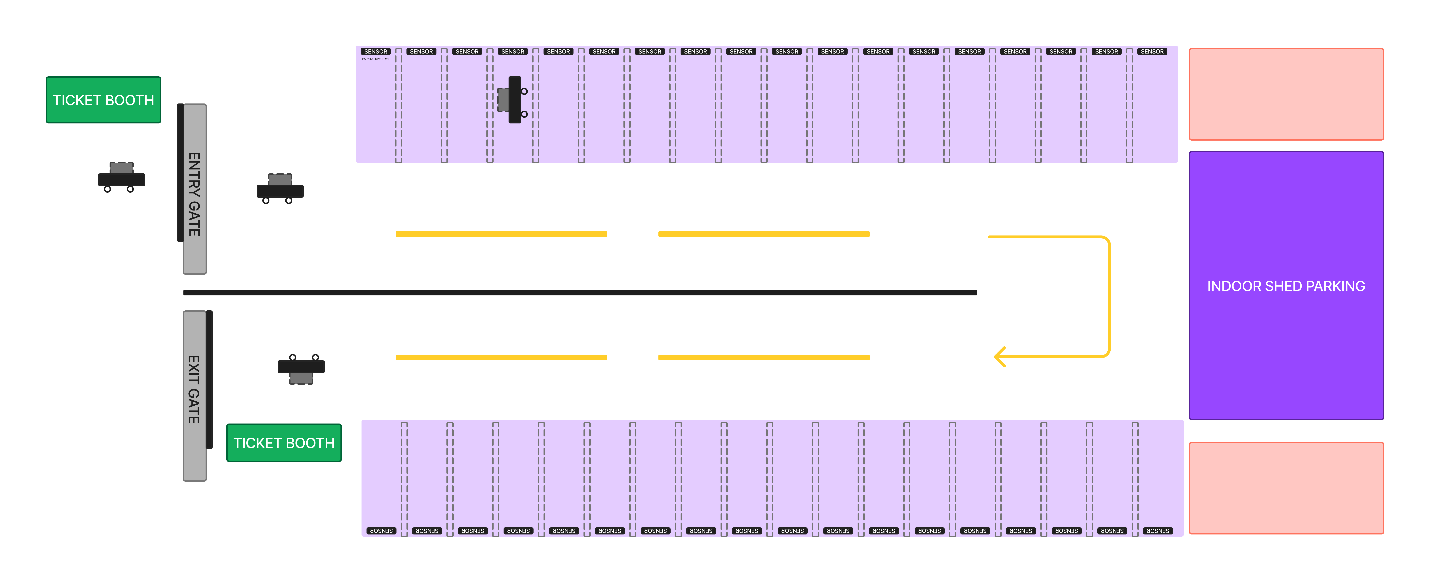
# Benefits:

The expected benefits of implementing the SmartPark system at NUST include:

1. Improved parking efficiency and utilization, leading to reduced congestion and wait times.
2. Enhanced user experience with real-time availability updates, convenient payment options, and streamlined entry/exit processes.
3. Increased accessibility for all users, including those with disabilities, through designated spaces and accessibility features.
4. Operational efficiency gains through automation, data analytics, and optimized staff allocation.
5. Enhanced security measures for vehicles and parking areas, ensuring a safe environment.
6. Promotion of sustainable transportation practices such as carpooling and electric vehicle support.
7. Insights-driven planning for future infrastructure development and parking policy enhancements.

# System Overview

The NUST Smart Parking System revolutionizes the parking experience through an integrated approach to management and automation. As vehicles approach the entry gate, they are seamlessly guided through the ticketing process, with options for payment via cash or a parking card. Crucially, the system continuously monitors parking space availability using sensors deployed in each space, ensuring that users are directed to vacant spots upon entry. Once payment is complete, vehicles are granted access, and entry gates open automatically, allowing smooth entry into the facility. Through sophisticated algorithms, the system allocates parking spaces efficiently, minimizing congestion and maximizing space utilization. Each parking ticket comes with a predefined duration, typically 60 minutes, after which vehicles must exit the parking lot. Failure to do so results in automatic blocking of the parking space until an overtime fine is paid by the driver. This fine must be settled before the vehicle can exit the facility through the exit gate. Centralized monitoring and management systems oversee the entire operation, collecting real-time data on space availability, occupancy, and transactions, enabling proactive management and rapid resolution of any issues. Overall, the NUST Smart Parking System offers a seamless, user-friendly experience while optimizing space usage and enhancing operational efficiency.



# System Specification

## Data Types:

* **VehicleType:** Enum {Car, Motorcycle}
* **ParkingSpaceType:** Enum {Compact, Handicap, Regular}
* **ParkingLotType:** Enum {Indoor, Outdoor}
* **ParkingStatus:** Enum {Occupied, Vacant}
* **PaymentType:** Enum {Cash, Card, Mobile}
* **GateStatus:** Enum {Open, Closed, Opening}
* **FineStatus:** Enum {Imposed, Paid, None}

## Variables:

* **occupiedSpaces:** Integer (Initialized to 0)
* **availableSpaces:** Map of VehicleType to Array of ParkingSpace (Initialized with all spaces)
* **ticketedSpaces:** Map of ParkingSpace to Ticket
* **fines**: Map of Vehicle to (FineAmount, FineStatus)
* **entryGateStatus**: GateStatus (Initialized to Closed)
* **exitGateStatus**: GateStatus (Initialized to Closed)
* **cash/card:** bool
* **Change:** integer

## Functions/Actions:

### enterParkingLot(vehicle: Vehicle):

* IF occupiedSpaces < maxCapacity AND availableSpaces[vehicle.type] exists AND entryGateStatus == Open:
  + Assign a suitable space to the vehicle (from **availableSpaces**).
  + occupiedSpaces++
  + entryGateStatus = Opening (start closing process)
  + Return (Ticket, AssignedSpace)
* ELSE: Return "No space available" or "Gate closed"

### exitParkingLot(ticket: Ticket):

* IF ticket.isValid() AND (fines[ticket.vehicle] is None OR fines[ticket.vehicle].status == Paid) AND exitGateStatus == Open:
  + Mark **ticket.space** as Vacant
  + occupiedSpaces--
  + exitGateStatus = Opening (start closing process)
  + Return "Exit successful"
* ELSE IF fines[ticket.vehicle] exists AND fines[ticket.vehicle].status == Imposed:
  + Return "Fine pending"
* ELSE: Return "Invalid ticket" or "Gate closed"

### makePayment(ticket: Ticket, paymentType: PaymentType, amount: Money):

* IF amount >= ticket.price:
  + Process payment
  + IF payment successful: Return "Payment successful"
  + ELSE: Return "Payment failed"
* ELSE: Return "Invalid ticket or insufficient amount"

### calculateFine(ticket: Ticket, parkedDuration: Time):

* IF parkedDuration > allowedDuration:
  + Calculate fineAmount
  + fines[ticket.vehicle] = (fineAmount, Imposed)
  + Return fineAmount
* ELSE: Return 0

### payFine(vehicle: Vehicle, amount: Money):

* IF fines[vehicle] exists AND amount >= fines[vehicle]:
  + Process fine payment
  + IF payment successful:
    - fines.remove(vehicle)
    - Return "Fine paid successfully"
  + ELSE: Return "Fine payment failed"
* ELSE: Return "No outstanding fine or insufficient amount"

### updateGateStatus(gate: Gate, newStatus: GateStatus):

* IF gate.status != newStatus:
  + Simulate gate movement (delay, change status)

## System Constraints:

* 0 <= occupiedSpaces <= maxCapacity
* Only one vehicle can be in the entry/exit process at a time.
* A vehicle can only stay in the parking for 60 minutes without a fine.
* The exit gate remains closed until a valid ticket is presented and any fines are paid.
* The system cannot enter a deadlock state.
* The system can handle multiple cars entering and exiting simultaneously, adhering to capacity limits and safety protocols.
* The payment system should be secure and reliable..
* The fine calculation mechanism should be transparent and fair.
* The system should provide clear feedback to users about parking availability, payment status, and fines.

## Verification Properties:

* All reachability properties (E<>) should hold true to ensure every state is accessible.
* All safety properties (A[]) should hold true to guarantee system correctness under all conditions.
* The system should be deadlock-free (A[] not deadlock).
* If the parking lot is full, it should still be possible for cars to exit (A[] (occupiedspace == maxcapacity imply not deadlock)).
* A car should not be able to park and exit simultaneously (A[] not (car.parking && car.exiting)).

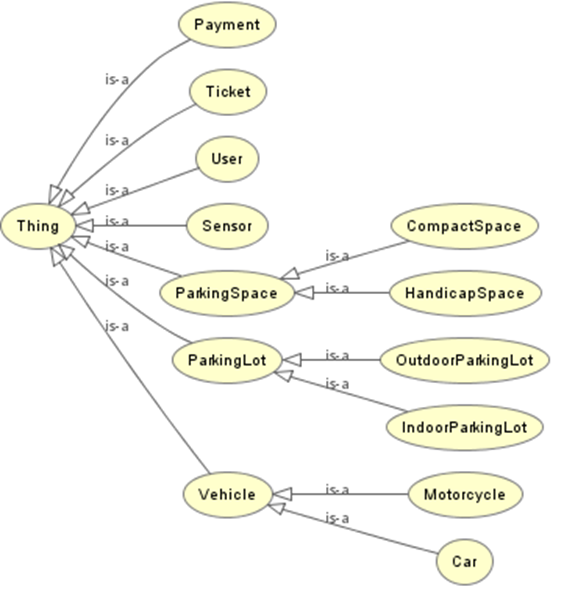
## Logical Constraints:

* **Occupancy:** 0 <= occupiedSpaces <= maxCapacity
* **Space Availability:** availableSpaces.length + occupiedSpaces = maxCapacity
* **Ticket Validity:** A ticket is valid if its space is Occupied and the current time is within the allowed parking duration.
* **Parking:** A car can only park if the car is in the correct parking space.
* **Fine payment:** A car can only leave after paying any withstanding fines.

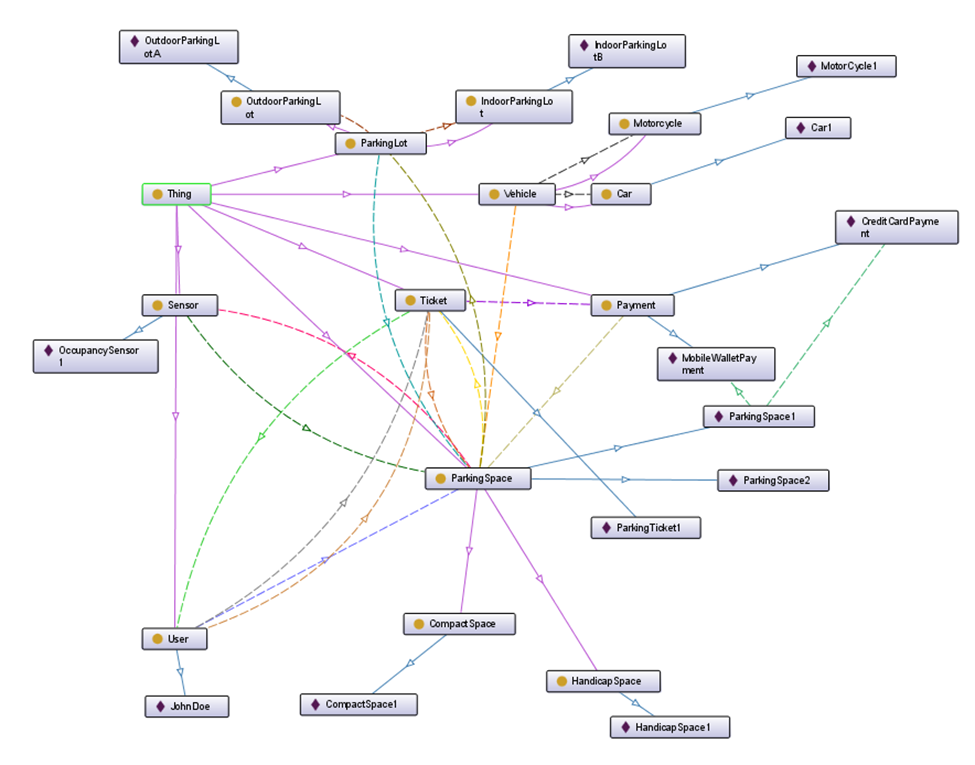
## Behavioral Model:

* **AG** (car.enters -> **EF** (car.parks AND ticket1.issued))
  + *Always Globally*, if a car enters, it will *Eventually in the Future* park and a ticket will be issued.
* **AG** (car.parked AND (time - ticket.issuedTime > maxParkingTime) -> **AF** fine1.imposed)
  + *Always Globally*, if a car is parked beyond the maximum time, *Always in the Future* a fine will be imposed.
* **AG** (fine1.timeexceeded -> **EF** fine1.nofine)
  + *Always Globally*, if a fine is imposed due to time exceeding the limit, it will *Eventually in the Future* be cleared or paid.
* **AG** (car.parked -> ticket1.issued)
  + *Always Globally*, if a car is parked, it must have been issued a ticket.
* **AG** (NOT (entry1.gateopened AND exit1.gateopened))
  + *Always Globally*, the entry gate and exit gate cannot be open at the same time.
* **AG** (occupiedSpaces <= maxcapacity)
  + *Always Globally*, the number of occupied spaces should not exceed the maximum capacity.
* **AG** (car.block -> **AF** (fine1.timeexceeded AND exit1.gateopened))
  + *Always Globally*, if a car is blocked, it will eventually have a fine imposed, and the exit gate will open once the fine is paid.
* **AG** (pay.paymentprocessing -> **EF** (pay.notprocessing OR ticket1.ticket))
  + *Always Globally*, if payment is in progress, it will *Eventually in the Future* either complete successfully (ticket issued) or fail (payment not processed).
* **AG** (car.exiting -> **EF** car.start)
  + *Always Globally*, if a car is in the exiting state, it will *Eventually in the Future* reach the start state (leave the lot).
* **AG** (occupiedspace == maxcapacity -> NOT entry1.gateopening)
  + *Always Globally*, if the parking lot is full, the entry gate cannot be opening.
* **AG** ((card == true OR cash == true) -> **AF** ticket1.ticket)
  + *Always Globally*, if a card or cash payment is initiated, it will *Always in the Future* result in a ticket being issued (assuming valid payment).

## Class Heirarchy

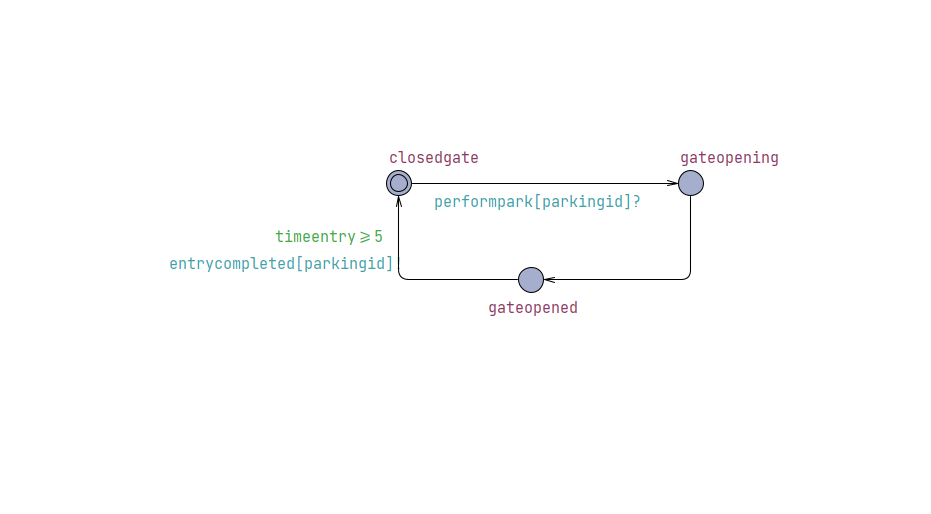


## OntoGraff



# Timed Automata Modeling

## PARKING LOT ENTRY GATE TIMED AUTOMATA



### Locations:

1. Gate Closed – Parking Lot Entry Gate (initial location)
2. Gate Opening
3. Gate Open

### Clocks:

Timeentry (continuous clock representing the passage of time)

### Channels:

* Performpark[parkingid]? : It is a signal received by the system which basically tells it to open the parking gate.
* Entrycompleted[parkingid]! : It is released by the parking gate to tell that the car has successfully parked.

### Transitions:

1. **Transition from Gate Closed to Opening Gate :**

* Synchronization: performpark[parkingid]?
* Reset: Time == 0

1. **Transition from Opening Gate to Open Gate:**

* Reset: Time ==0

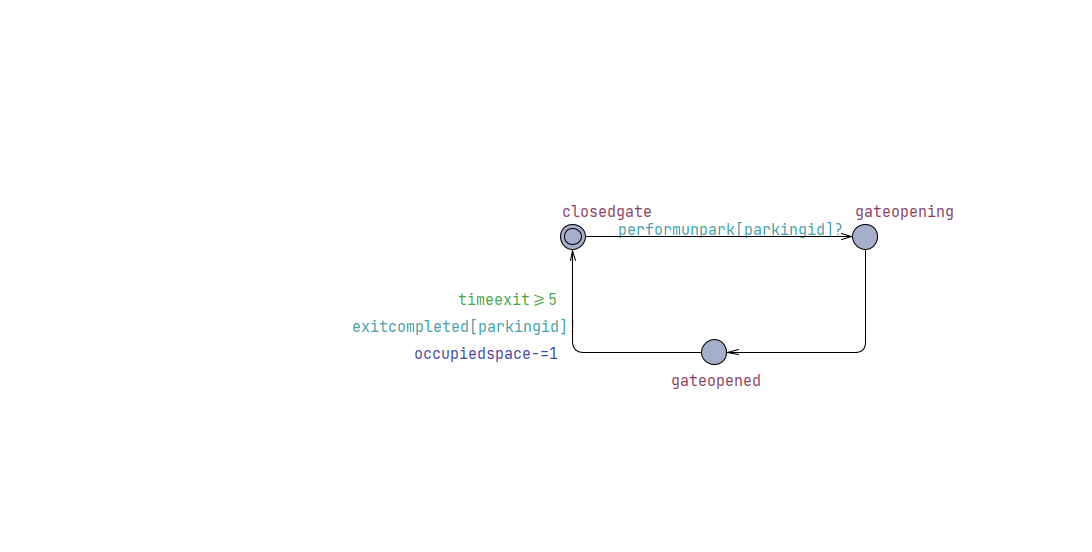
1. **Transition from Open Gate to Gate Closed:**

* Synchronization: Vehicle Entry Complete
* Guard: Time >= 5 sec

### Invariants:

1. Gate Closed: None
2. Gate Open: None
3. Opening Gate: time < 5

## PARKING LOT EXIT GATE TIMED AUTOMATA

****

### Locations:

1. Gate Closed – Parking Lot Exit Gate
2. Gate Opening
3. Gate Open

### Clocks:

Time (continuous clock representing the passage of time)

### Channels:

* Performunpark[parkingid]? : It is a signal received by the system which basically tells to open the parking gate.
* Exitcompleted[parkingid]! : it is released by the parking gate to tell that the car has left the system

### Transitions:

1. **Transition from Gate Closed to Opening Gate:**

* Synchronization: performunpark[parkingid]?
* occupiedSpace is decremented as an car leaves the Parking Lot

1. **Transition from Opening Gate to Gate Open:**

* Trigger: None
* Reset: Time == 0

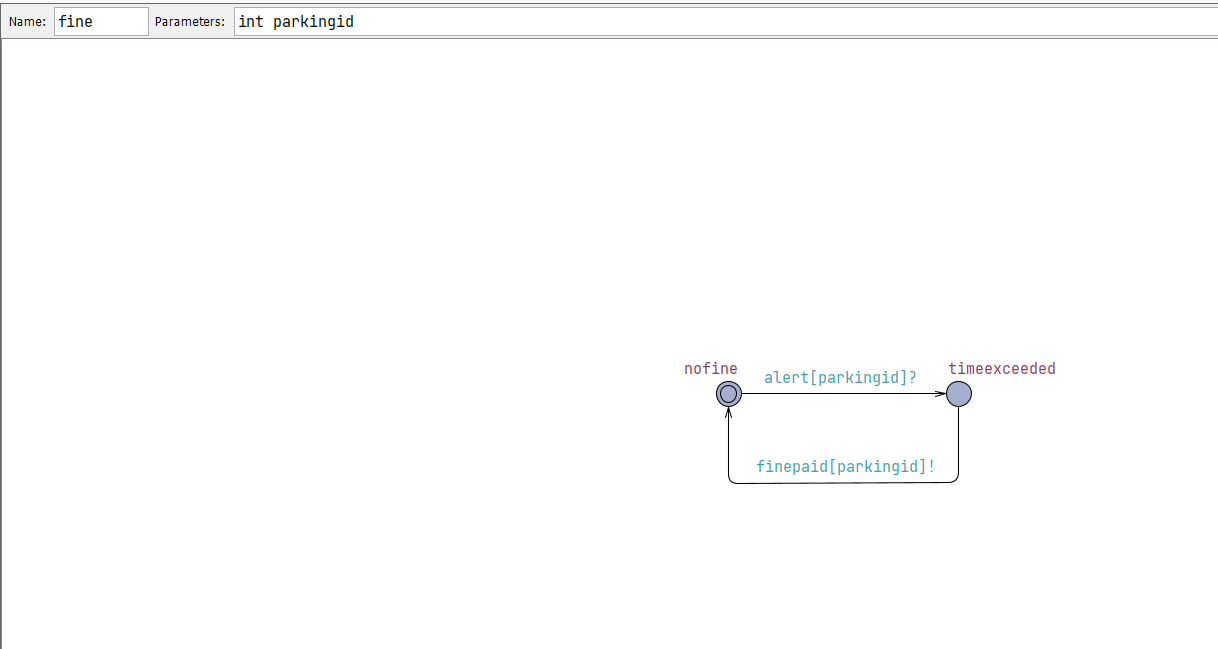
1. **Transition from Open Gate to Gate Closed:**

* Synchronization: ExitCompleted signal issued
* Guard: Time >= 5 sec

### Invariants:

1. Gate Closed: None
2. Gate Open: None
3. Opening Gate: time < 5 sec

## CAR PARKING SPACE FINE TIMED AUTOMATA

******

### Locations:

* nofine
* timeexceeded

### Clocks:

None

### Channels:

* Alert[parkingid]? : It is an alert generated by the system that the valid parking time (60 minutes) is overdue.
* Finepaid[parkingid]! : it is a signal emitted that the fine has been paid. It is received by the car automata and the car can thus exit the system.

### Transitions:

**1.Transition from nofine to timeexceeded:**

* Trigger: alert[parkingid]?

**2.Transition from timeexceeded to nofine:**

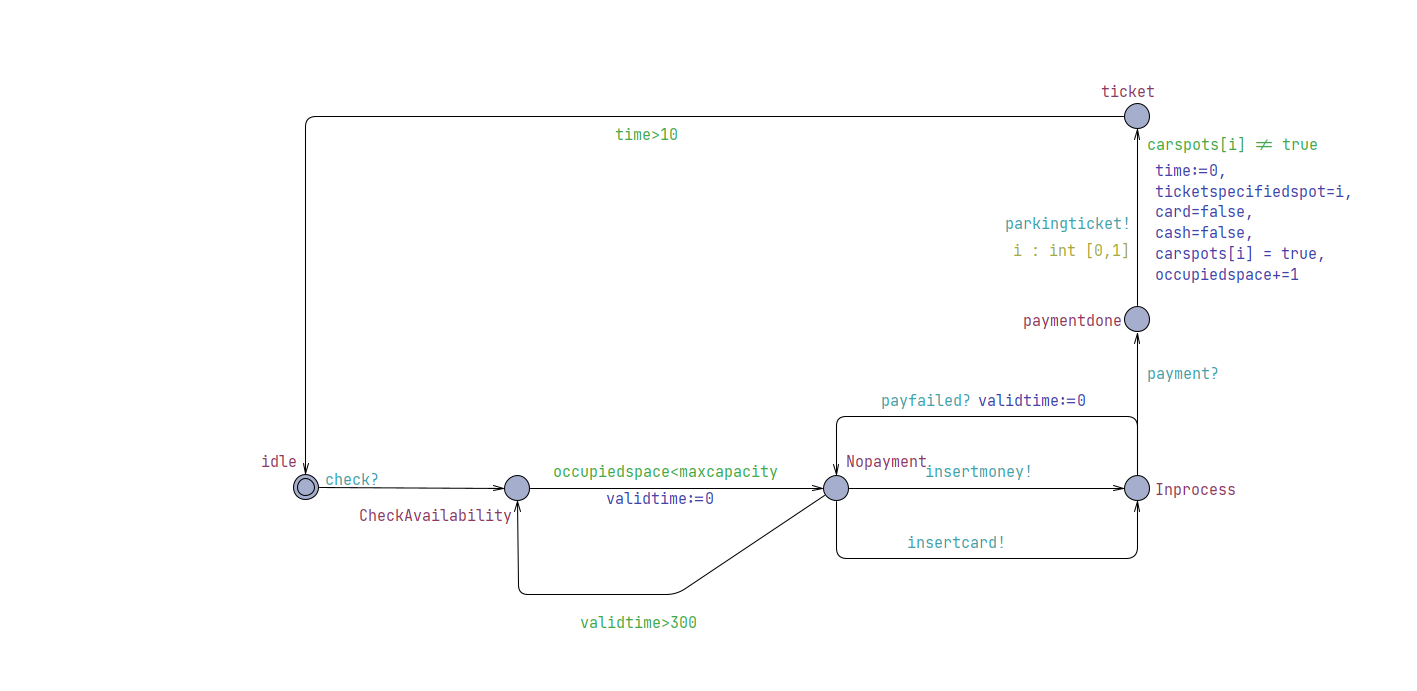
* Syncronization: finepaid[parkingid]!

So this transition is basically triggered when the fine is paid by the user.

### Invariants:

None

## TICKET TIMED AUTOMATA



### Locations:

* idle
* CheckAvailabilty
* NoPayment
* InProcess
* paymentdone
* Ticket

### Clocks:

* Time (continuous clock representing the passage of time)
* Validation time (continuous clock representing max limit for payment processing)

### Channels:

* Insertmoney! : tells that user inserted money
* Insertcard! : tells that user inserted cash
* Parkingticket! : issues the aprking ticket to the user
* payment? : receives the signal that payment has been done
* Leftparking? : receives the signal that the user has left parking and thus could enter again.

### Transitions:

1. **Transition from idle to CheckAvailability**

* Trigger: Signal from a car indicating a request for a parking space (check?).
* Action: Move to the CheckAvailability state to verify if there are available parking spots.

1. **Transition from Check Availability to No Payment:**

* Trigger: Vehicle Approaches and Input signal to sensor
* Guard: OccupiedSpace < MaxCapacity
* Reset: ValidationTime == 0

1. **Transition from NoPayment to Check Availability:**

* Trigger: ValidationTime
* Guard:ValidationTime > 300

1. **Transitions from NoPayment to InProcess :**

* Triggers: InsertMoney OR InsertCard
* Reset: Time = 0

1. **Transitions from InProcess to NoPayment :**
   * Trigger: payfailed?
   * Reset: ValidTime = 0
2. **Transition from InProcess to payment done :**

* Trigger: payment? channel.
* Reset: time = 0
* Action: Upon successful payment, move to paymentdone state.

1. **Transition from payment done to ticket:**

* Trigger: None
* Synchronization: parkingticket!
* Select: Ticket number from the range [0,1] (i.e., two parking spots).
* Update:
  + time = 0
  + Reset cash and card flags.
  + Assign the parking spot to the car.
  + Increment the occupiedspace counter.
* Action: Issue the parking ticket and move to the ticket state.

1. **Transition from ticket to idle :**

* Trigger: parkingticket! channel.
* Reset: time = 0
* Action: After issuing the ticket, return to the idle state.

### Invariants:

InProcess : Time < 5 sec

## VEHICLE TIMED AUTOMATA

### Local Declarations

* **carticket**: Represents the ticket assigned to the car.
* **currposition**: Current position of the car in the parking process.
* **wait**: A clock variable to measure waiting time.
* **parked**: A boolean to indicate if the car is currently parked.

### Locations:

* idle
* Start
* FindParkingSpot
* parking
* Park
* exiting
* Block
* finecleared
* Exit Gate

### Clocks:

* Wait (continuous clock representing waiting time)

### Channels:

* Parkingticket?: signal which tells if the user has received a parking ticket
* Performpark!: signal which tells if the user can park
* Entrycompleted?: signal which tells if entry into the parking has been completed by the user
* Performunpark!: signal which tells the system that the user wants to leave their space.
* Exitcompleted?: signal which tells if exit from the parking space has been completed by the user
* Leftparking! : signal issued when user leaves the parking
* Alert!: signal issued to tell that the user has overstayed their allocated time.
* Finepaid!: signal issued to tell that the user has paid their fine.

### Transitions:

1. **Idle to start:**

* From idle to start using the check! synchronization channel.
* Syncronization: check!

1. **Transition from Start to FindParkingSpot:**

* Trigger: parkingTicket?
* Select: parking space number from 1-2 at which vehicle currently is.

1. **Transition from FindParkingSpot to FindParkingSpot:**

* Guard: carticket != currposition
* Select: number of parking space adjacent to the current one.

1. **Transition from FindParkingSpot to Parking:**

* Trigger: performpark[carticket]!
* Guard: carticket == currposition

1. **Transition from Parking to Park:**

* Trigger: entrycompleted[carticket]?
* Guard:Parked = true
* Reset: wait = 0

1. **Transition from Park to Block:**

* Trigger: wait time is more than 60
* Guard: wait > 60 min
* Synchronization: alert[carticket]!

1. **Transition from Park to Exiting:**

* Trigger: performunpark[carticket]!
* Guard: wait <= 60 min

1. **Transition from Exiting to Exit\_Gate:**

* Trigger: exitcompleted[carticket]?
* Guard: parked = false
* Reset: wait == 0

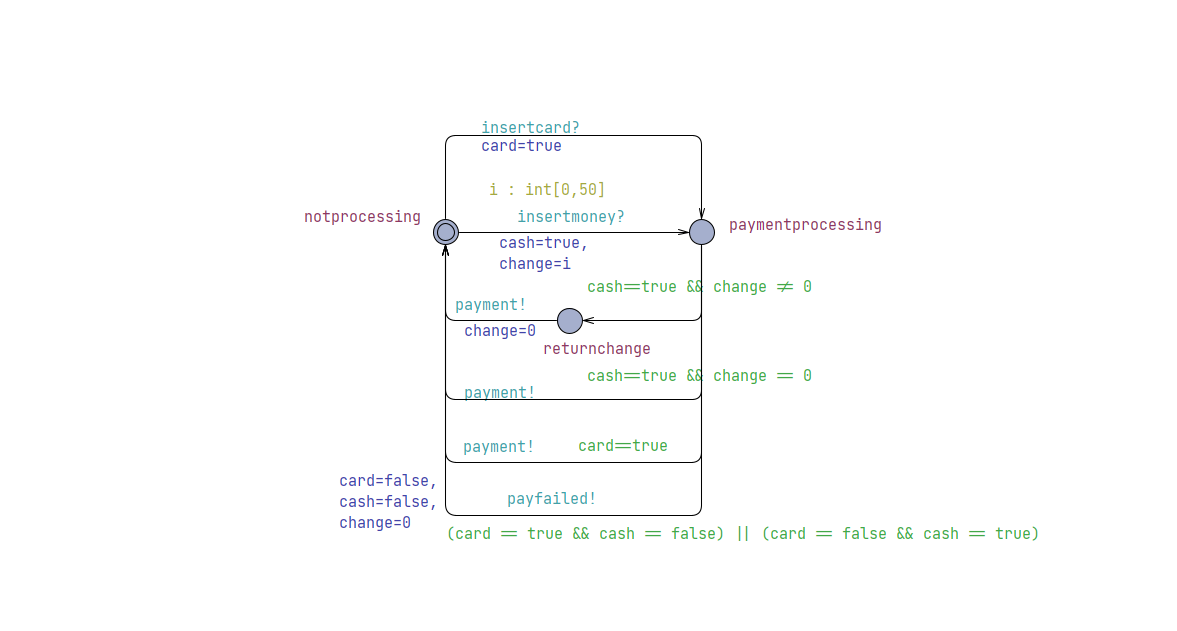
1. **Transition from Block to finecleared:**

* Trigger: Finepaid[carticket] signal from fine automata
* update: wait=0

1. **Transition from finecleared to Exiting :**

* Trigger: performunpark[carticket]!

## PAYMENT TIMED AUTOMATA

******

### Locations:

* notprocessing
* Paymentprocessing
* nochange

### Clocks:

None

### Channels:

* insertcard? : It is a signal if the user gives a card for payment.
* Insertcash?: It is a signal if the user gives cash for payment.
* Payment!: signal issued when payment is successful
* payfailed! : it is a signal emitted that the payment has failed.

### Transitions:

**1.Transition from noprocessing to paymentprocessing:**

* Triggers: insertmoney?
* Select: value of Change from 1-50
* Update: cash==true

**2.Transition from noprocessing to paymentprocessing:**

* Triggers: insertcard?
* Update: cash==true

**3.Transition from paymentprocessing to noprocessing :**

* Syncronization: payment!
* Guard: cash==true and change ==0

So this transition is basically triggered when cash payment is done but there is no change to be returned.

**4.Transition from paymentprocessing to noprocessing :**

* Syncronization: payment!
* Guard: card==true

So this transition is basically triggered when card payment is done

**5.Transition from paymentprocessing to nochange:**

* Trigger: change!=0
* Guard: cash==true and change !=0

So this transition is basically triggered when cash payment is done and there is change to be returned.

**6.Transition from nochange to noprocessing :**

* Syncronization : payment!
* Update : change = 0

So this transition is basically triggered when change is returned and ticket is issued

**7. Transition from paymentprocessing to noprocessing :**

* Trigger: failure of payment for some reason
* Guards: card == true XOR cash == true
* Reset: Change= 0, card = false and cash = false
* Effect: payfailed! Signal issued

### Invariants:

None

## Declarations

A computer screen shot of a computer code

Description automatically generated

Car local declarations

A screenshot of a computer

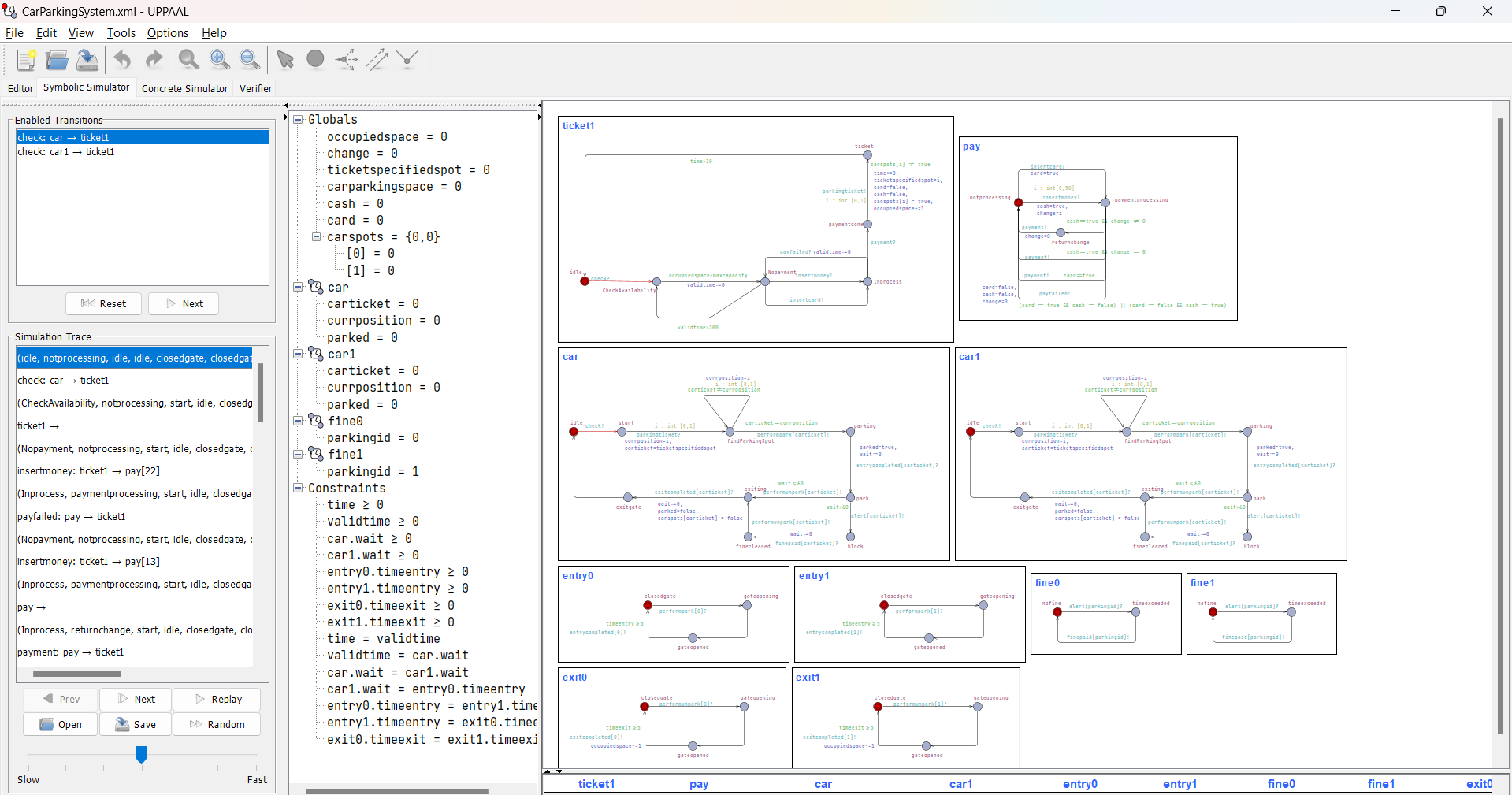
Description automatically generated

## System Declarations

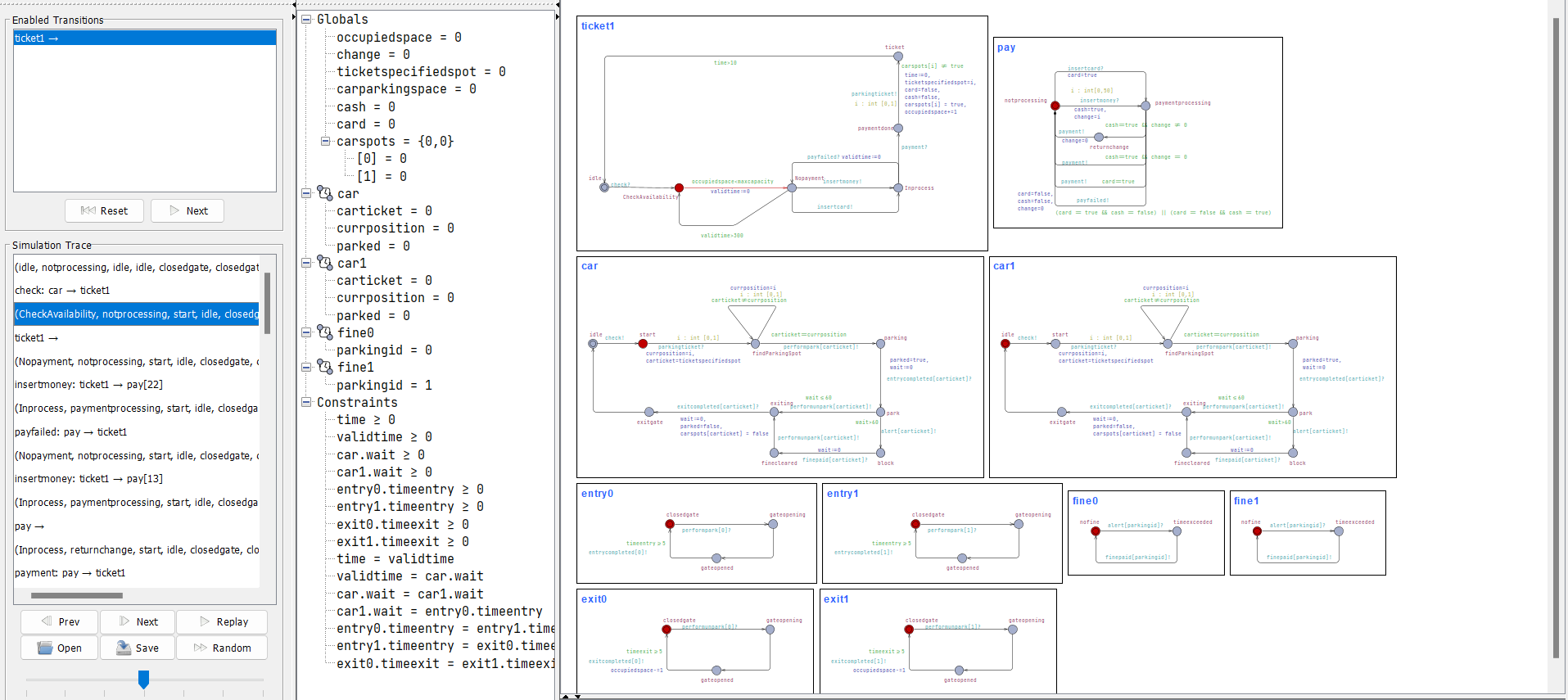
A screenshot of a computer

Description automatically generated

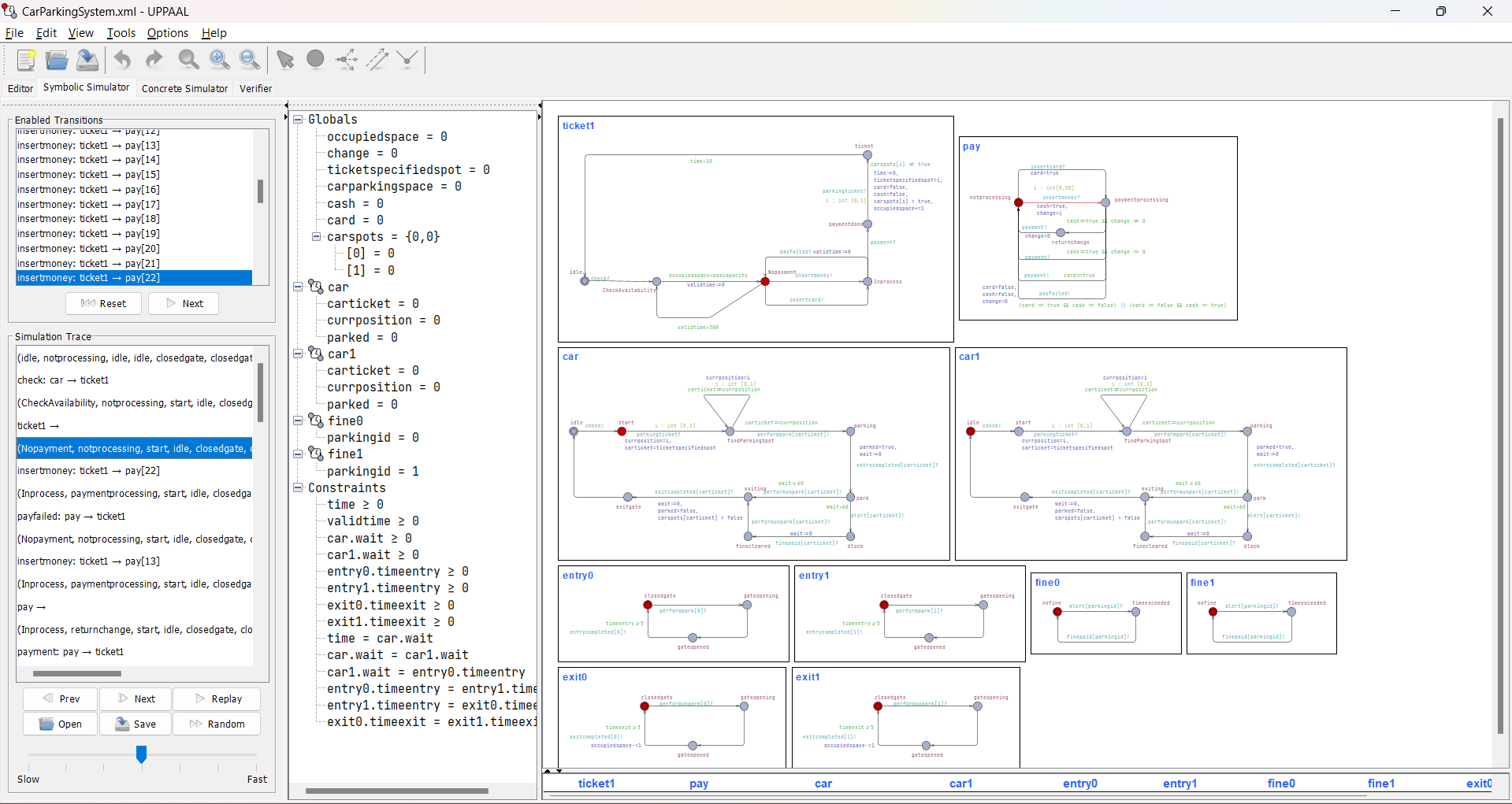
## UPPAAL Simulation



Car1 is sending signal to check for availibility.



Car1 is at the gate and checking for the availability of space



Space is available and the user moves to nopayment

A screenshot of a computer

Description automatically generated

User pays through cash and has rs 22 change to be returned.

A screenshot of a computer

Description automatically generated

Payment is failed and the user is going to retry with a different note, now change to be returned is RS 13.

A screenshot of a computer

Description automatically generated

User change is returned and payment is successful

A screenshot of a computer program

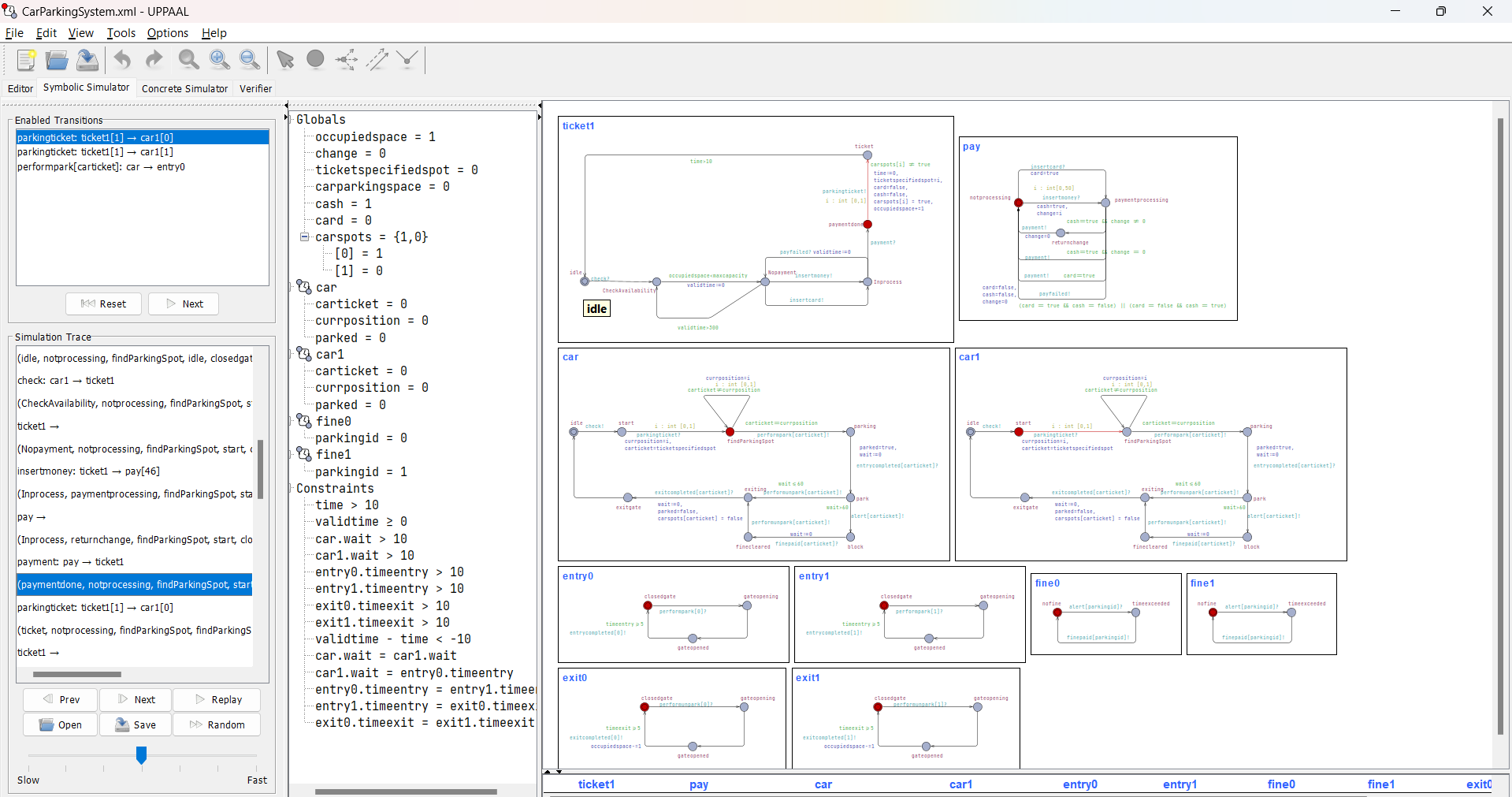
Description automatically generated

Payment module has returned to noprocessing following a successful payment and ticket module is at payment stage and generating a ticket.

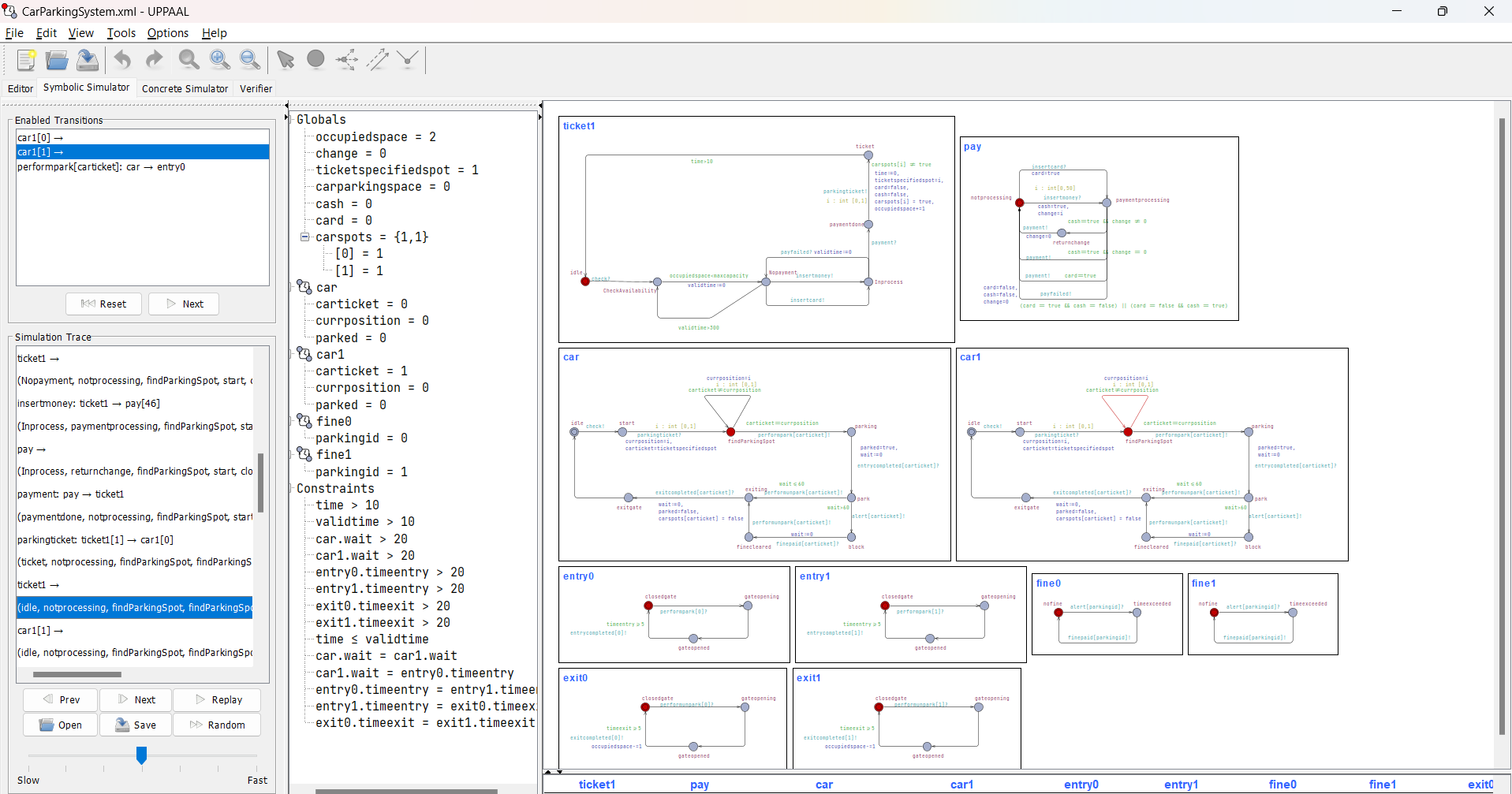
A screenshot of a computer

Description automatically generated

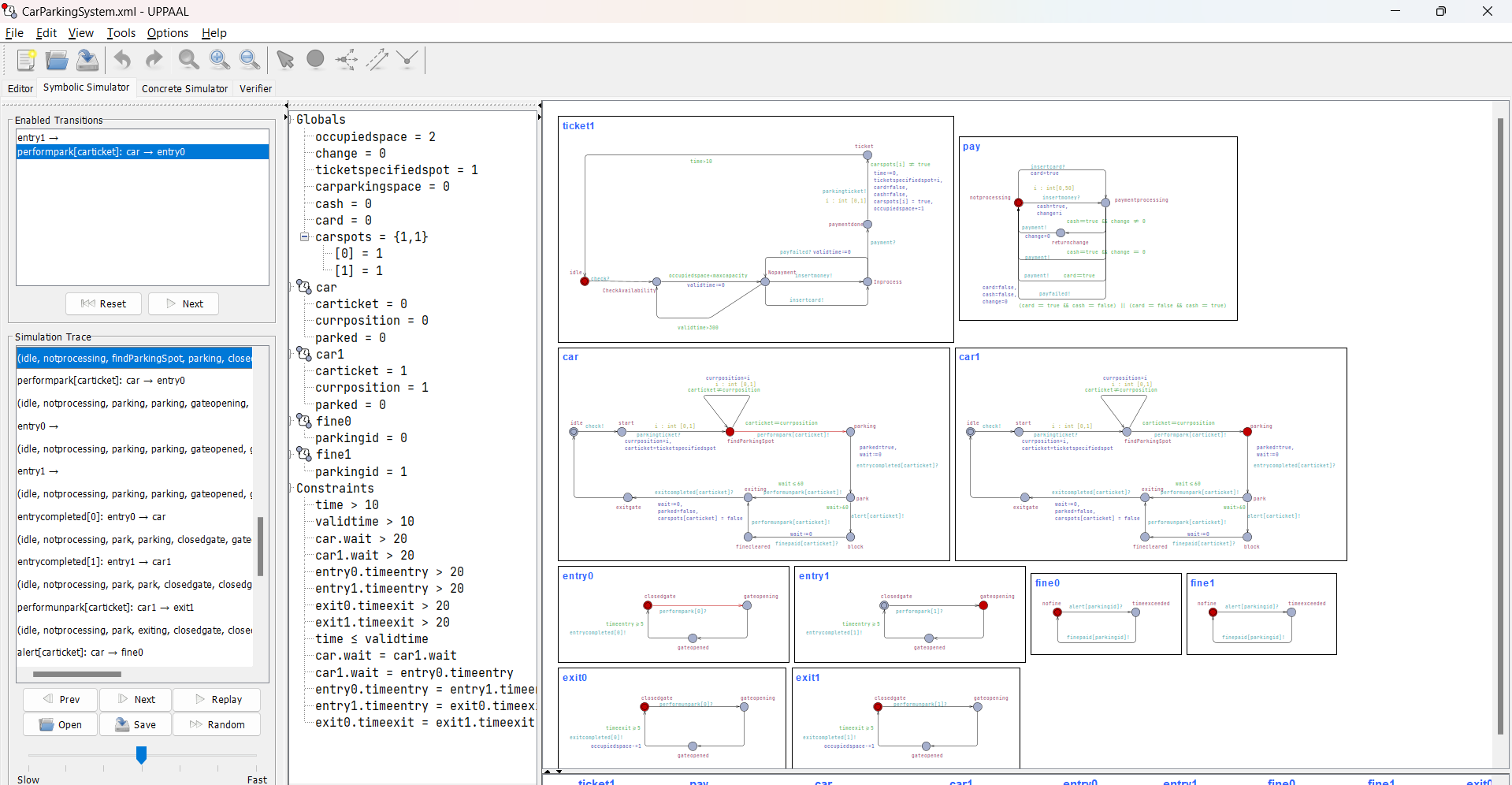
Car got ticket id 0 and is at the space, meanwhile another car is at the teicketing booth checking for space.



Second car pays for the ticket and gets a ticket of 1 as only that spot is available. It is currently at spot 0.



Second car tries to find the spot whereas first car is yet to start the parking process



Second car ahs found the spot whereas first car is parking

A screenshot of a computer

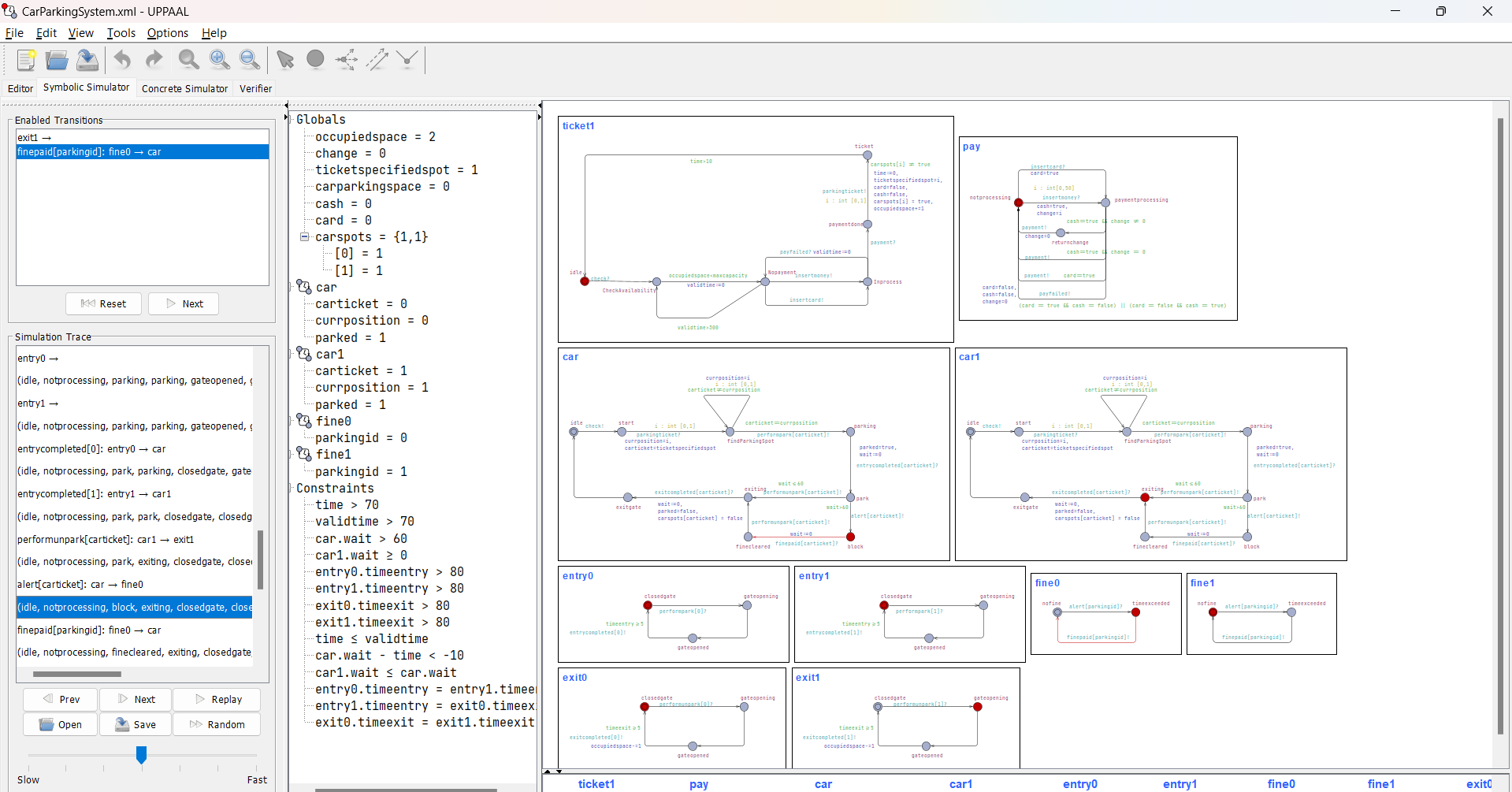
Description automatically generated

Both cars have started the parking process together and the gates for both the parking spots are opening.

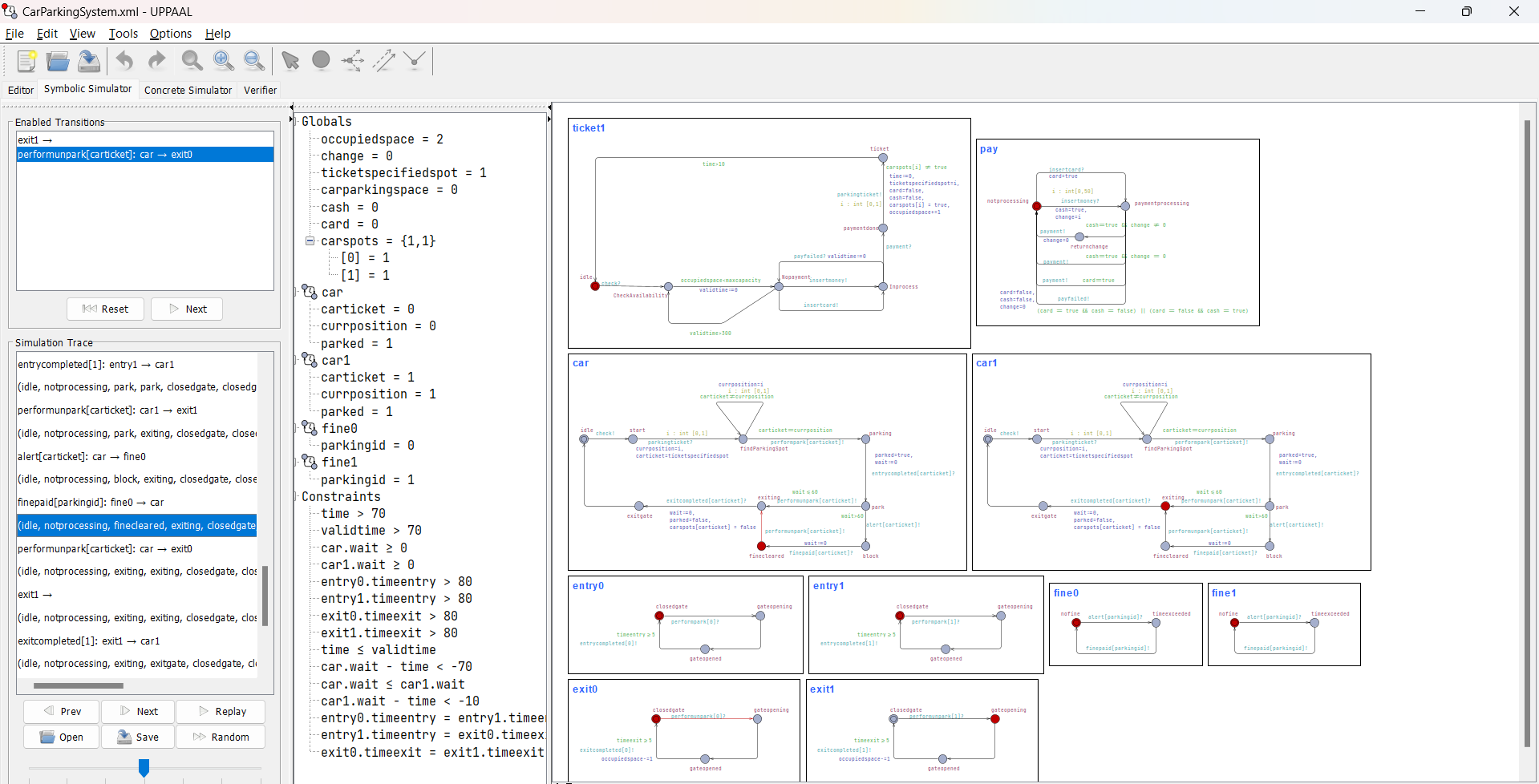
A screenshot of a computer

Description automatically generated

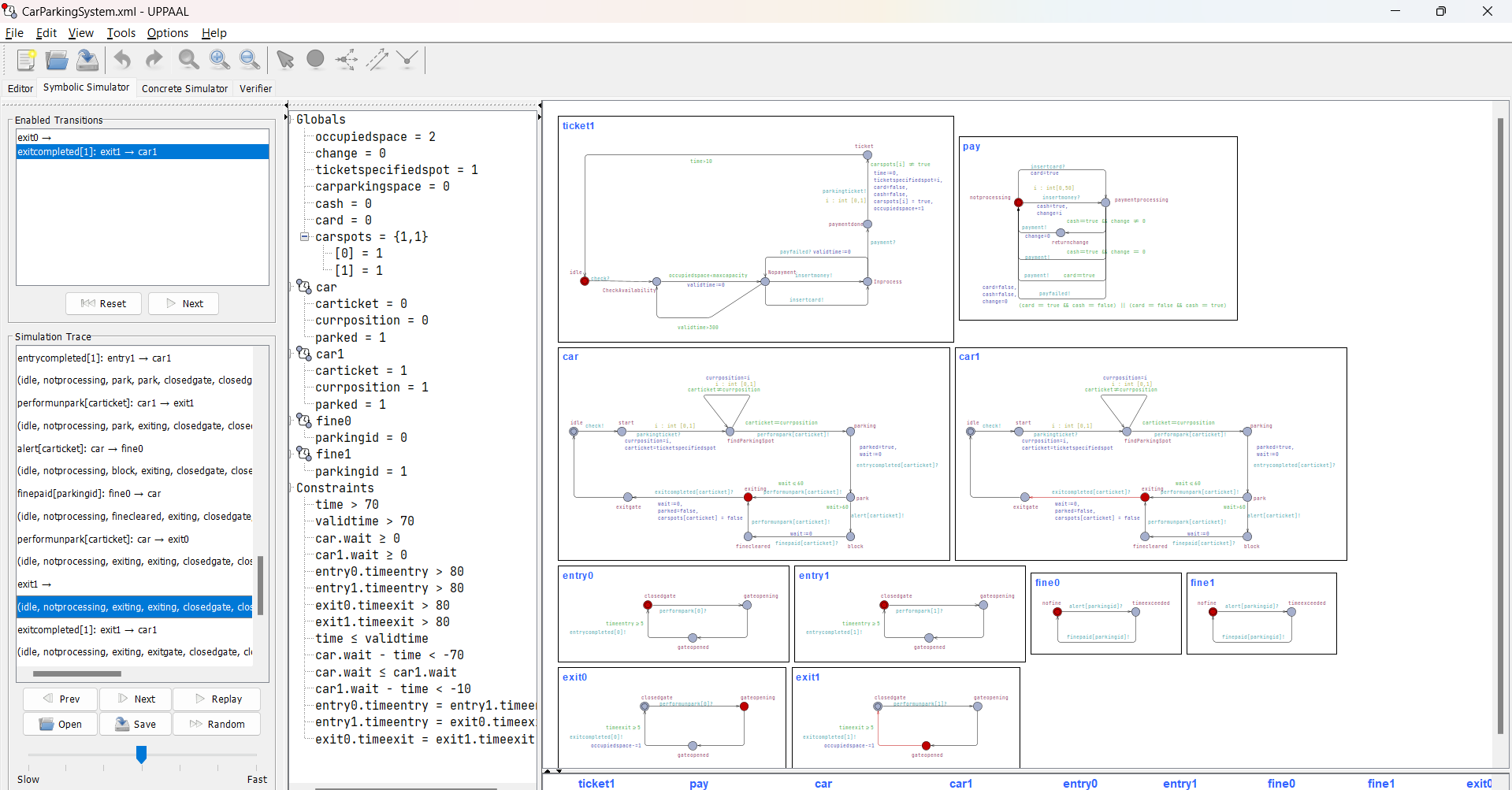
Both cars have parked successfully



Car1 has initiated the exit process and the exit gate for that spot is openeing whereas car has overstayed and is now in block state, has to pay fine



Fine has been paid whereas the other car is exiting still



Car1 parking gate has opened, wheras car has also given the signal to exit, so its gate is also openeing

A screenshot of a computer

Description automatically generated

Car1 has exited and is now in idle state whereas car is exiting still.

A screenshot of a computer

Description automatically generated

Both cars in original state.

## Swimlanes

A screenshot of a computer program

Description automatically generated A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

# Verification:

### Deadlock:

1. A[] not deadlock: Ensures the system will never enter a state where no progress can be made.
2. A[] (occupiedspace == maxcapacity imply not deadlock): The system cannot be in a deadlock state when the parking lot is full.

### SAFETY

1. A[] not (car.parking && (entry0.closedgate && entry1.closedgate)): A car cannot be in the parking state if both entry gates are closed.
2. A[] not (card == true and change != 0): Ensures that a card is not inserted while change is not zero.
3. A[] not (entry1.gateopened && exit1.gateopened): Ensures that entry gate 1 and exit gate 1 cannot be opened simultaneously.
4. A[] not ((fine1.timeexceeded || fine0.timeexceeded) && car.exiting): A car cannot be in the exiting state if any fine time has exceeded.
5. A[] not (car.exiting && exit1.closedgate): A car cannot be in the exiting state if the exit gate is closed.
6. A[] not (car.block && fine1.nofine && fine0.nofine): A car cannot be in the blocked state if there are no fines imposed.
7. A[] not (car.block && entry0.gateopened && entry1.gateopened): A car cannot be blocked if both entry gates are opened.
8. A[] not (car.parking and car.parked == true): A car cannot be in the parking state and already parked state simultaneously.
9. A[] not (pay.returnchange and change == 0): Ensures that the system cannot be in the return change state if there is no change to return.
10. A[] not (pay.paymentprocessing and (cash == false and card == false)): The payment processing state cannot occur if neither cash nor card is inserted.
11. A[] not (cash == true and card == true): Both cash and card cannot be inserted at the same time.
12. A[] not (car.currposition != car.carticket and car.parking): Ensures that the car’s current position matches the car’s ticket when in the parking state.
13. A[] not (car.carticket == car1.carticket && car.parked == true): Ensures that two cars cannot have the same ticket and be parked at the same time.
14. A[] not (occupiedspace > maxcapacity): The occupied space cannot exceed the maximum capacity of the parking lot.

### Utility:

1. A<> ((card == true || cash == true) imply ticket1.ticket): If a card or cash is inserted, a ticket will be issued.
2. A<> (ticket1.Inprocess imply ticket1.paymentdone): If the ticketing process is ongoing, it will eventually be completed with a payment done.
3. A<> (car.parking imply car.exiting): If a car is in the parking state, it will eventually transition to the exiting state.
4. A<> (ticket1.paymentdone imply (car.parking || car1.parking)): Once the payment is done, either car or car1 will be in the parking state.
5. A<> (car.findParkingSpot imply car.parking): If a car is finding a parking spot, it will eventually be in the parking state.
6. E<> (entry1.gateopened imply car.exiting): If entry gate 1 is opened, the car will eventually be in the exiting state.
7. E<> (exit1.gateopened imply car.idle): If exit gate 1 is opened, the car will eventually reach the idle state.
8. E<> (car.block && fine1.timeexceeded imply car.finecleared): If a car is blocked and the fine time has exceeded, the car will eventually have its fine cleared.

### Reachability:

The system reaches the initial state:

1. E<>car1.start

The system reaches all the states:

1. E<>car.block
2. E<>car.park
3. E<>car.findParkingSpot
4. E<>car.parking
5. E<>car.exiting
6. E<>car.exitgate
7. E<>car.finecleared
8. E<>car.start
9. E<>fine1.nofine
10. E<>fine1.timeexceeded
11. E<>ticket1.CheckAvailability
12. E<>ticket1.Nopayment
13. E<>ticket1.Inprocess
14. E<>ticket1.ticket
15. E<>ticket1.paymentdone
16. E<>pay.notprocessing
17. E<>pay.paymentprocessing
18. E<>pay.returnchange
19. E<>entry1.closedgate
20. E<>exit1.closedgate
21. E<>entry1.gateopening
22. E<>exit1.gateopening
23. E<>entry1.gateopened
24. E<>exit1.gateopened

### LIVENESS

1. E<> (car.parking && (entry1.gateopening || entry0.gateopening)): If a car is parking, either entry gate (entry1 or entry0) will eventually start opening.
2. E<> (car.exiting && exit1.gateopening): If a car is exiting, the exit gate will eventually start opening.
3. E<> (car.block && fine1.timeexceeded): If a car is blocked and the fine time has exceeded, the fine automata state for fine to be implemented will eventually be reached.
4. E<> (car.finecleared && fine1.nofine): If a car has cleared its fine, it will eventually reach a state where no fine is imposed.

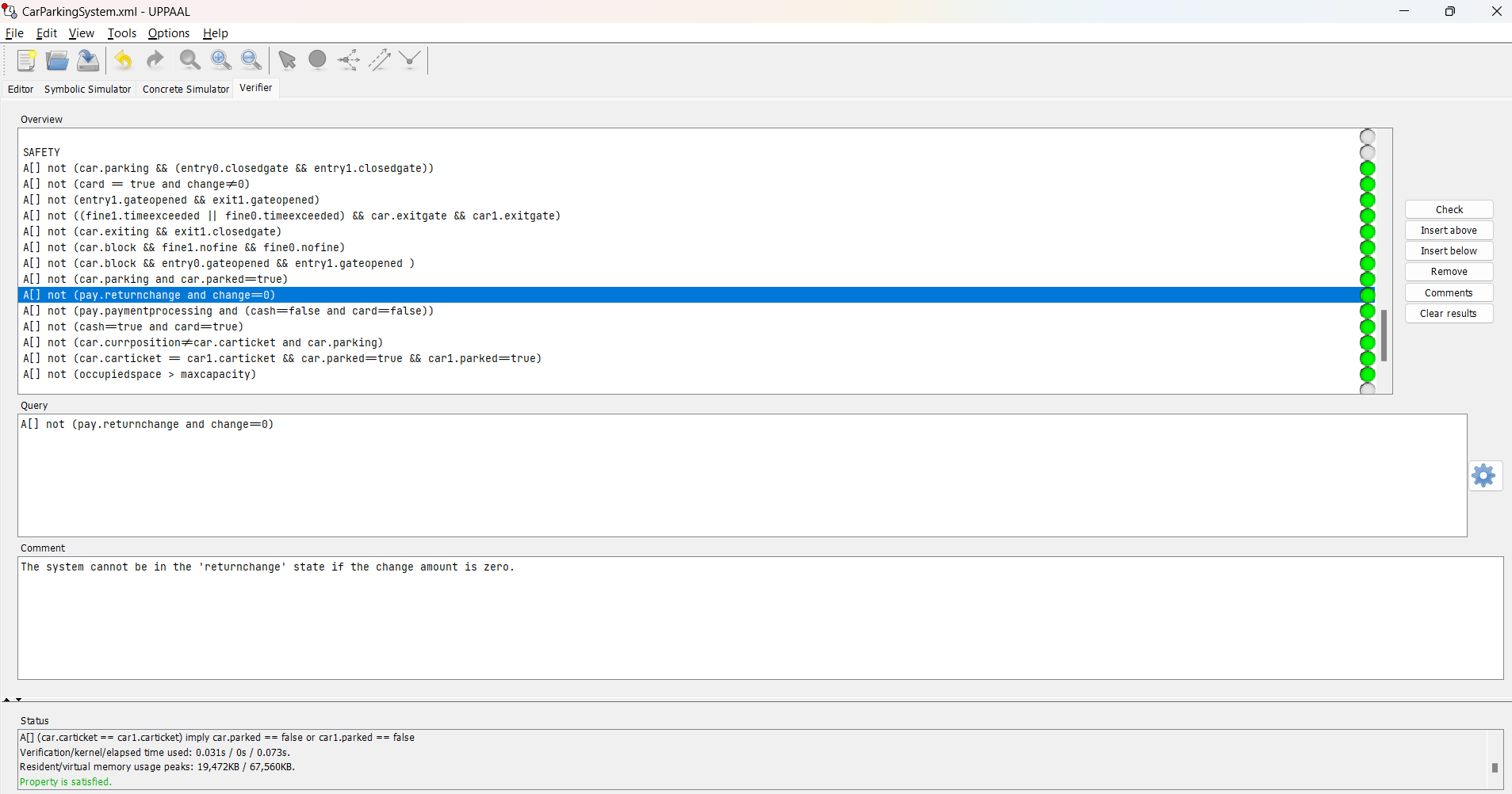
### MUTUAL EXCLUSION:

1. A[] ((ticket1.paymentdone && occupiedspace == 0 && car.parked == false && car1.parked == false) imply ((car.start || car1.start) && exit1.closedgate && exit0.closedgate && entry0.closedgate && entry1.closedgate && fine1.nofine && fine0.nofine)): If the ticket payment is done and the occupied space is zero, and neither car nor car1 is parked, then either car or car1 should be in the start state with all gates closed and no fines imposed.

### Fairness

* E<> (ticket1.Inprocess imply car.parked): This property states that whenever a car is in the state of processing a ticket (ticket1.Inprocess), it will eventually reach a state where it is parked (car.parked). This ensures that the ticketing process will be completed, and the car will find a parking spot.
* E<>(car.exiting imply exit1.gateopened): This property guarantees that whenever a car is in the process of exiting (car.exiting), the exit gate (exit1.gateopened) will eventually open to allow the car to leave. This ensures that cars can leave the parking area without getting stuck.
* A[] (car.block imply car.wait > 60): This property asserts that if a car is in a blocked state (car.block), it means that it has been waiting for more than 60 seconds (car.wait > 60). This ensures that a car is only considered blocked if it has been waiting for a significant amount of time, preventing false positives of blockages.
* A[] (parked==true imply (car.park or car.block or car.finecleared or car.exiting)): This property ensures that if a car is marked as parked (parked==true), it is either in the process of parking (car.park), is blocked (car.block), or has received a fine (car.fine). This comprehensive check ensures that a parked car's state is consistently monitored and accounted for in one of the three possible conditions.

## Verification in uppal:



# CONCLUSION

In conclusion, the application of timed automata modeling to the smart car parking system at NUST has yielded valuable insights into its operation, performance, and potential for optimization. By leveraging the formalism of timed automata, we have been able to capture the system's temporal dynamics with precision, enabling thorough analysis and verification of its behavior under various scenarios. Our findings underscore the effectiveness of timed automata as a modeling tool for complex real-world systems, facilitating the identification of bottlenecks, optimization opportunities, and resilience-enhancing measures.