


# SCORE 2011 Contest

## Project Summary Report

Transport4You: an intelligent public transportation manager

Team Members: Li Yi (undergraduate)  
[liyi@comp.nus.edu.sg](mailto:liyi@comp.nus.edu.sg)  
Yang Hang (master by course)  
[winterno1@hotmail.com](mailto:winterno1@hotmail.com)  
Wu Huanan (master by course)  
[wuhuanan1119@gmail.com](mailto:wuhuanan1119@gmail.com)



## Table of Contents

<b>1. Introduction .....</b>	<b>3</b>
<b>2. Development Process .....</b>	<b>3</b>
<b>3. Requirements: Problem Statement .....</b>	<b>4</b>
<b>4. Requirements Specification .....</b>	<b>5</b>
4.1 Entity Relationship .....	5
4.2 Requirement Elicitation .....	6
4.3 Functional Requirements.....	6
4.4 System Quality Requirements.....	7
4.5 Use Case Model.....	7
4.6 Requirement Definitions.....	8
<b>5. Design .....</b>	<b>9</b>
5.1 Architecture Overview .....	9
5.2 Formal Design Model.....	9
5.3 Detailed Design .....	13
<b>6. Implementation .....</b>	<b>14</b>
6.1 Strategies of Fault Detection and Correction.....	14
6.2 Methodology of User Behavioral Analysis .....	16
6.3 Experiment on Bluetooth Device Detection.....	18
6.4 Using PAT as Trip Planning Service.....	18
6.5 Implementation Details of Server Core.....	19
6.6 Implementation Details of UI and Simulators .....	20
<b>7. Management Plan.....</b>	<b>21</b>
7.1 Team Introduction.....	21
7.2 Communication.....	21
7.3 Risk Management.....	22
7.4 Artifacts Management .....	22
<b>8. Project Plan.....</b>	<b>22</b>
8.1 Development Plan.....	22
8.2 Activity Plan.....	23
<b>9. Verification and Validation .....</b>	<b>23</b>
9.1 Testing by Model Checking .....	23
9.2 Traditional Testing .....	25
<b>10. Lessons Learned .....</b>	<b>25</b>
<b>Works Cited .....</b>	<b>26</b>

## 1. Introduction

This is a summary report for the software engineering project, Transport4You, which is intended for the SCORE 2011 Contest. This report summarizes the techniques and approaches used in the project, and also provides information on the software development process and project management.

Formal methods are extensively applied in our project throughout the whole development process. In addition, a sophisticated and self-contained model checker, Process Analysis Toolkit (PAT) (Sun, Liu, Dong, & Pang, 2009) (Sun, Liu, Roychoudhury, Liu, & Dong, 2009), is used at both design and testing stages. The application of formal methods, especially formal modeling and verification, elevates the design quality and confidence of the development of real-life transportation management system.

This report is organized as follows; Section 2 deals with the development process we used in the project, followed by problem statement and requirement analysis in Section 3 and Section 4. Section 5 and 6 discuss system design and implementation. After that, the management plan and project plan are provided in Section 7 and 8. Finally, we will leave the validation, verification and the lessons learned to the last two sections.

## 2. Development Process

The Transport4You intelligent public transportation manager system has some rather new concept and features compared to the existing systems. Our understanding of the system is evolving as more and more analysis of the system is done during the development process. Some requirements specifications and designs in the initial iteration may not be applicable in the latter iterations and need to be modified. Therefore, we choose an iterative and incremental development process to better fit this particular project. We divide the software development process up into iterations. Modifications and increments are applied to the previous version in each iteration, till the project is completed. Therefore, after initial planning and preliminary requirement analysis, we implement a minimal system with some fundamental features. Based on that, we constantly review and extend the system model as well as the design according to team discussions and test runs. For example, to solve the over detection problem discovered in the design stage, we modify the system requirements on the timing of bus fare deduction to accommodate our solution after consulting the project stakeholders.

In the design stage, we construct a design model that captures the main structural and functional properties of the system. The model is first informally represented in UML, using class diagrams and sequence diagrams to capture the important system characters and behaviors. The UML model is then converted into formal CSP-OZ (Fischer, 1997) specification, which is able to precisely describe the patterns of interactions among different software components in an object-oriented fashion. After that, the CSP-OZ model is translated into CSP#, which is the modeling language of PAT. The CSP# model is then verified in the

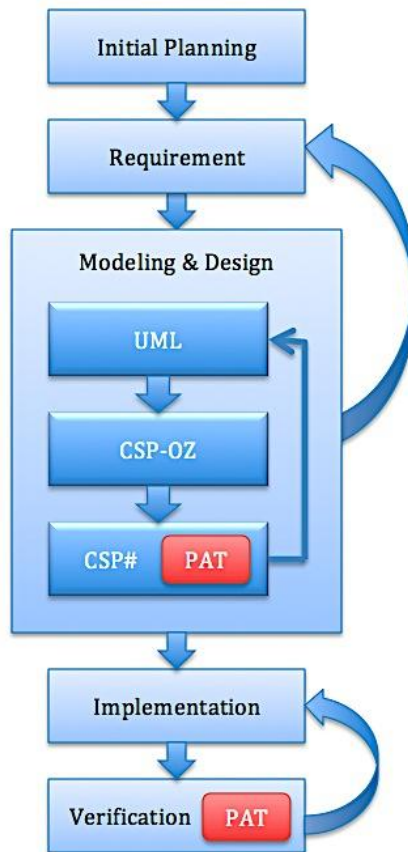


Figure 1 Development Process

model checker and provides us accurate evaluations on the stability and reliability of the software system. According to the verification results, we then modify the UML design model and repeat the process until all the required behavioral and functional goals are achieved.

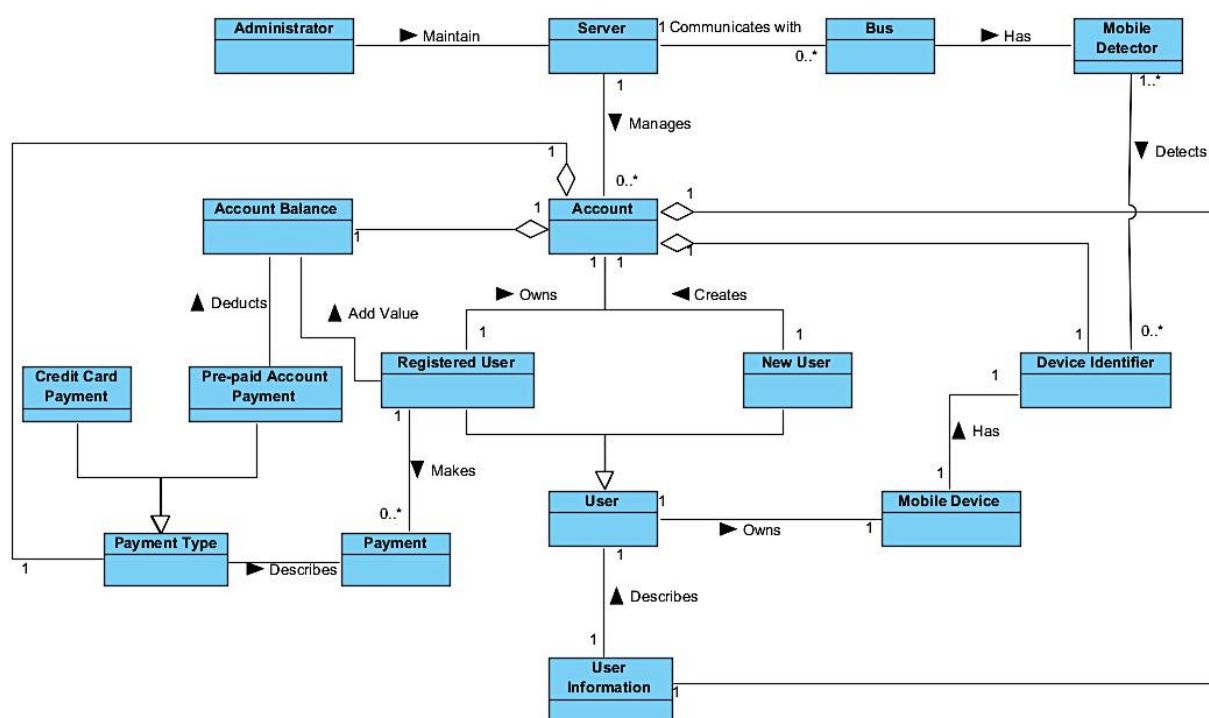
Our implementation strictly follows the CSP-OZ design specifications to ensure that the actual product preserves the same properties as the abstract model. In verification and validation stage, PAT is also used to generate test cases and execute actual code. The execution results are then compared to the expected outcomes. The detailed descriptions of this approach are discussed in the later part.

### 3. Requirements: Problem Statement

Up until the 1970s and into the early 1980s, conductors were a common feature of many local bus services in larger towns and cities in the UK and Ireland (contributors, 2010). Conductors can still be found on buses in many countries even today. However, with the fast development of information and communication technologies, automated fare collection system and many types of digital tickets have replaced bus conductors, which greatly enhances the efficiency and accuracy of fare collection. To further improve the service quality and effectiveness, many municipal transportation authorities are seeking new public transportation management solutions.

To be specific, a system that is able to provide customized trip information and timely responses to each subscriber is to be built to satisfy the increasing needs. In other words, the new system should not only play the role of a bus conductor, but also be a trip advisor who informs the users of changes in the lines and possibly suggests the optimized route for them. Moreover, to free passengers from troublesome boarding procedures, the system should not require users to do any additional actions, such as, tap a card or insert a ticket. The system should also be reliable and fault-tolerant in the sense that no major glitches should occur during the normal operating hours and no serious consequences should be caused to avoid any loss or inconvenience of users. In addition, the system must be easily extendable to larger scales to fit various situations in different cities.

## 4. Requirements Specification



### Figure 2 Entity Relationship Diagram

## 4.1 Entity Relationship

It is a good starting point to introduce various entities involved in the transportation management system. To begin with, there are two kinds of users, namely new user and registered user. New users can create their account online to become registered. Only registered users have the privilege of boarding on buses and receiving personalized trip information. Administrators are able to access the data stored on the server and maintain as well as manage server configurations. Mobile detectors installed on every bus are able to detect the signal of mobile devices carried by nearby passengers. The unique device identifier that is stored in the account captures the identity of a registered user. Moreover, there are also two types of payment. User can pay the fare either by credit card or deducting directly from the prepaid account balance.

## 4.2 Requirement Elicitation

We obtain our requirement list by studying the project description in the first place. We list out all the relevant functional and non-functional requirements that appear in the description document. After that, we discuss the priority and relevance of each requirement in our team meetings. We also reference to some existing public transportation management systems, such as the “iris” (Intelligent Route Information System, more information can be found at <http://www.sbstransit.com.sg/iris/overview.aspx>) of SBS Transit, Singapore. IRIS provides many useful services to its users, such as, “iris NextBus” that allows users to find the arriving time of the next bus, “iris Journey Planner” that plans route given the start location and the destination, “SMS service” that sends subscribers useful information, etc. We particularly selected journey plan as one of our software requirements because it meets the goal of this project very well, which is providing customized services to each user. The “SMS service” coincides with the requirement in the project descriptions. Thus we also include this service as one of our requirements. In addition, we have frequent communications with the project stakeholder to make sure that the requirement elicitation meets the needs of the client.

## 4.3 Functional Requirements

### General Requirements

The goal of the project is to develop a distributed system residing on both the central server and bus embedded systems that deducts bus fare and provides tailored information to subscribers. The basic services of the system are user boarding and alighting detection, user trips logging, automatic fare deduction, account balance top up, user registration, personal information management, and related users notification upon road conditions updated. The system should be hosted in three locations: the central server, client devices (including cell phones and other mobile devices with Wi-Fi or Bluetooth enabled) and bus embedded systems.

### Fault Correction Requirements

In practice, the Bluetooth and Wi-Fi signals are beyond the physical boundaries of a bus. Thus, a detector on bus can detect the users outside the bus and even users on another bus that is nearby. Therefore, when the central server receives a bus detection report, it should try its best to correct the erroneous information that might occur during the detection. For instance, there may be multiple buses report a same subscriber is on board at the same time.

### User Behavioral Analysis Requirements

In general, a user tends to have his or her own traveling patterns, which means he or she takes some buses more often than the others. The system should analyze trip logs to determine what the users’ regular routes are. This not only could help service providers re-arrange bus lines to meet the changing needs of the public, but also is useful when related users need to be notified because of occasional bus line interruptions.

## 4.4 System Quality Requirements

### Simulation

In the case of this project, it is impractical to implement a real distributed system of such a scale. The testing and demonstration of the system will be impossible if a large number of bus lines and passengers are involved. Therefore, a simulator is to be built for the demonstration purpose. The simulator simulates the activities of buses and passengers, as well as the interactions among them.

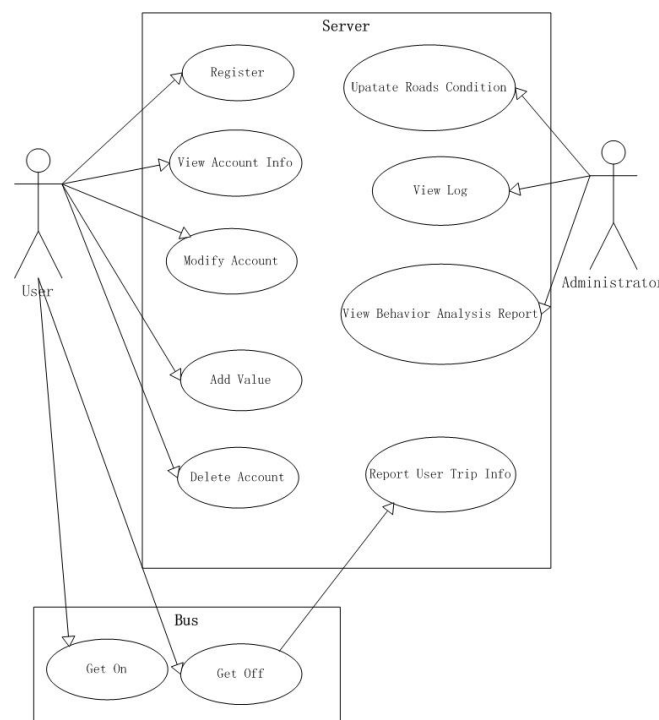
### Usability

In both design and implementation, we should keep in our minds that the operations of users should be kept to a minimum. Additionally, it should be very easy to learn how to use the system. Hence, the user interface of the system should be intuitive and concise.

### Avoid over charge

In any case, including system failures and undetermined situations, the subscribers should not be over charged by the system. The benefits of subscribers should be valued the most.

## 4.5 Use Case Model



**Figure 3 Use Case Diagram**

There are mainly two actors directly interacting with the system, namely user and administrator. The system also contains two sub-systems, located on server and bus respectively. User interacts with both the server system and the bus system while administrator only manages and maintains the server system.

Due to the limited space, the detailed use case descriptions are not shown here. They can all be found in our requirement specification documents.

#### 4.6 Requirement Definitions

We categorize the requirements into five requirement groups as shown below.

**Table 1 Requirement Groups**

Identification	Requirement Group
CORE	Main Application Core
UIM	User Information Management
SIM	Simulator UI and Animation
FC	Fault Correction
UBA	User Behavior Analysis

The following table lists all the requirements sorted by their requirement groups.

**Table 2 List of Requirements**

Identity	Description	Host
	Main Application Core	
CORE-1	Detecting Users	Bus
CORE-2	User boarding/alighting report	Bus
CORE-3	E-ticket Maintains	Server
CORE-4	T_COST and V_VAL setting	Client/ Server
CORE-5	User Account Balance Operation (add value, charging)	Client/ Server
CORE-6	User Notification (via SMS)	Server
	User Information Management	
UIM-1	New User Registration	Client/Server
UIM-2	User setting and personal information modification	Client/Server
UIM-3	User Unregistration	Client/Server
	Simulator UI and Animation	
SIM-1	Animation of Simulator	Simulator
SIM-2	Simulator UI	Simulator
SIM-3	Server basic functions and Communication between server and simulator	Simulator/Server
	Fault Correction	
FC-1	Level-1 Correction [Section 6.1.1]	Bus
FC-2	Level-2 Correction [Section 6.1.2]	Server
	User Behavior Analysis	
UBA-1	User Trips Logging	Server
UBA-2	User Trips Time Distribution Statistics	Server
UBA-3	User Behavior Analysis (with bottom line setting) [Section 6.2]	Server



## 5. Design

### 5.1 Architecture Overview

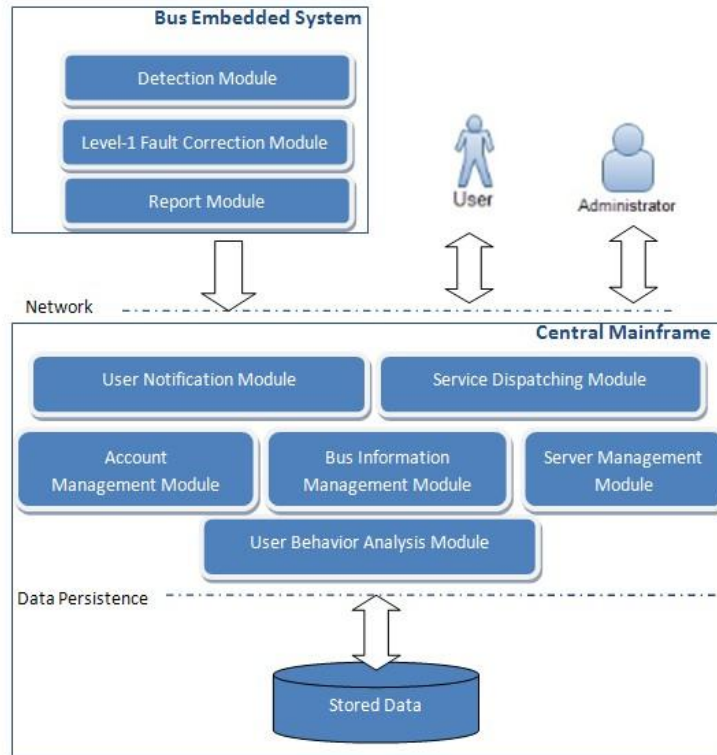


Figure 4 Architecture Design

As shown in Figure 4, the Transport4You system consists of two sub systems, namely the bus embedded system and the central mainframe. The bus system is responsible for passenger detection, part of the fault correction and detection results report to the central server. In contrast, the server system deals with all kinds of service requests from users and administrators, information management, as well as user notification. The two sub systems communicate via TCP connections and at the same time interact with users and administrators.

### 5.2 Formal Design Model

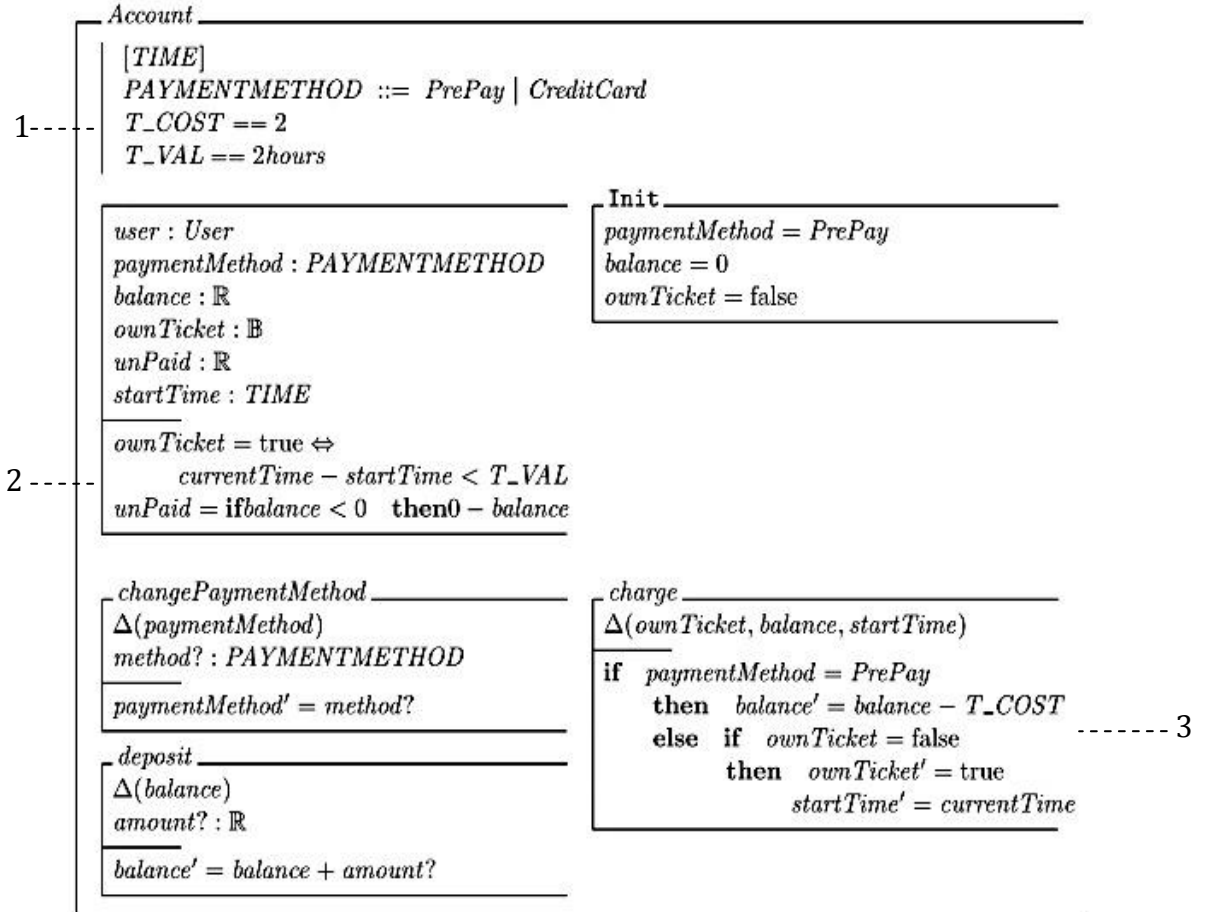
In this part, we apply CSP-OZ to specify our UML design and translate it into the modeling language CSP# which can be verified in the model checker PAT. CSP-OZ, blending of Object-Z and CSP, is to associate an Object-Z class with a failure divergence semantics of CSP, the main idea is to transform Object-Z classes to CSP processes and to compose them by arbitrary CSP operators. The reason of using this specification language mainly relies on three aspects: First, CSP-OZ not only can help us to specify both static and dynamic aspects of the system, but also guide our implementation in a better way because of its Object Oriented features. Second, this approach relates UML design (class diagrams and sequence diagrams), CSP# model in PAT and the implementation with each other and therefore smoothly connects the whole development process. Third, CSP# includes data operations and shared variables which is a combination of CSP and simple Z, thus there is no gap between CSP-OZ and CSP# when doing the

translation. Due to space limitations, here we only pick the class *AccountManager* as an illustrating example.

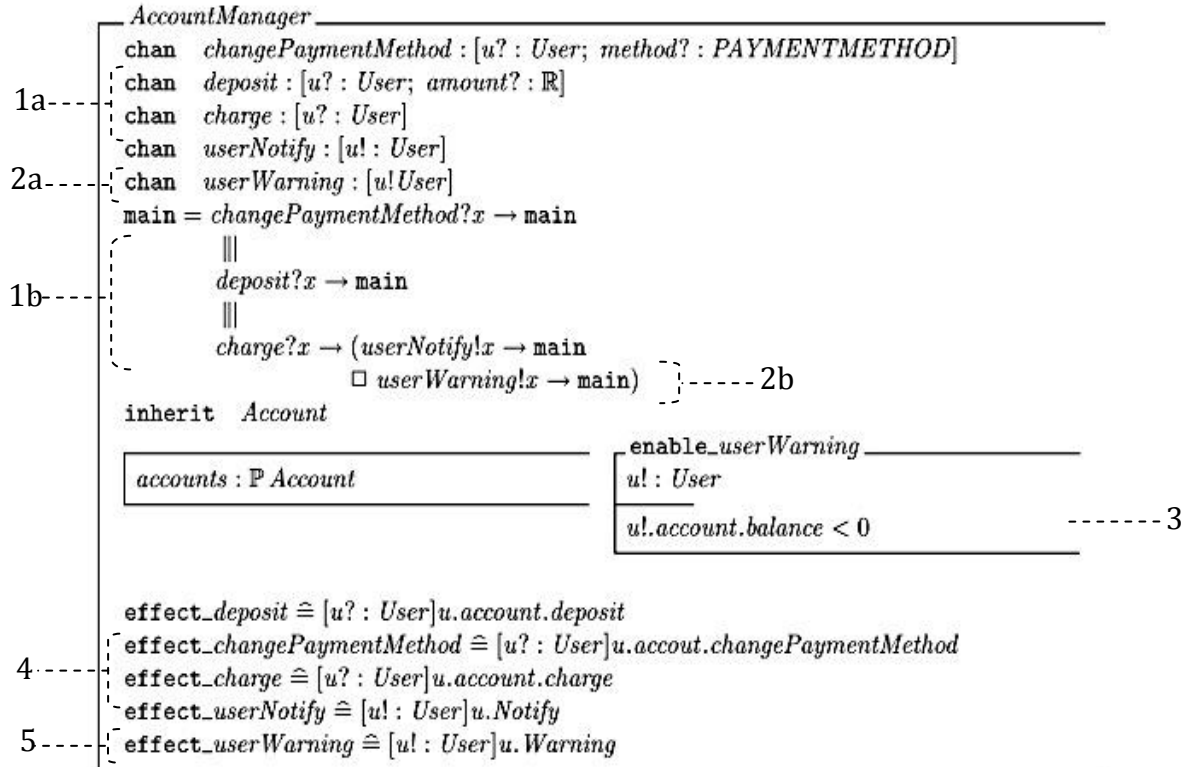
### CSP-OZ Specification

According to our design, when there is a request of charging a user, the request will be sent to the class *AccountManager*. In addition, when a user requests to change his or her payment method or to make a deposit to his or her account, the requests are also sent to *AccountManager*.

The detailed descriptions of the class can be seen as follows.



1. User can choose whether to pay by prepay account or by credit card.  
 T\_COST indicates the charge fee for every trip.  
 T\_VAL indicates the validity period for every ticket generated.
2. The property *ownTicket* is a bool type. A ticket will be expired after T\_VAL period since it is generated.  
 The property *unPaid* is the balance due in the account.
3. When a charging request is received, the system will first check the payment method of the user, if he/she pays by prepay account, T\_COST money will be deducted from the account, if paying by credit card, the system will check whether there exists a valid ticket, if not, T\_COST amount of money will be deducted from the credit card and a ticket will be generated automatically.



1. The class *AccountManager* is mainly responsible for **(b)** dealing requests related to user account through **(a)** corresponding interfaces.
2. *AccountManager* is also in charge of sending user notifying and user warning requests.
3. When a user's account balance is inadequate, a *userWarning* will be sent to him/her.
4. As *AccountManager* inherits the class *Account*, the operations related to account are the same as they are in class *Account*.
5. These two operations are in another class *UserNotification* that will not be discussed here for the limited space.

### CSP# Model

As mentioned before, the CSP# language can be treated as a combination of CSP and simple Z which is very similar to CSP-OZ. So the translation from CSP-OZ to CSP# is straightforward. Here, we still take the *AccountManager* class as an example, the translated CSP# language is as follows.

```

1 channel ch_changePaymentMethod 99;
2 channel ch_deposit 99;
3 channel ch_charge 99;
4 channel userNotify 99;
5 channel userWarning 99;
6 AccountManager() = ch_changePaymentMethod?user.method -> AccountManager()
7                   |||
8                   ch_deposit?user.amount -> AccountManager()
9                   |||
10                  ch_charge?user -> AccountManager();
11 Charge(user) = ch_charge!user ->
12               ifa(!user.HasTicket()){
13               ifa(user.IsPayByCreditCard()){
14                   deductFromCC{user.SetHasTicket(true)}
15                   -> ch_userNotify!CreditCard.user
16                   -> Skip}

```

```

17         else{
18             deductFromPA{user.ChargeUser(T_COST)}
19             -> ch_userNotify!PrePay.user
20             -> ifa(user.Balance()<T_COST){
21                 ch_userWarning!Warning.user
22             -> Skip}}}][]Skip;

```

Codes from line 1 to line 5 are the channel definitions, which are written according to the channel declaration part of the CSP-OZ schema. The part from line 6 to line 10 is the main process of the class *AccountManager* and the remaining part is the function *Charge()* whose procedure are exactly the same as that of the *Charge()* function in the CSP-OZ schema. Although the translation is done manually, the process is straightforward. We can translate all the other CSP-OZ schemas to CSP# in the same way. It means that we can thus verify the CSP-OZ specification in PAT!

### Verification

PAT supports a number of different assertions that are queries about the system behaviors. It supports the full set of Linear Temporal Logic (LTL) as well as classic refinement/equivalence relationships. After we put the whole model into PAT, we set a series of properties to check.

Property	Assertions in PAT	Result
Deadlock free	#assert System() deadlockfree;	✓
Response	//A user must be informed after getting on a bus #assert System()  = [] (getonbus -> <> (ch_SMS!1    ch_SMS!0    ch_SMS!2));	✓
	//A user must be warned when balance is insufficient #define state_bns user_prepay_acc<=0; #assert System()  = <> (state_bns -> <> ch_SMS!3);	✓
Reliability	//User may not be charged when he/she owns a ticket #define state_ot user_ticket == true; #assert System()  = [] !(state_ot && (deduct_PP    deduct_CC));	✗ (fixed)
	//A user's account will be increased as much as user's deposit #define state_bd user_prepay_acc>=deposite; #assert System()  = [] (ch_acc_rec!Deposit -> <> state_bd);	✓

After verification, we found a design bug in the function *Charge()*, as specified by the schema *Charge*. That is, if the user already holds a valid ticket but changes his payment method from credit card to account balance, the system will still charge him or her. The cause of this bug is that we first check the payment method instead of checking whether the user already holds a valid ticket.

Next, we extract a small part of the system as an illustrating example to show our design process. More specifications and verifications about the system can be found in the detailed documentations.

## 5.3 Detailed Design

### Static Design

At the static design stage, we represent the relationships among all sub-systems that are described in the architectural design by an all-in-one class diagram. The class diagram depicts the structure of the system in details. It directly instructs the implementation work, as the components in the diagram can be transferred into classes in the real code.

As shown in Figure 5, we divide the system into three parts as is specified in the architectural design. The “Log Manager” class corresponds to the Log component in the architectural design diagram. Among other classes, the abstract interface, “IDeviceDetect” is defined for extendibility purpose such that other hardware detectors can be supported in the future. What also need to be mentioned here is that the hardware IDs of the mobile devices can be detected within a certain range and collected as a set of strings, which we have already experimented on and proved.

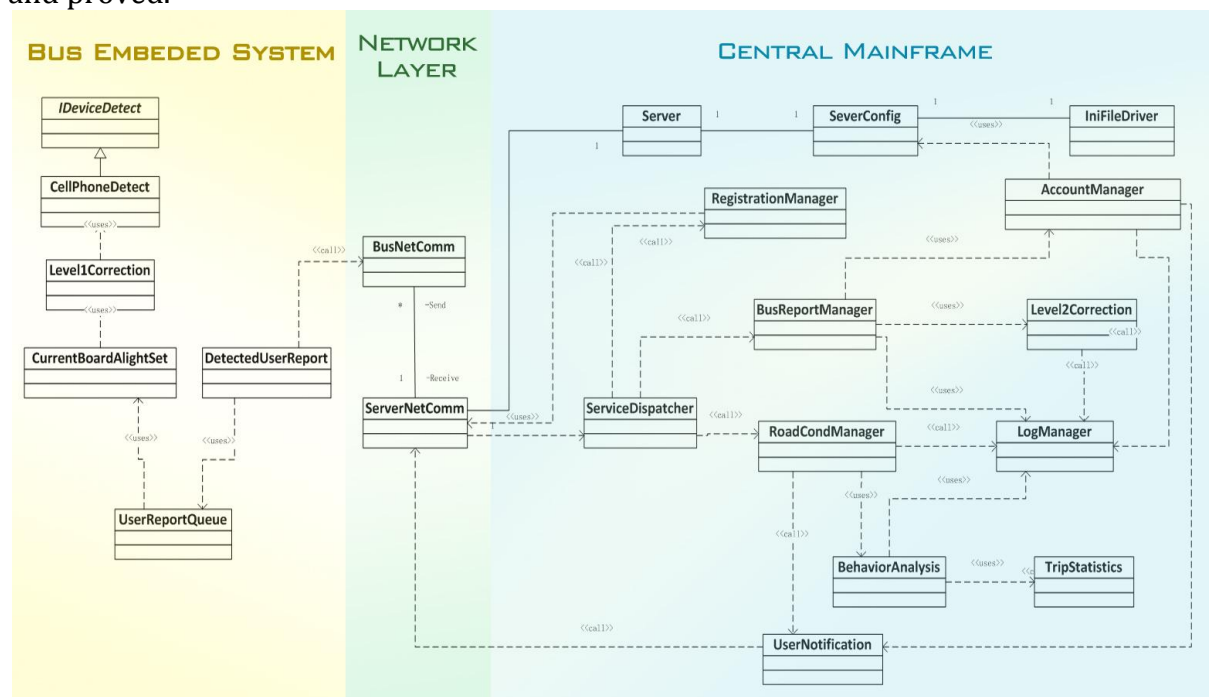


Figure 5 Class Diagram

### Dynamic Design

In the dynamic design part, we specify critical and complex system behaviors described in the architectural design by sequential diagrams. Those sequential diagrams are able to capture the system behaviors in details, thus can helps the understanding of the system dynamic features and guide our implementation. The following sequential diagram in Figure 6 is an example for user charging process.

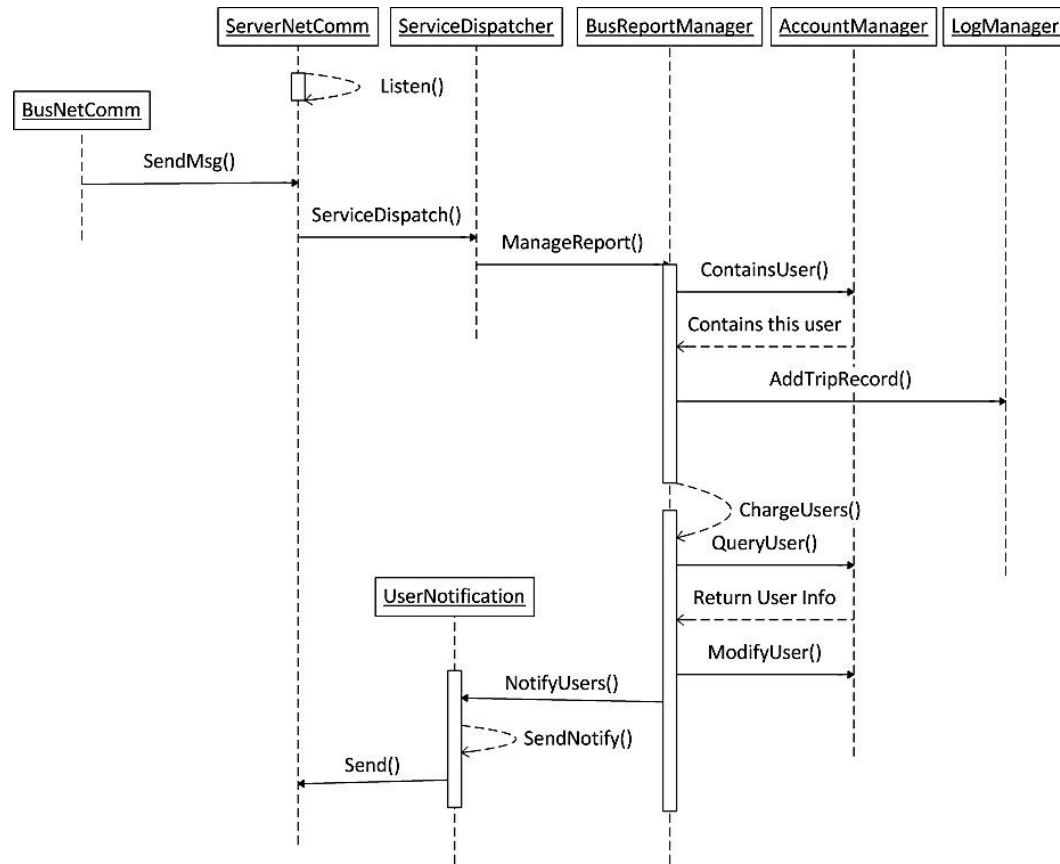


Figure 6 User Charging Sequential Diagram

## 6. Implementation

### 6.1 Strategies of Fault Detection and Correction

To accurately detect passengers on board, the detecting hardware must have a detecting scope exactly overlap with the physical volume of bus. Unfortunately, neither Wi-Fi nor Bluetooth detector can meet the requirement. So fault detection and correction have to be done on the software level.

Usually, the detecting radius of Bluetooth and Wi-Fi is beyond the physical boundaries of bus. So we first make some assumptions: the passengers who are on board can definitely be detected; on the other hand, overly detection may happen under the following two situations.

*Situation 1:* A bus may detect innocent people nearby (on the street or at a bus stop).

*Situation 2:* When two buses move in parallel, detector on one bus may detect passengers on the other bus.

Based on the analysis above, we suggest a two-level strategy to correct potential errors in detection.

#### Level 1 (Bus Level)

At the first level, we let detector on bus detect repeatedly for two or three times and merge the detecting results by set union operation. The detection process must be executed at least twice between any two bus stops. There are two purposes for doing this. First, multiple detections almost avoids the situation 1, as human walking speed is much slower than that of the buses in most cases. Second, it also reduces the probability



of situation 2, as two buses can hardly keep in parallel all the time. Following is the working flow of the repeating detection algorithm.

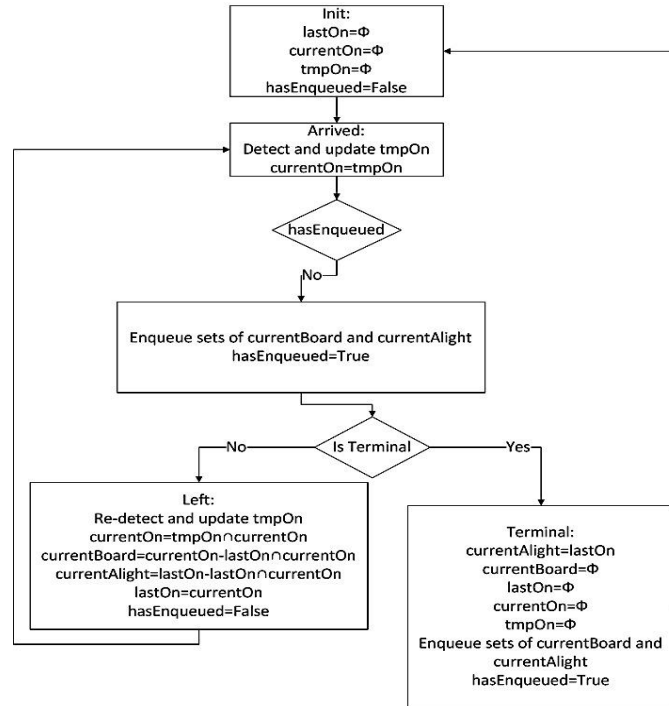


Figure 7 Repeating Detection Work Flow

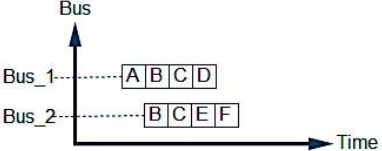
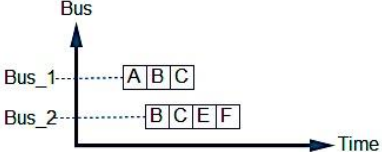
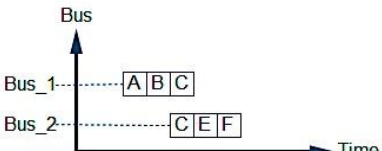
### Level 2 (Server Level)

As mentioned above, situation 2 cannot be totally avoided in Level 1 correction. Thus some ambiguous data will be sent to the server. So fault correction is still needed on the server level. However, the non-determinism of the behaviors of users and buses makes the correction algorithm very complex. In order to guide our algorithm design, we first analyze all possible scenarios that will lead to confusion. Thanks to PAT, after a systematic model checking, we get all the possible error-causing scenarios and successfully get the algorithm. Our approach is as follows:

- Step 1:* Start from a small model, only two bus lines (bus<sub>1</sub>: A-->B-->C-->D; bus<sub>2</sub>: B-->C-->E-->F) and only one passenger whose trip path is fixed (get on bus<sub>1</sub> at A, transfer to bus<sub>2</sub> at C, get off bus<sub>2</sub> at F).
- Step 2:* Implement the above model in PAT, and simulate overly detection in the model (when two buses get to the same stop, they will detect each other's passengers).
- Step 3:* Get the results generated by PAT and analyze the data.

The detail running results and analysis can be seen in the following table:

Traces (Generated by PAT)	Analysis		
	Simplify	Visualization	Predicate
get.2.B-->get.2.C-->get.1.A--> get.1.B-->get.1.C-->get.2.E--> get.2.F -->get.1.D-->Skip	1AD2CF		1AC2CF (Correct)

get.1.A-->get.1.B-->get.2.B--> get.2.C-->get.1.C-->get.2.E--> get.2.F-->get.1.D-->Skip	1AD2BF		1AC2CF 1AB2BF (Uncertain)
get.1.A-->get.1.B-->get.2.B--> get.1.C-->get.2.C-->get.2.E--> get.2.F-->get.1.D-->Skip	1AC2BF		1AC2CF 1AB2BF (Uncertain)
get.2.B-->get.1.A-->get.1.B--> get.1.C-->get.2.C-->get.2.E--> get.2.F-->get.1.D-->Skip	1AC2CF		1AC2CF (Correct)

According to the table above, there exist some situations in which server is not able to decide the correct results (e.g., scenario 2 and 3). Therefore, there still exist errors after the correction process despite of the extremely low possibility. We build a bigger model and refine our algorithm according to it to reduce the error rate to the minimum. To be noted that, we also use this approach to test the real fault correction code. The results not only prove the reliability of the code, but also the correctness of our algorithms. More details can be found in section 9.1.

## 6.2 Methodology of User Behavioral Analysis

The goal of user behavioral analysis is to determine users' preferences of different bus lines as well as their regular traveling period such that the system is able to inform the relevant registered users when a particular bus line is interrupted in a certain time period. To achieve this goal, the server needs to perform analysis on a regular basis, for example, once a day. The analysis is based on the trip information data, which is derived by matching boarding and alighting records stored on the central server. A trip information record includes user ID, trip-starting time, trip-ending time and the bus line name.

CellPhoneAddr	BoardTime	AlightTime	BusLine
19	9:05 PM	10:09 PM	Line6
7	9:29 AM	11:10 AM	Line6
17	9:35 PM	10:53 PM	Line4
11	5:12 PM	6:00 PM	Line5
10	8:16 AM	8:31 AM	Line3
8	7:42 AM	8:05 AM	Line1
1	9:42 AM	9:43 AM	Line1
9	8:12 PM	8:17 PM	Line0
11	5:38 PM	5:56 PM	Line0
3	10:02 AM	11:28 AM	Line6
5	10:10 AM	11:58 AM	Line0
10	8:10 PM	9:47 PM	Line0
3	10:00 AM	11:28 AM	Line6

Figure 8 User Behavioral Trip Information Records

When the analysis process begins, all the new trip information records are traversed once. In the meantime, statistics of each registered user are calculated and recorded on the server. Apart from the total count of taking each bus line, the time distribution of the user's traveling time also needs to be calculated. We divide the operating hours of bus lines into discrete one-hour-long periods and



construct a time axis based on that. Then for each trip record, the traveling time span is projected on to the time axis and different time spans of records are accumulated together. After all the trip records are processed, a column chart that shows the traveling time distribution can be constructed for each user. We build an integrated simulator, “User Behavioral Analyst” to generate trip information records and then perform analysis on the data. Figure 9 and Figure 10 are the screen shots of the analyzing results from the “User Behavioral Analyst”.

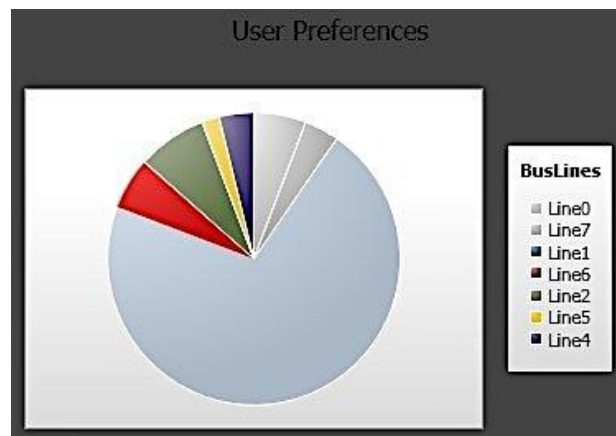


Figure 9 User Preferences Analysis Result

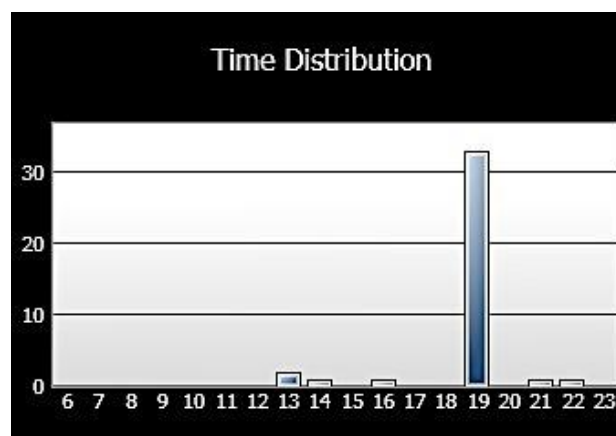


Figure 10 Time Distribution Analysis Result

Given the analysis results, whenever a particular bus line is interrupted in a certain time period (e.g., “Line1” closed from 9 a.m. to 10 a.m.), the related users who are supposed to be informed can be easily figured out. Following the previous example, the system will go through each user and determine if “Line1” is a regular bus line of him or her by looking at the percentage of taking “Line1” among all the bus lines. If the percentage is higher than the predefined threshold (e.g., 30%), then the line is considered a preferred bus line for the user. By looking at the time distribution, the system is able to determine if 9 a.m. to 10 a.m. is a regular traveling period of the user and decides whether the interruption is related to him or her.

### 6.3 Experiment on Bluetooth Device Detection

As mobile device detection is a part of the main functions of the bus embedded system and we do not plan to fully implement it due to the hardware complexity, we experiment on the Bluetooth device detection API and simulate the device detection in the simulator. We use a shared-source library, “32feet.NET” to realize the Bluetooth device detection.

After importing the library, it is fairly easy to get the information of Bluetooth devices around the detector by using the provided method “DiscoverDevices()”. This method returns an array of objects “BluetoothDeviceInfo” that contains device name and device address. Therefore, we could uniquely identify a particular registered user by accessing the device address. We could also convert the array to a data set to facilitate the fault correction and detection algorithm mentioned before.

For simplicity, we assume the discovery for Wi-Fi devices is similar to that for Bluetooth. Hence, the system architecture design is not affected if Wi-Fi detection is implemented.

### 6.4 Using PAT as Trip Planning Service

We also implement an experimental function, “trip planning”, which makes use of the model checking capability of PAT as a route planning service. This function provides a guide for users who are not familiar with the bus routes and need suggestions for choosing bus lines. This can also be applied to provide alternative optimal path to subscribers, although it is not fully implemented.

There are two advantages of using PAT as the underlying planning service. Firstly, it saves the time of implementing an AI planning algorithm for the trip-planning problem. The only job we need to do is to convert the map and bus line configurations to a CSP# model and then ask PAT to do a reachability test to find a solution that satisfies the goal. Another advantage is that the searching algorithm of PAT is highly efficient such that the performance of trip planning is ensured with no extra effort.

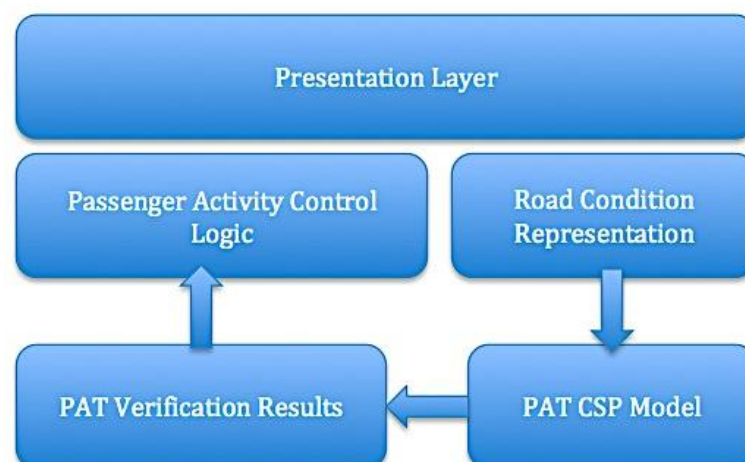


Figure 11 Simulator Architecture

The simulator generates a CSP# model during execution according to the current road conditions and bus line configurations, whenever a subscriber is querying about which route to choose. Users can choose their starting point as well as destination on the simulator interface. By clicking on the “Plan” button, the underlying support modules will generate a CSP# model according to what have been chosen and pass it to PAT. After interpreting the returned results from PAT, the system is able to display the planned route and detailed instructions to users.

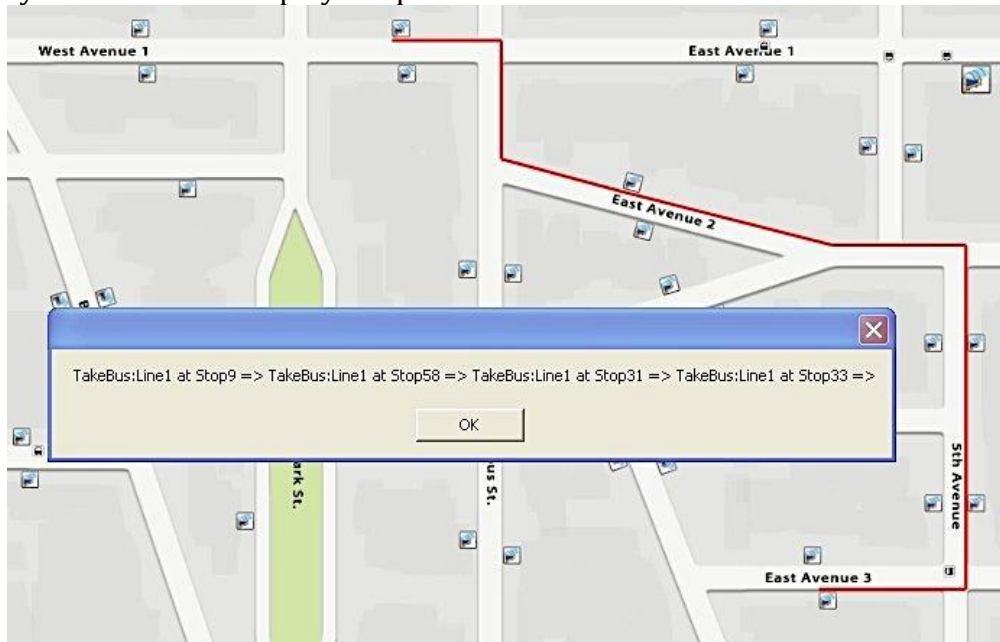


Figure 12 Route Planning Result

## 6.5 Implementation Details of Server Core

The implementation of the server core strictly follows the CSP-OZ specifications. Each object in the design model is directly converted into class in the real code. Methods in the class correspond to operations in objects and they share the same preconditions and effects. The responsibilities of each component are described below.

Component	Responsibilities
<b>Server</b>	Server main control
<b>ServerConfig</b>	Provide access to T_COST and T_VAL values
<b>IniFileDriver</b>	Provide operations on initialization file
<b>BusReportManger</b>	Manage bus reports and update fare and trip information
<b>RegistrationManger</b>	Provide service for user registration
<b>AccountMangager</b>	Provide access to the account information
<b>UserNotification</b>	Interface for user notification
<b>ServerNetComm</b>	Provide network communication services
<b>ServiceDispatcher</b>	Dispatch services to different service providers
<b>TripManger</b> (Inherits Logmanager)	Manage trip logging

Furthermore, we follow the typical Client-Server model when implementing the server core. ServerNetComm listens at some ports and waits for incoming TCP connections. ServiceDispatcher starts new threads to dispatch services, such that

the listening thread is not blocked. Main data structures and message types used are described below.

Name	Information	Members
<b>RequestInfo</b>	Service Request Message	Service Type
		Content of request
<b>BusReportMsg</b>	Bus Report Message	Report Time
		Bus Line Name
		Stop Name
		Boarding/Alighting
		Bus line name
		User ID set
<b>UserInfo</b>	User Account Information	User Cell Phone Hardware ID
		Name
		Cell phone number
		Whether holds a valid ticket
		Ticket issuance time
		Payment type
		Credit card number
		Account balance
<b>TripRecord</b>	Trip Information Record	Time
		Bus line name
		Boarding/Alighting
		Stop name

## 6.6 Implementation Details of UI and Simulators

### Interface

The two simulators we build, “Transport4You Simulation” and “User Behavior Analyst” are coded in C# and use Windows Presentation Foundation (WPF) for user interface rendering. Data binding is extensively used for dynamically updated data display because of its logical separation of the data from the presentation. For instance, the bus line and user status are bind to the corresponding objects, so that they are instantly updated when the underlying data changes. Thus, using data binding also reduces line of code significantly by avoiding manually writing codes for UI updates.

An external library for WPF development, “WPF Toolkit” is also used in our project. One component of the collections, “WPF Themes” that changes the appearance of the window without any modification to the original codes is applied to furnish our user interface. In addition, “Data Visualization Toolkit” is used to draw charts and diagrams to visualize the user behavioral analysis results.

### Animation

The bus animation is created using the path animation feature of the WPF Storyboard. A PathGeometry object, which provides the path of the bus line that moves along, controls the path animation. We encode the path coordinates list in a bus line configuration file. The bus line movement can be directly modified under the “Route” tab in the server window. New bus stops can be added into the

bus line route. Parameters including stop names and coordinates of the existing stops can also be modified in this panel. The modifications will take effect once the server window is closed or the “update” button is clicked.

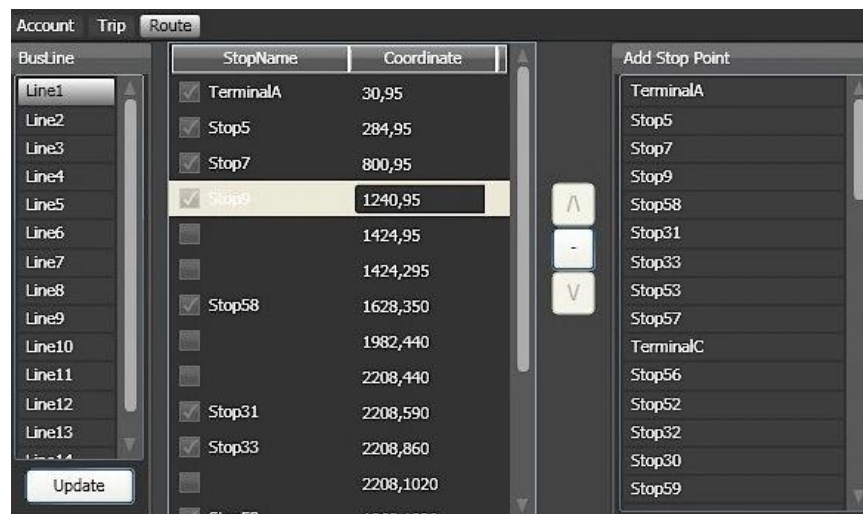


Figure 13 Route Modification

## 7. Management Plan

### 7.1 Team Introduction

Our team consists of three students all from National University of Singapore, two of us are master students and the other one is an undergraduate student. Everyone has a background of formal methods in software engineering, but only rest on theory. SCORE2011 provides us a great chance to apply our knowledge in a real project.

More details of team members and responsibilities are as follows:

Name	Initials	Expertise	Responsibility
Yang Hang	YH	OO design Network development, .Net platform	Project Manager Architect Coding
Li Yi	LY	Control and animation programming, .Net platform DB design	UI Design DB Design Coding
Wu Huanan	WHN	Formal specification Formal verification Testing	Testing Modeling Documentation

### 7.2 Communication

As we are a small group and most of the time we can sit together, we have more chances to conduct face-to-face meetings. However, on the other hand, small team makes everyone’s workload increased and brings much more pressure. Moreover, with fewer people, ideas and opinions are also limited. To solve these problems, we proposed several plans. First, every Tuesday we held a meeting to keep track of the weekly progress and discuss the next week’s plan. Second,

when brainstorming, we encourage different opinions on a single problem. Then we discuss and pick the best solutions from the pool of ideas. Third, we keep regular communication with our stakeholders to ensure that our system is on the right track.

### 7.3 Risk Management

For this project, the shortage of manpower is the main source of risks. For example, some potential risks are falling behind the schedule, technique barriers, and lacking of quality control.

Risk	Level	Prevention
Fall behind schedule	High	Make a proper plan according to individual capability. Mutual supervision between team members.
Technique barrier	High	Use familiar techniques. In case of meeting some unexpected unknown tech-problem, consult experts immediately.
Lack of quality control	Medium	Review and refine weekly work in the meeting. Get feedback from project reviewers.
Requirements change	Low	Keep active communication with the stakeholders before finish the first round requirement specification.

### 7.4 Artifacts Management

We produce corresponding documentations at each stage of the development. Once a file was created, every member can access and modify until it is fixed. However, when a file needs to be modified, the original version shall be preserved. Version control and standard naming format are used to manage the documents.

Although we are a small group, and only two of us are involved in coding, version control and coding style are still strictly enforced. Both documentations and code are checked and backup daily.

## 8. Project Plan

### 8.1 Development Plan

The project we choose involves hardware. For example, a Bluetooth and Wi-Fi device detector should be installed on a bus to detect the passengers getting on or off from a bus. For simplicity, we decide to not fully implement those components. We use a simulator to emulate those components instead. Other components of the system, such as mainframe server and user portal are put in real implementation.

The following table lists the details of our development plan.

	Design	Specification	Simulation	Actual Implementation
User behaviors		√	√	
User registration	√	√	√	

Bus behaviors		✓	✓	
Bus embedded system	✓	✓		✓
Network layer	✓			✓
Central mainframe	✓	✓		✓

## 8.2 Activity Plan

Following the iterative and incremental development process, we divide the project into two iterations, marked by our two project deadlines. Some results of the first iteration are reviewed and modified in the second iteration. According to this, we divide the project into several parts and assign reasonable time period to each part. The initial schedule can be represented as the following Gantt diagram.

ID	Activity	October					November					December					January	
		W38	W39	W40	W41	W42	W43	W44	W45	W46	W47	W48	W49	W50	W51	W52	W1	W2
1	Project Planning	■																
2	Requirements Analysis	■	■	■	■	■	■											
3	Architecture Design		■	■	■													
4	Detailed Design (UML)			■	■	■												
5	Formal Specification & Verification (CSP-0Z & CSP#)			■	■	■												
6	Simulator Implementation							■	■	■								
7	Server Core Implementation								■	■	■							
8	Testing							■	■	■	■							
9	Deadline for Course Project										★							
10	Requirements Review											■	■	■	■			
11	Design Review												■	■	■			
12	Modify Implementation													■	■	■		
13	Testing (Based on Formal Method)														■	■	■	
14	Final Report																■	■
15	Deadline for Score2011																	★

## 9. Verification and Validation

### 9.1 Testing by Model Checking

What we want to highlight here is that we use model checking to help testing complex components such as “fault correction”. To test “fault correction”, we design a CSP# model that describes the behaviors of buses and users. For example, different buses may share some common bus stops, and users may board or alight from the buses at any stop. This model allows us to systematically generate test cases according to the actual interactions between users and the system with no pain. PAT supports user defined C# library, i.e., an object or method defined in the C# library can be invoked as a part of the model. This creates a way of linking events of the interface models with code expected to test (Sun, Liu, & Cheng, 2010).



We write an interface class in C# and combine it with the test model to facilitate the connection with the actual code. Thus, a transition in the test model will trigger the execution of the real code. We then can compare the execution results to the expected ones. After execution, 51350 test cases are generated. Meanwhile, all the test results are acquired. The following is one detailed test log.

*[Flag:2AD],[Records:2AD3CE],[Final:2AD],[Trace:get.2.A->get.3.B->leave.3.B->u.on.2.A->leave.2.A->get.2.B->leave.2.A->get.2.C->get.3.C->leave.2.C->leave.3.C->get.2.D->get.3.E->leave.3.E->get.3.F->u.off.2.D->Skip]*

In the above record, the contents after “Flag” represent the actual trip information of the user. Here, “2AD” means the user takes bus “Line2” from “StopA” to “StopD”. “Records” is followed by the trip information reported to the server. In this case, “2AD3CE” means that the user first takes bus “Line2” from “StopA” to “StopD”, and then transfers to bus “Line3”, boarding at “StopC” and alighting at “StopE”. Obviously, such bus report will lead to an undecidable situation, because the user can be on either one of the buses between “StopC” and “StopD”. In addition, “Final” is followed by the corrected results after the fault correction process. The contents after “Trace” indicates the detailed execution sequence of the test model. Following the given trace, the interpretation is as follows. “get.2.A”, which is an event name in CSP#, means that bus “Line2” arrives at stop A. “leave.3.B” means that bus “Line3” leaves from “StopB”. “u.on.2.A” and “u.off.2.D” means that the user boards on “Line2” at “StopA” and alights from “Line3” at “StopB”. Therefore, because “Final” and “Flag” share the same contents, fault correction successfully corrects the bus report.

Some typical examples in our verification results are shown below.

Records	Meanings
[Flag:2AD],[Records:2AD],[Final:2AD]	No need to correct
[Flag:2AC3CF],[Records:2AC3CF],[Final:2AC3CF]	Report is the same as reality and getting right route and right charge
[Flag: 2AD],[Records: 2AD3CE],[Final:2AD]	Report is not the same as reality and after correction, getting right route and right charge
[Flag:2AC3CF],[Records:2AC3BF],[Final:2AB3BF]	Report is not the same as reality and after correction, getting wrong route but right charge
[Flag: 2AC3CE],[Records:2AD3CE],[Final:2AD]	Report is not the same as reality and after correction, getting wrong route and less charge

Statistics of the verification results are shown below:

Correct fare cases	Count	Percentage
<b>Correct route and correct fare</b>	48728	94.9%
<b>Incorrect route and correct fare</b>	1428	2.8%
Incorrect fare cases	Count	Percentage



<b>Incorrect route and less fare</b>	1194	2.3%
<b>Incorrect route and more fare</b>	0	0%
Summary	Count	Percentage
<b>Total Correct fare cases</b>	50156	97.7%
<b>All cases</b>	51350	100%

As shown in the table above, with fault correction, the bus fare can be charged correctly in the most cases. More importantly, the system never over charges the users, which perfectly meets the requirement.

## 9.2 Traditional Testing

We also applied traditional unit tests and global black box tests. The unit tests are carried out periodically along with the implementation while the integration tests are done in the end to ensure that the software is bug-free and meets the requirement specifications.

### Main Functions Test Results:

Action	Expected Result	Test Result
User Detection	Cell phones and be detected by PC equipped with Bluetooth and return a set	OK
User registration	User is recognized as a registered user by system	OK
User login and modify his setting	User succeeds in logging into system and can modify his setting	OK
E-ticket payment	When user holds an valid ticket, he/she will be charged only once within the duration that the e-ticket is valid	OK
Account balance payment (With no fault report)	User can be charged by T_COST	OK
Bus Detection Report	Bus can detect and report users correctly	OK
Simulator UI and animation basic test	Simulator can display coherently with system behaviors and UI can preference normally	Failed/Fixed

## 10. Lessons Learned

This project was done in a relatively short time and the workload is quite intense for a team of only three people. The first challenge to us is how to effectively manage time and manpower in an effective way. We realized this problem at the very beginning of the project and made detailed management plan and project plan beforehand. Those plans now appear very helpful in guiding our development pace and workload distribution.

This project is also a very good chance for us to apply what we have learned in formal methods classes to a real system. We have seen the powerfulness of formal verification when we first found new bugs by model checking. Also thanks to formal specification, the system is designed more rigorously with clear requirements before starting the implementation.

We would like to treat this project as not only a software engineering contest, but also a precious learning opportunity. Many new techniques and concepts that we encountered along the way now become part of our intellectual wealth. In fact, those are the things that we value the most.

A short demonstration can be found at:

<http://www.youtube.com/watch?v=EQDxHJTVuzQ>

## Works Cited

contributors, W. (2010, December 21). *Conductor (transportation)*. Retrieved January 2, 2011, from Wikipedia:  
[http://en.wikipedia.org/w/index.php?title=Conductor\\_\(transportation\)&oldid=403472179](http://en.wikipedia.org/w/index.php?title=Conductor_(transportation)&oldid=403472179)

Fischer, C. (1997). CSP-OZ: a combination of object-Z and CSP. *Proceedings of the IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems* (pp. 423-438). Canterbury: Chapman & Hall, Ltd.

Sun, J., Liu, Y., & Cheng, B. (2010). Model Checking a Model Checker: A Code Contract Combined Approach. *The 12th International Conference on Formal Engineering Methods (ICFEM 2010)*. Shang Hai.

Sun, J., Liu, Y., Dong, J. S., & Pang, J. (2009). PAT: Towards Flexible Verification under Fairness. *The 21th International Conference on Computer Aided Verification (CAV 2009)* (pp. 709-714). Grenoble: Springer.

Sun, J., Liu, Y., Roychoudhury, A., Liu, S., & Dong, J. S. (2009). Fair Model Checking with Process Counter Abstraction. *The sixth International Symposium on Formal Methods (FM 2009)* (pp. 123-139). Eindhoven: Springer.