

Software Design Document

For


Transport4You

Version 1.0 approved

Prepared by Li Yi, Yang Hang and Wu Huanan

National University of Singapore

Created on 10/11/2010



Revision History

Date	Description	Author	Comments
10/11/2010	Version 0.5	Yang Hang	First Draft
10/01/2011	Version 0.9	Wu Huanan	Second Draft
24/02/2011	Version 1.0	Li Yi	Final Draft

Document Approval

This Software Design Document has been accepted and approved by the following:

Signature	Name	Title	Date
	Yang Hang	Project Manager	28/02/2011

Table of Contents

Revision History	2
Document Approval.....	2
1. Introduction	5
1.1 Purpose	5
1.2 Intended Audiences	5
1.3 Document Organization	5
2 Design Overview.....	7
2.1 Entity Relationships	7
2.2 Design Using Formal Methods.....	7
2.2.1 CSP-OZ	8
2.2.2 CSP#	9
2.2.3 Process Analysis Toolkit	9
2.3 Design Workflow.....	10
3. System Architectural Design.....	11
3.1 System Overview	11
3.2 Bus Embedded System	11
3.3 Central Mainframe	12
4. Detailed Functional Design.....	13
4.1 Component Hierarchy	13
4.2 Formal Specification of Components.....	14
4.2.1 Component: Bus Embedded System.....	14
4.2.2 Component: Bus Report Manager	17
4.2.3 Component: Account Manager.....	18
4.2.4 Component: Registration Manager	19
4.2.5 Component: Road Condition Manager	20
4.2.6 Component: User Notification	21
4.3 Dynamic Design	23
4.3.1 Scenario: Detection and Report	23
4.3.2 Scenario: Fare Deduction and User Notification	23
4.3.3 Scenario: User Registration	25
4.3.4 Scenario: Road Condition Update.....	25
5. Design Verification	26
5.1 Translating to CSP#	26
5.2 Verification Results	28
6. Interface Design	30
6.1 Transport4You Simulator Main Window	30
6.2 Transport4You Simulator Server Window	30
6.3 Transport4You Simulator User Window	32
6.4 User Behavior Analyst Window.....	32
7. Implementation Consideration	35

7.1 Strategies of Fault Detection and Correction.....	35
7.2 Methodology of User Behavioral Analysis	37
7.3 Experiments on Bluetooth Device Detection	38
7.4 Using PAT as Trip Planning Service.....	38
Appendix: CSP# model.....	41
References	45

1. Introduction

This section provides an overview of the entire software design document.

1.1 Purpose

This Software Design Document (SDD) describes all architectural, interface and component-level design for the Transport4You Intelligent Public Transportation Manager (IPTM) system. This document follows strictly from the IPTM Software Requirements Specification (SRS) which clearly specifies the major requirements and constraints of the system. It is a living document that is expected to evolve throughout the design process. During conceptual design it provides a 'broad-brush' perspective of the design with details to be added during subsequent design phases. The IPTM system SDD also serves as a detailed guideline in the implementation stage by providing the details for how the software should be built. Within the SDD are narrative and graphical documentation of the software design for the IPTM system, including class diagrams, sequence diagrams and formal design models. Developers should refer to the designs and specifications in this document for their entire development process.

1.2 Intended Audiences

The intended audiences of stakeholders for this design description of the IPTM include:

- Transport4You IPTM system development team:
 - Project Manager, who must review and approve the SDD.
 - Designers, who constantly review and revise the SDD.
 - Developers, whose implementations have to be based on the designs specified in this SDD.
- Transport4You IPTM system project stakeholders (clients):
 - Managers.
 - Customer Representatives, who must approve it.

1.3 Document Organization

This document is organized into the following sections:

- Introduction, which introduces the design document for the IPTM system to its readers.
- Design Overview, which gives a brief description of the IPTM system design, including its structure, workflow and techniques used.
- System Architectural Design, which describes the overall structure of the IPTM system and the functionalities of the sub systems.
- Detailed Functional Design, which includes component hierarchy, formal specifications and dynamic design.

- Design Verification, which describes the methods and results of the design formal verification process.
- Interface Design, which specifies the user interface design of the IPTM system.
- Implementation Consideration, which presents special design considerations on implementation including strategies of fault detection and correction, methodology of user behavioral analysis, experiments on Bluetooth device detection and using PAT as trip planning service.
- Appendix, which presents the CSP# design model.

techniques and model checking tools are extensively used in the IPTM system design.

2.2.1 CSP-OZ

Object-Z is an extension of the formal specification language Z, which facilitates specification in an object-oriented style and improves the clarity of large specifications through enhanced structuring. CSP (Communicating Sequential Processes) (Hoare, 1978) is an event-based formal method for describing patterns of interaction in concurrent systems. The combination of these two languages, called CSP-OZ (Fischer, CSP-OZ: a combination of object-Z and CSP, 1997), takes the best of these approaches. It is a wide range specification language for complex distributed communicating systems like satellites, railroad or telecommunication systems.

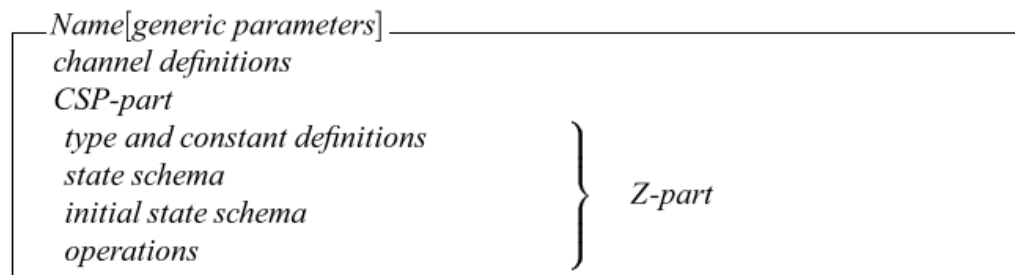
2.2.1.1 CSP

As an event based formalism, CSP is able to describe behaviors and interactions of complex processes using very simple syntax. The syntax and fundamental language primitives can be summarized as follows:

P :=	STOP	(Process that do nothing)
	SKIP	(Process that terminates successfully)
	e->P	(Prefixing)
	P □ P	(External choice)
	P ∩ P	(Internal choice)
	P P	(Interleaving)
	P [X] P	(Synchronous parallel)
	P \ X	(Hiding)
	P; P	(Sequential composition)
	P ∇ P	(Interrupt)

2.2.1.2 Syntax of CSP-OZ

In general, a CSP-OZ specification is a collection of interacting objects, they communicate with each other via channels in the style of CSP. The basic structure of a CSP-OZ class is as following:



More detailed semantics and syntaxes of CSP-OZ can be found in (Fischer, CSP-OZ: a combination of object-Z and CSP, 1997).

2.2.1.3 Informal semantics of CSP-OZ

As a combination of Object-Z and the process algebra CSP, the general idea of CSP-OZ is to argument the state-oriented Object-Z specification with the specification of behavior in the style of CSP. (Fischer & Wehrheim, Model-Checking CSP-OZ Specifications with FDR, 1999) The OZ part is used to handle the data refinement and the definitions of constants, invariants and functions, while the CSP part is to specify the ordering among events. Enable schemas of operations can be used to define data-dependent behavior.

2.2.2 CSP#

CSP# is the model description language of PAT which extends CSP by embedding data operations. It combines high-level compositional operators from process algebra with program-like codes. Following are some of the features of CSP#:

- Data Variables
 - CSP# is weak typing language and therefore no typing information is required when declaring a variable.
- Data Operations
 - Like other programming languages, CSP# can use assignments, if-else, while and other statements to do data operations.
- Event Prefixing
 - CSP# allow user to add condition expressions before events and assignment expressions after events.
- Process Constructs
 - CSP#, extending CSP, can express various operations between processes in order to model the system.

2.2.3 Process Analysis Toolkit

A sophisticated and self-contained model checker, Process Analysis Toolkit (PAT) (Sun, Liu, Dong, & Pang, 2009), is used at the design stage to formally verify the correctness and reliability of the design model.

PAT is designed to verify event-based compositional models specified using CSP as well as shared variables and asynchronous message passing. It supports automated refinement checking, model checking of LTL extended with events and various ways of system simulation.

One of the unique features of PAT is that it allows user to define static functions and user defined data type in C# language in order to use them in the models. These C# classes are built as DLL files and loaded when models import them. Once they are defined, they can be used afterwards directly in any models.

2.3 Design Workflow

A design model that captures the main structural and functional properties of the system is constructed for the purpose of design verification. The model evolves along with the whole design process. 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 specification, which is able to precisely describe the patterns of interactions among different software components in an object-oriented fashion. However, currently there is no tool support for CSP-OZ. Therefore, the CSP-OZ model is translated into CSP#, which is the modeling language of PAT and has very similar idea on process abstraction with CSP-OZ. The CSP# model is then verified in the 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.

3. System Architectural Design

3.1 System Overview

The Transport4You IPTM system consists of two sub systems, namely the bus embedded system (BES) and the central mainframe (CM). 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.

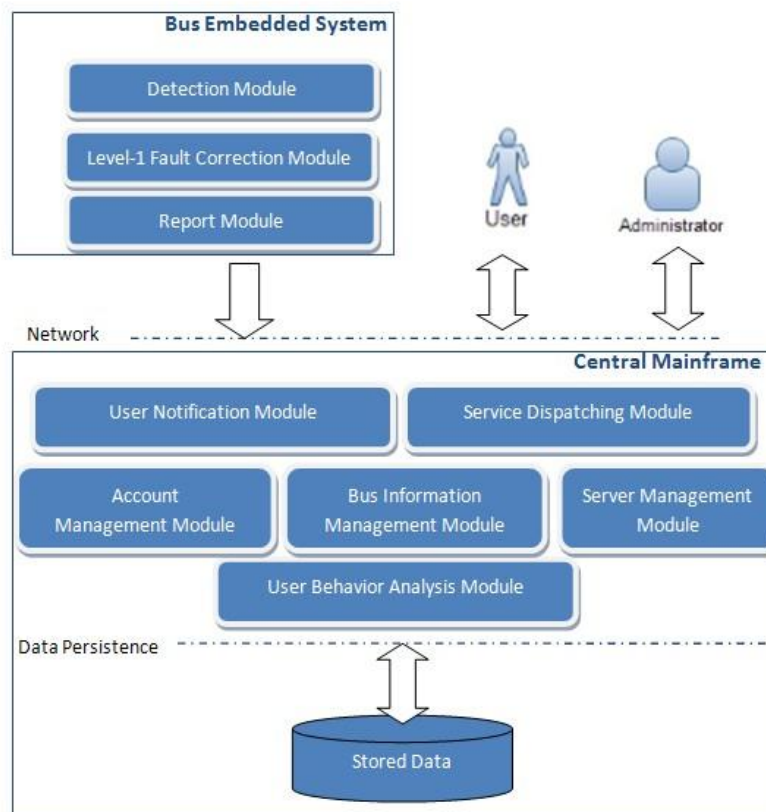


Figure 2 System Architecture Diagram

3.2 Bus Embedded System

BES consists of three major modules:

- Detection Module, which is responsible for passenger detection. It directly communicates with the detection hardware which is installed on every bus. Detection Module provides real-time data of the current on-board passenger set to other dependent modules.

- Level-1 Fault Correction Module, which filters out part of undesirable erroneous on-board data and passes more accurate on-board information to the Report Module.
- Report Module, which processes the on-board information and sends them through network connection to the central mainframe.

3.3 Central Mainframe

CM has the following sub modules:

- Service Dispatching Module, which constantly dispatches incoming service request messages to related service providing modules. Provided services include Bus Reporting Service, Account Management Service, Road Condition Update Service, etc.
- User Notification Module, which sends out SMS messages to subscribers via SMS sending network.
- Account Management Module, which processes all the account related requests, including registration request, balance update request, personal information update request, etc.
- Bus Information Management Module, which preprocesses bus report messages and carries out the Level-2 Fault Correction to get more accurate trip information. After getting the corrected trip information, then the module proceeds to subsequent actions.
- Server Management Module, which provides an interface for administrators to maintain and manage the data on the central mainframe.
- User Behavior Analysis Module, which retrieves and analyses user trip records to produce user preferences data.
- Data Storage module, which is in charge of data storing and logging. Part of the persistent store (user trip information) is implemented using database permitting searchable access to the contents.

4. Detailed Functional Design

This section presents the detailed functional design of the IPTM system. First, the hierarchical structure of the system components is given to capture the relationships and interactions among components. Then the CSP-OZ specifications which model the desired system behaviors and required properties are provided for each component. In the end, sequential charts and workflow diagrams that describe the typical scenarios of the system are presented to illustrate the dynamic aspects of the system.

4.1 Component Hierarchy

In this subsection, the relationships among all sub-systems described in the architectural design are represented 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 directly transferred into classes in the real code.

As shown in Figure 3, the system is divided into three parts as is specified in the architectural design. The “*LogManager*” class that is used by many other classes corresponds to the Data Storage Module 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 is experimented on and proved.

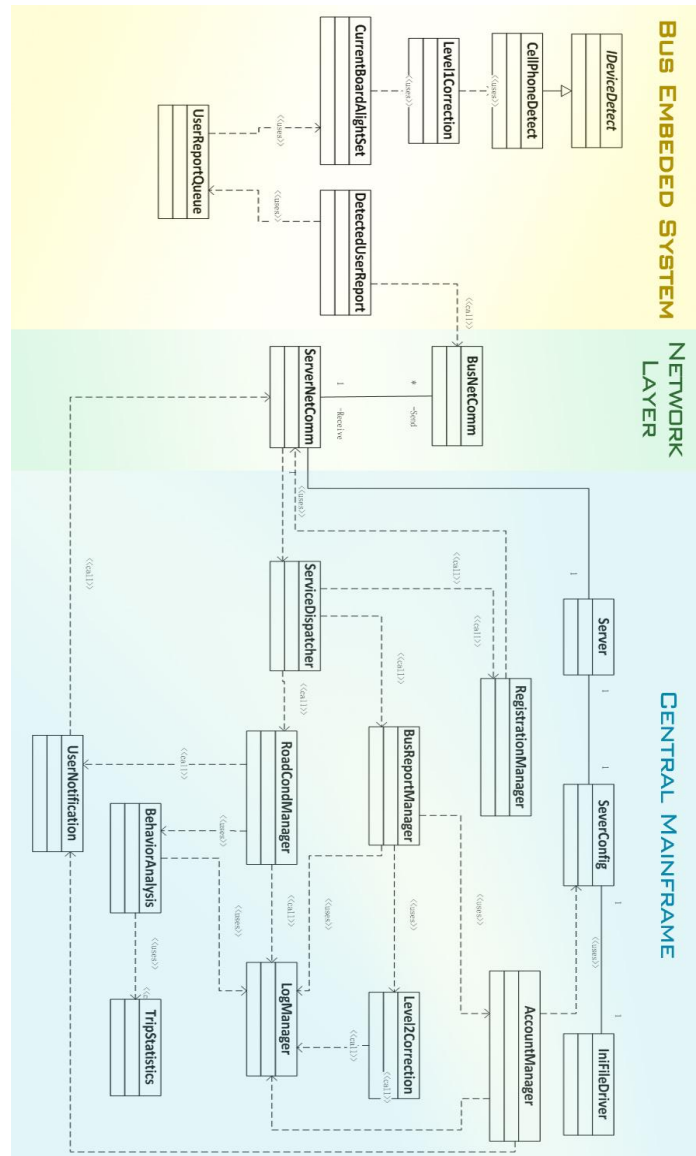


Figure 3 Class Diagram

4.2 Formal Specification of Components

This subsection formally specifies the detailed designs of all the significant components of the IPTM system using CSP-OZ. The contents are grouped by the different components they belong to.

4.2.1 Component: Bus Embedded System

The specification for bus embedded system is from two strongly correlated aspects: bus operation and bus behavior. In this case, bus moving along an established route and stopping at certain location can be treated as its behavior, while detecting on-board passengers and uploading the information to the central server are its operations. Although bus behavior is not part of the system, it is still useful for design verification. Some details about the CSP-OZ specification in Figure 4 are as follows:

1. The first three channels represent the interfaces to the environment, while the last two channels represent the network connections to the central mainframe.
2. The CSP part of the specification shows clearly that a bus only detects after it leaves the bus stop and immediately upload the information.
3. Every bus has its unique ID and has an established route.
4. The property *remain* indicates a queue of bus stops which the bus has yet arrived. Since every bus follows a fixed route, so the property *remain* is always a subset of its *route*. The property *location* records the previous bus stop the bus has just arrived.
5. The enable schema of *arriveStop* demonstrates that at any time a bus can only arrive at the bus stop which is the head of *remain*. And based on the cardinality of *remain* we can judge whether it **(a)** arrives at a bus stop which is not a terminal or **(b)** arrives at the terminal.

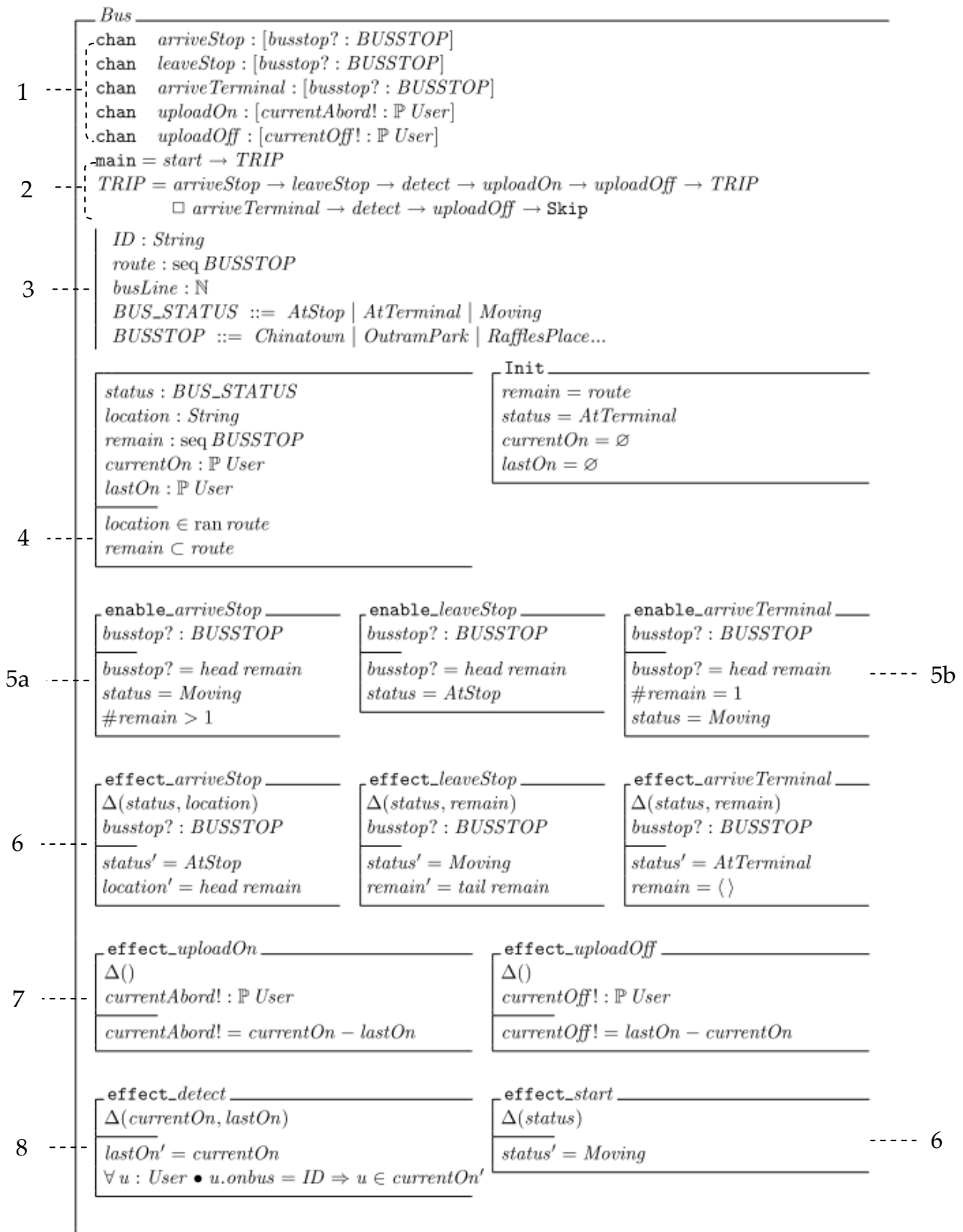


Figure 4 Bus Class

6. The effect schemas of bus behavior, which update the status or location of bus.
7. The effect schema of upload operation.
8. The *detect* operation generates the current on-board user set. The detailed detection and fault correction strategies used here can be found in Section 7.1.

4.2.2 Component: Bus Report Manager

After the bus embedded system has the *currentAbord* ready, it uploads the set to the class *BusReportManager* through the channel *uploadOn*. Figure 5 below is the CSP-OZ specification of *BusReportManager*.



Figure 5 *BusReportManager* Class

1. The process *CHARGE* in CSP part continuously charge the user in *currentAbord* till every user in the set has been charged once.
2. The function *setTOque* transfers all the elements in a set to a queue to ensure that no user is charged more than once.

3. The enable schema of *uploadOn* and effect schema of *doneCharging* ensure that *BusReportManager* will never receive another set until it finishes processing users in the current set in hand.
4. Charged user will be popped out from the queue which guarantees that no user is charged twice or escapes from being charged.

4.2.3 Component: Account Manager

When a user is to be charged, a request will be sent to *AccountManager* which inherits the class *Account*. 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 here. Following are the CSP-OZ specifications for the two classes:

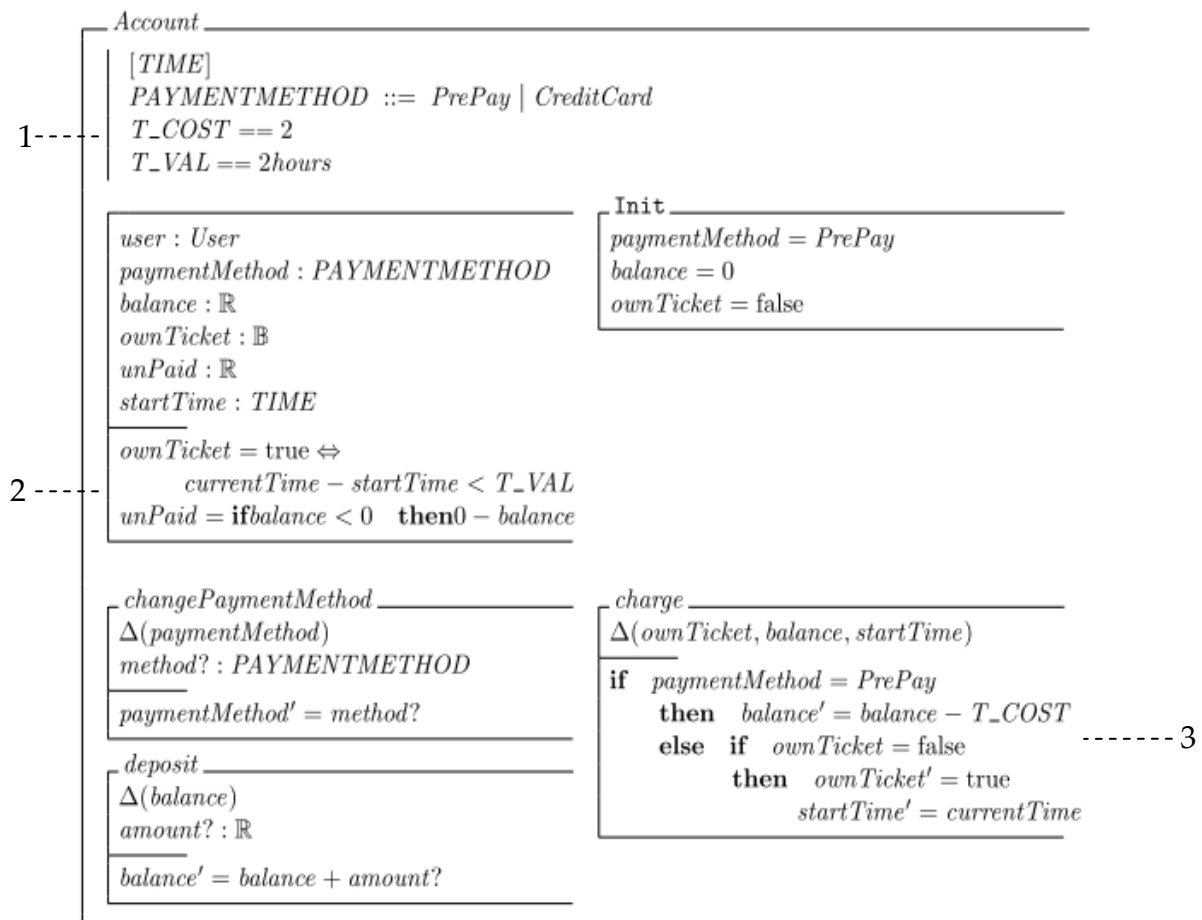


Figure 6 Account Class

1. User can choose whether to pay by prepay account or credit card. *T_COST* indicates the amount of fare for each trip. *T_VAL* indicates the validity period for every e-ticket.
2. The property *ownTicket* is Boolean type. A ticket will be expired after *T_VAL* period since it is issued. The property *unPaid* is the amount of negative balance.

3. When a charging request is received, the system will first check the payment method of the user, if he or she pays by prepay account, T_COST money will be deducted from the account, and if paying by credit card, the system will check whether there exists a valid ticket, if not, T_COST money will be deducted from the credit card and a ticket will be generated automatically.

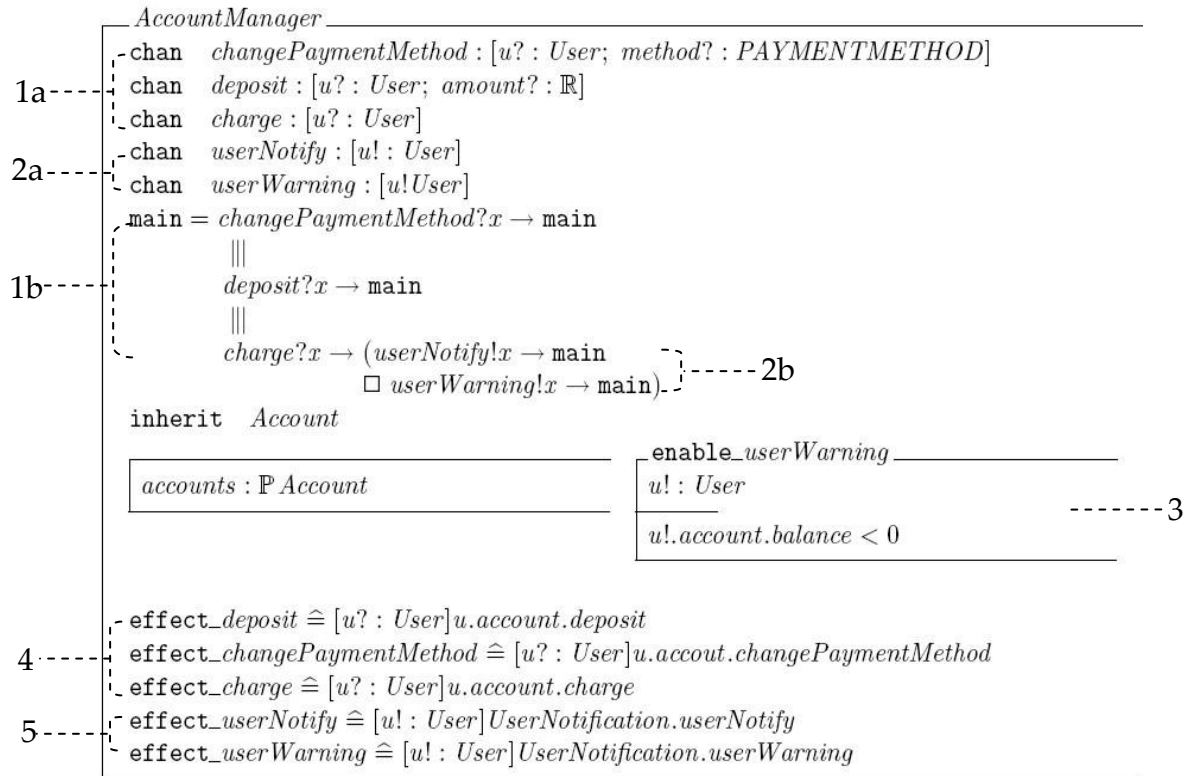


Figure 7 AccountManager Class

1. The class *AccountManager* is mainly responsible for **(b)** dealing requests related to user's account through **(a)** corresponding interfaces.
2. *AccountManager* is also in charge of sending **(b)** user notification and **(a)** user warnings.
3. When a user's account balance is inadequate, a *userWarning* will be sent to him or her.
4. As *AccountManager* inherits the class *Account*, the account operations are the same as those in *Account*.
5. These two operations are in the class *UserNotification*.

4.2.4 Component: Registration Manager

Figure 8 shows the CSP-OZ specification of *RegistrationManager* class.

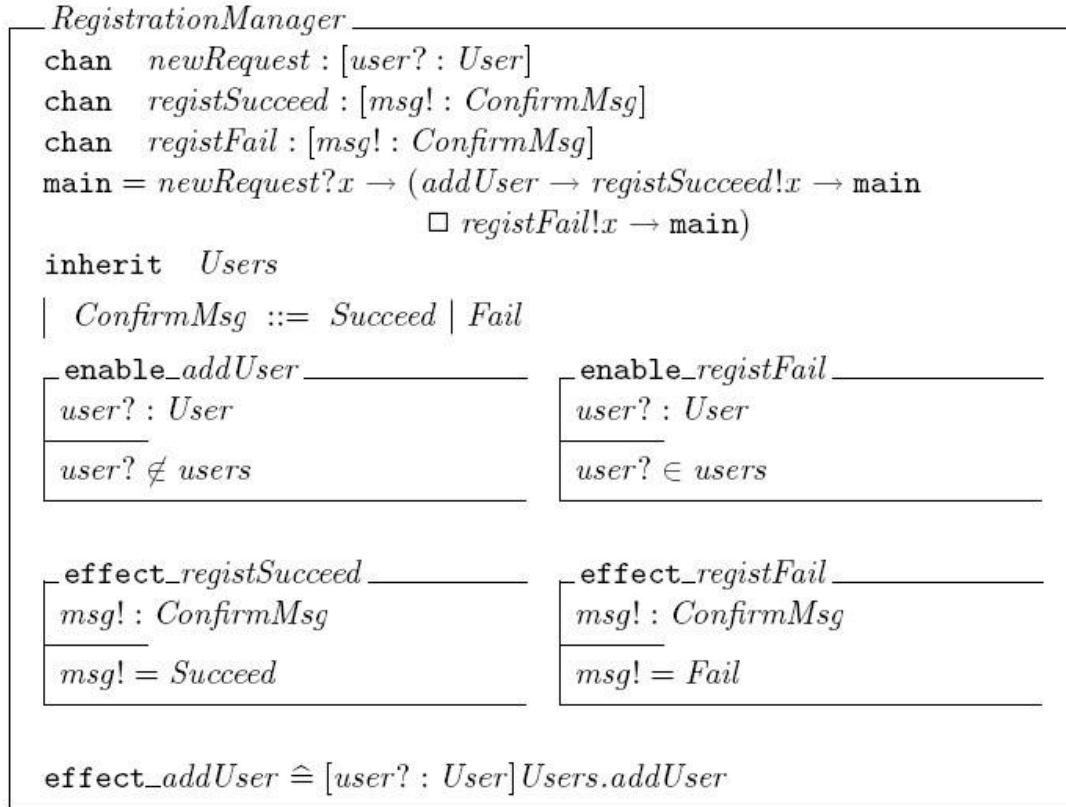


Figure 8 RegistrationManager Class

Figure 9 shows the CSP-OZ specification of the user base which is used in the *RegistrationManager* class.

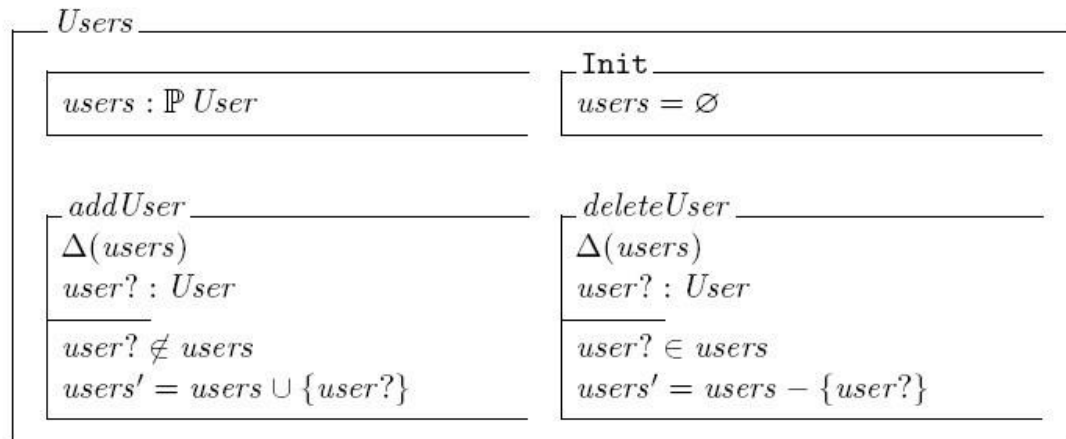


Figure 9 Users Class

4.2.5 Component: Road Condition Manager

Figure 10 shows the CSP-OZ specification of the *RoadConditionManager* class.

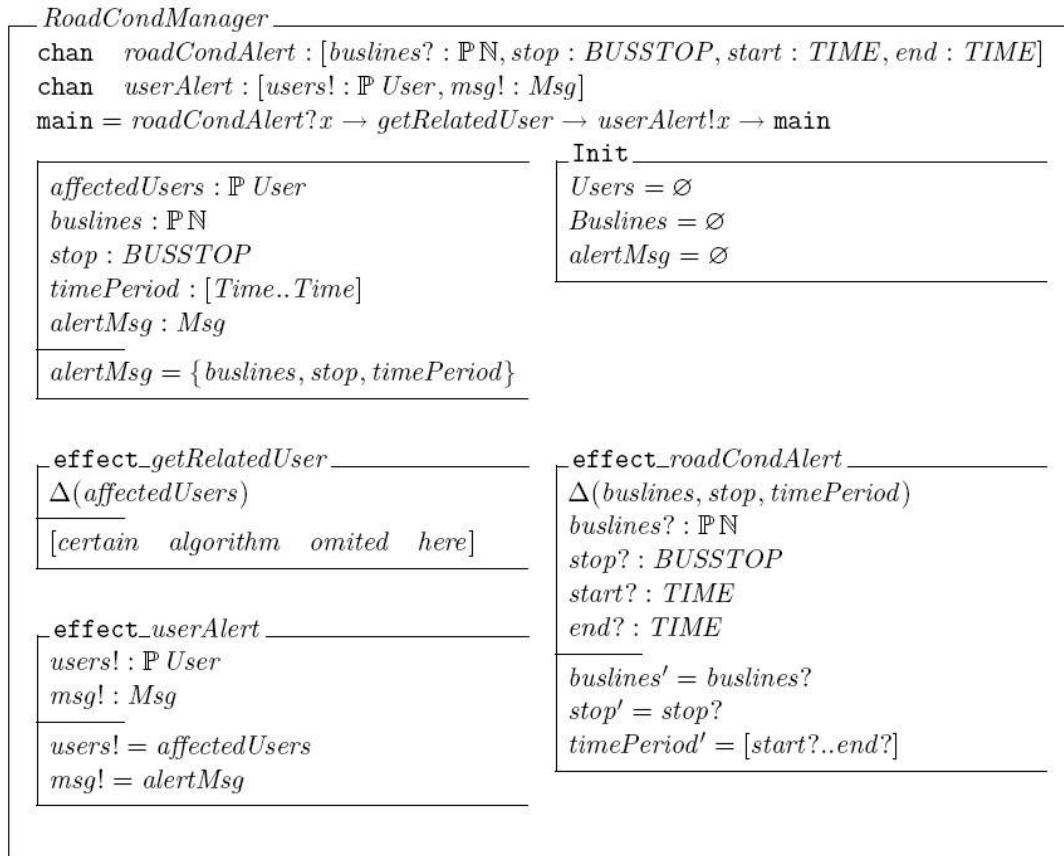


Figure 10 *RoadCondManager* Class

4.2.6 Component: User Notification

Figure 11 shows the CSP-OZ specification of the *UserNotification* class.



Figure 11 UserNotification Class

4.3 Dynamic Design

This subsection specifies the system design from a dynamic angle by providing sequence charts and workflow diagrams. The contents of this part are organized by scenarios.

4.3.1 Scenario: Detection and Report

The following diagram captures the calling sequence of a detection and report process in bus embedded system:

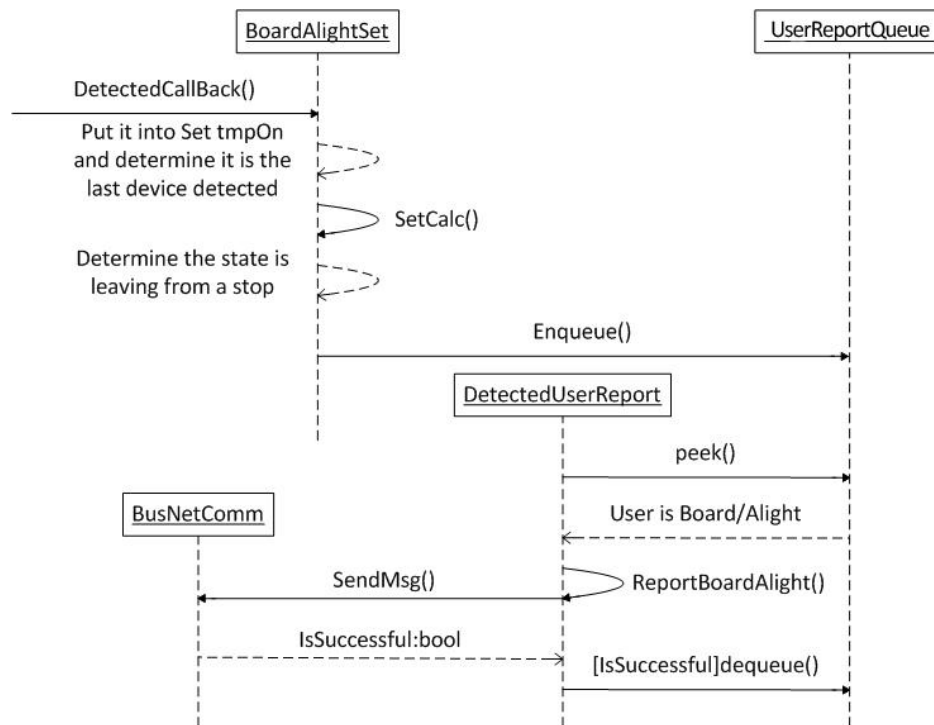


Figure 12 Detection and Report Process

4.3.2 Scenario: Fare Deduction and User Notification

The following diagram captures the calling sequence of a fare deduction and user notification process which happens after an onboard user is detected and reported:

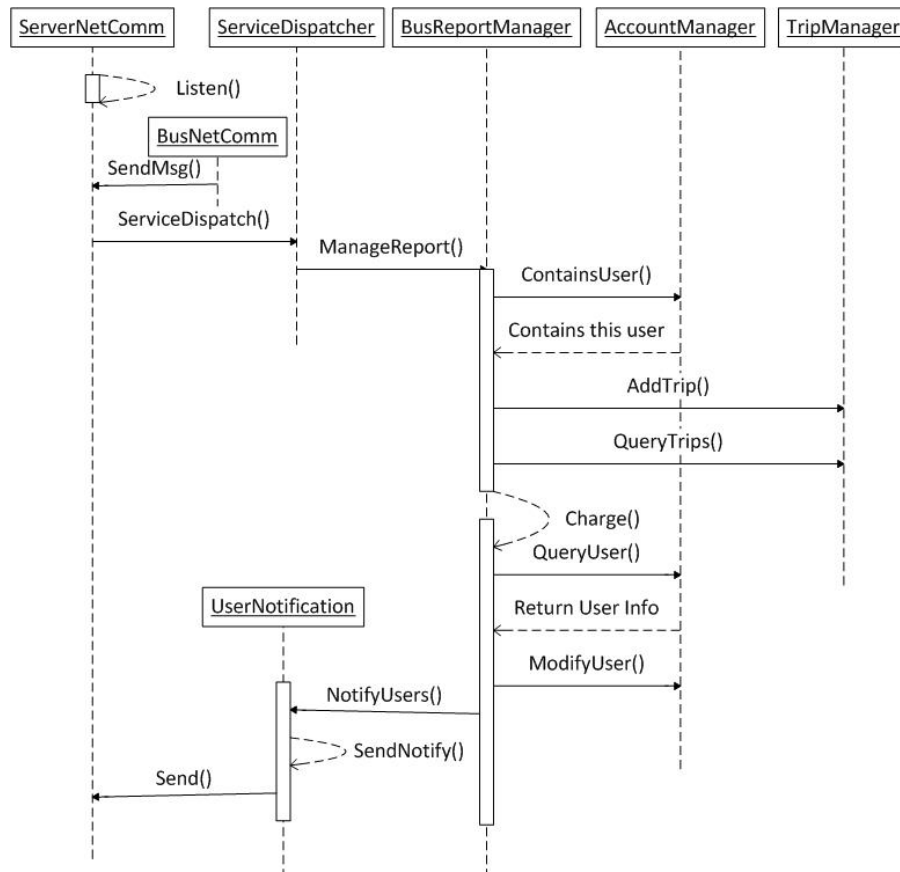


Figure 13 Fare Deduction and User Notification

The detailed fare deduction workflow is as follows:

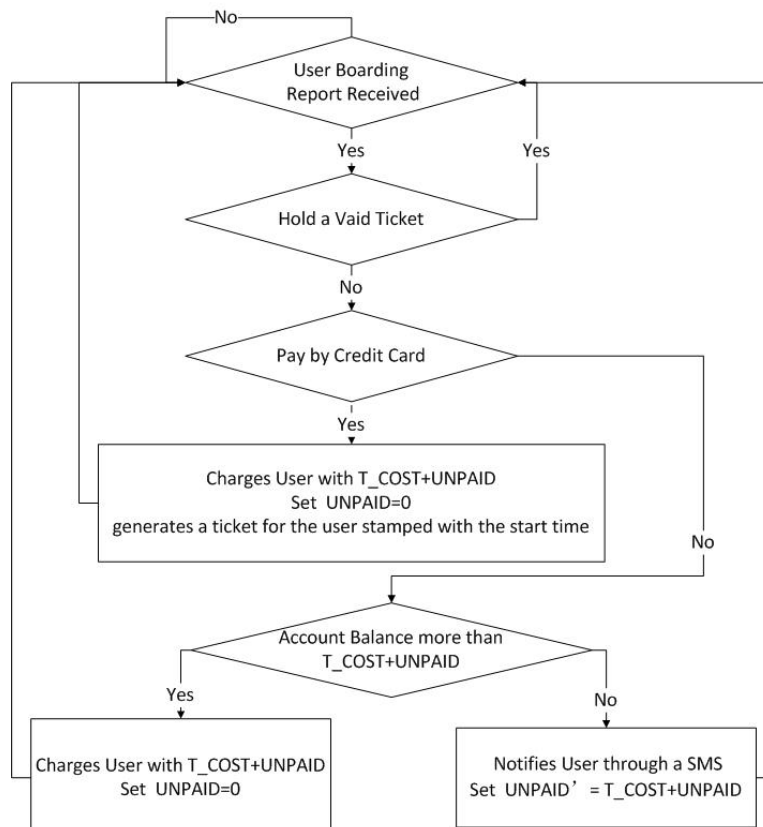


Figure 14 Fare Deduction Flow

4.3.3 Scenario: User Registration

The following diagram captures the calling sequence of a user registration process:

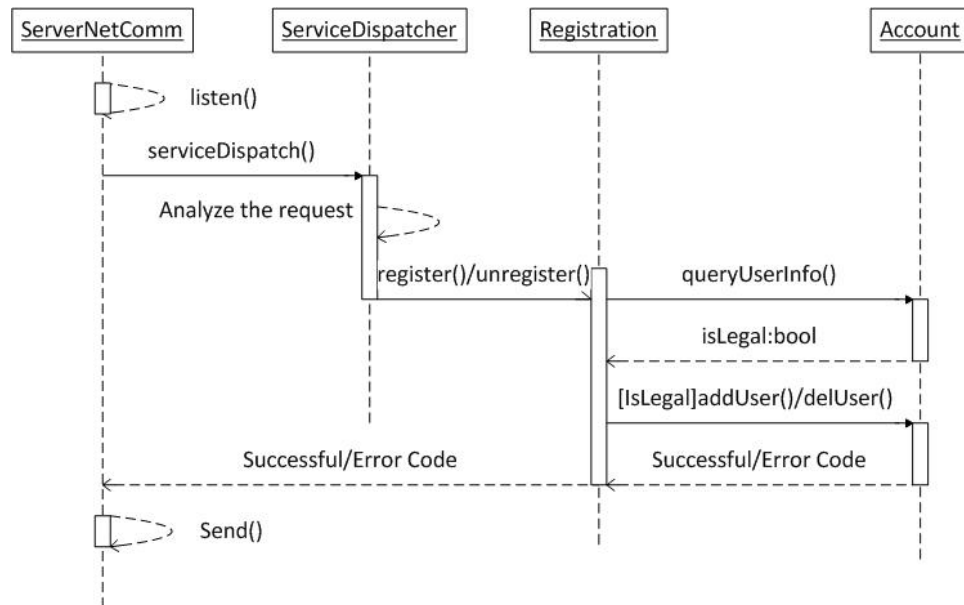


Figure 15 User Registration Process

4.3.4 Scenario: Road Condition Update

The following diagram captures the calling sequence of a road condition update process:

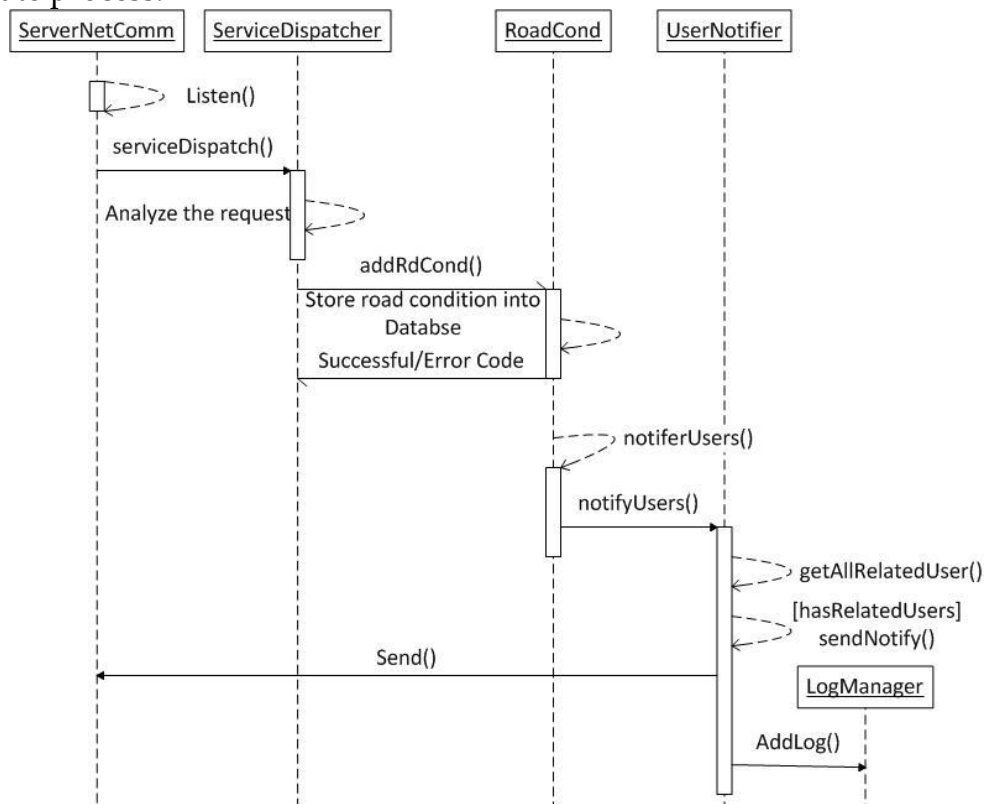


Figure 16 Road Condition Update Process

5. Design Verification

This section provides information on the design verification approach adopted and the verification results in the IPTM system design.

5.1 Translating to CSP#

CSP#, the model description language of PAT extending CSP by embedding data operations, combines high-level compositional operators from process algebra with program-like codes. It makes CSP# like "CSP + simple Z". For example, the syntax of CSP# allows user to add guards before events and assignment expressions after events. It is very similar to Z notation that has data aspects limitation for each operation, and it is also similar to CSP-OZ syntax which has enable schema (optional) and effect schema for every event or operation. Thus, an enable schema in CSP-OZ can be directly translated to a guard condition in CSP#. Based on these points, the translation from CSP-OZ to CSP# is feasible. *UserBehavior* that is written for verification purpose depicts the essential behaviors of a user. The translation of the *UserBehavior* object is shown below.

CSP-OZ Specification	CSP#
<p><i>UserBehavior</i></p> <pre> chan getOn : [bus? : Bus] chan getOff : [bus? : Bus] main = getOn?x → getOff?x → main USER_STATUS ::= AtStop OnBus </pre> <div> <div> <p><i>Init</i></p> <pre> location : String status : USER_STATUS onbus : String status! = OnBus ⇔ onbus = Nil </pre> </div> <div> <p><i>enable_getOn</i></p> <pre> bus? : Bus status = AtStop bus?.status = AtStop location = bus?.location </pre> </div> <div> <p><i>enable_getOff</i></p> <pre> bus? : Bus status = OnBus onbus = bus?.ID bus?.status = AtStop </pre> </div> </div> <div> <div> <p><i>effect_getOn</i></p> <pre> Δ(status, onbus) bus? : Bus status' = OnBus onbus' = bus?.ID </pre> </div> <div> <p><i>effect_getOff</i></p> <pre> Δ(status, location, onbus) bus? : Bus status' = AtStop location' = bus?.location onbus' = Nil </pre> </div> </div>	<p>[These two channels are both connected to the environment (not with other process), we can only use event with compound form to capture the meaning]</p> <p>[definition part: CSP# use a loose type, normally, we can only use integer and bool, alternatively we can enum to get the String type]</p> <pre> enum{AtStop,OnBus}; var location; var status=AtStop; var onbus; </pre> <p>[operation/event part: the translation is quite intuitive, CSP# allows to put condition expressions before event and statement block after event, which just meet the needs of enable and effect schema]</p> <pre> [status==AtStop && bus_status==AtStop && location==bus_location] getOn.bus {status=OnBus;onbus=bus} [status==OnBus && onbus==bus && bus_status==AtStop] getOff.bus {status=AtStop;location=bus_location;} </pre>

Based on the previous example, there are some common rules for the translation from CSP-OZ specification to CSP# language:

Channel Definitions

Mainly, there are two kinds of channels: Environment Channel and Interface Channel:

- Environment Channel is a channel that gets information from the environment or pushes information to the environment.
- Interface Channel is a channel for communicating with other processes and classes. There exists real sender and receiver in this type of channel. Therefore, we have to define this kind of channel as a data channel in CSP#.

CSP Definitions

CSP# is based on CSP, so for this part, the translation is very intuitive.

Type and Constant Definitions

CSP# only supports integer and Boolean types in its own language constructs, which definitely cannot meet the needs. However, user defined type is supported in PAT using external C# classes which can be used to create all kinds of data structures.

State Schema

The state schema in CSP-OZ lists all data members of the class. These can also be declared as variables in CSP#.

Enable and Effect Schema

CSP# allows guard expressions before events and statement blocks after events, which exactly meets the needs of constructing enable and effect schemas.

Based on these rules, we may translate the *UserBehavior* class as follows:

```
var user_status[N_USER] = [AtStop(N_USER)];
var user_location[N_USER] = [Chinatown(N_USER)];

User_Geton_OP(user,bus,busstop) =
[user_status[user]==AtStop && user_location[user]==busstop && bus_location[bus]==busstop && bus_status[bus]==AtStop]
getonbus.user.bus.busstop {user_status[user]=bus} -> Skip;

User_Geton(user) = []bus:{0..2}@([]busstop:{Chinatown..MarinaBay}@User_Geton_OP(user,bus,busstop));

User_Getoff_OP(user,bus,busstop) =
[user_status[user]==bus && bus_location[bus]==busstop && (bus_status[bus]==AtStop || bus_status[bus]==AtTerminal)]
getoffbus.user.bus.busstop {user_status[user]=AtStop;user_location[user]=busstop} -> Skip;

User_Getoff(user) = []bus:{0..2}@([]busstop:{Chinatown..MarinaBay}@User_Getoff_OP(user,bus,busstop));

User_Trip(user) = User_Geton(user);User_Getoff(user);User_Trip(user)[]Skip;

User_Behavior() = ||user:{0..N_USER-1}@User_Trip(user);
```

Figure 17 *UserBehavior* Class CSP# Code

Besides the basic rules discussed above, CSP# also supports many advanced language constructs. For example, “if – else” statement can be used in CSP# to control the flow of a process. We may take the effect schema of the charge operation in *Account* class as an example:

```

charge
Δ(ownTicket, balance, startTime)
if paymentMethod = PrePay
then balance' = balance - T_COST
else if ownTicket = false
then ownTicket' = true
      startTime' = currentTime

```

The corresponding CSP# codes are:

```

ifa(paymentMethod==PrePay)
{charge.PrePay{balance=balance-T_COST}->Skip}
else ifa (ownTicket==false)
{charge.CreditCard{ownTicket=true}->Skip}

```

In addition, queues and sets used in the CSP-OZ specification can be easily translated to the CSP# model by defining our own data structures in PAT.

5.2 Verification Results

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 the whole CSP# model is put 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 the 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.

The workflow of user charging process is then modified and verified again. The new design model ensures the “Response” and “Reliability” properties, and thus ensures the design of the IPTM system is correct. The CSP# model is attached in the Appendix section for reference.

6. Interface Design

This section presents the user interface design of the IPTM system.

6.1 Transport4You Simulator Main Window

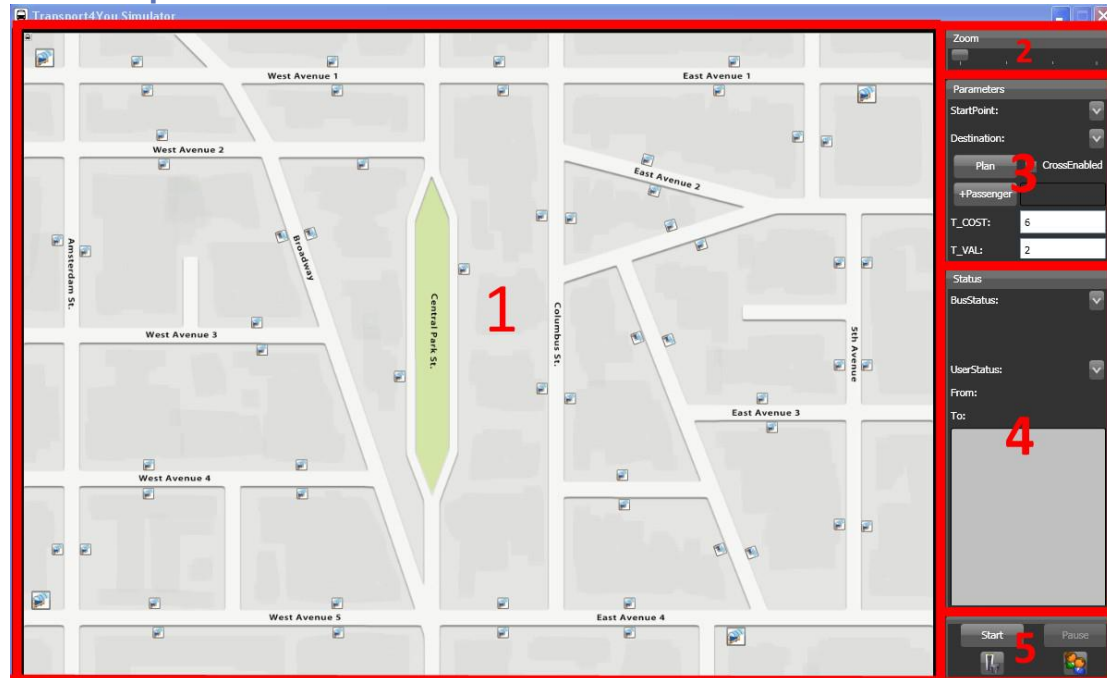


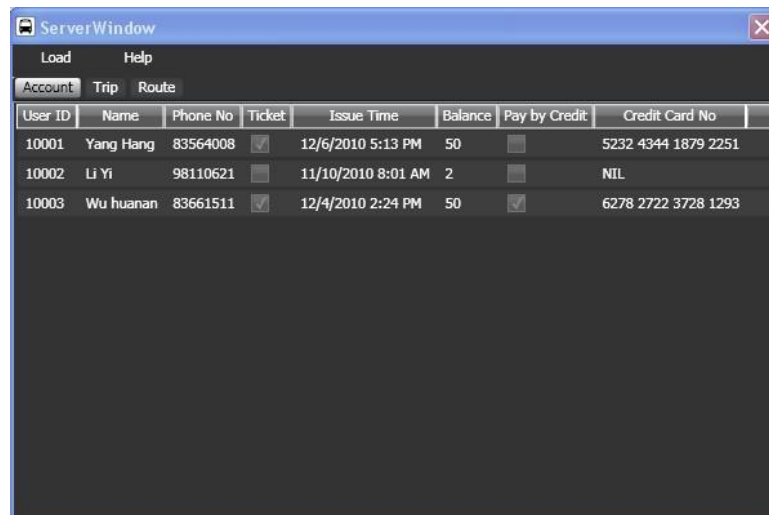
Figure 18 Simulator Main Window

Figure 18 shows the user interface of the Transport4You Simulator.

1. Map Viewer, which can zoom and scroll. Animation is enabled to show how passengers and buses travel from place to place. More information will be available when bus stop buttons are clicked.
2. Zooming Controller, which controls the zooming of the Map Viewer.
3. Parameters Panel, which is for the input of simulation parameters, including "T_COST", "T_VAL", passengers, etc.
4. Status Panel, in which the real-time status of bus lines and users are shown.
5. Control Panel, which provides animation controls and entry point for server window and user window.

6.2 Transport4You Simulator Server Window

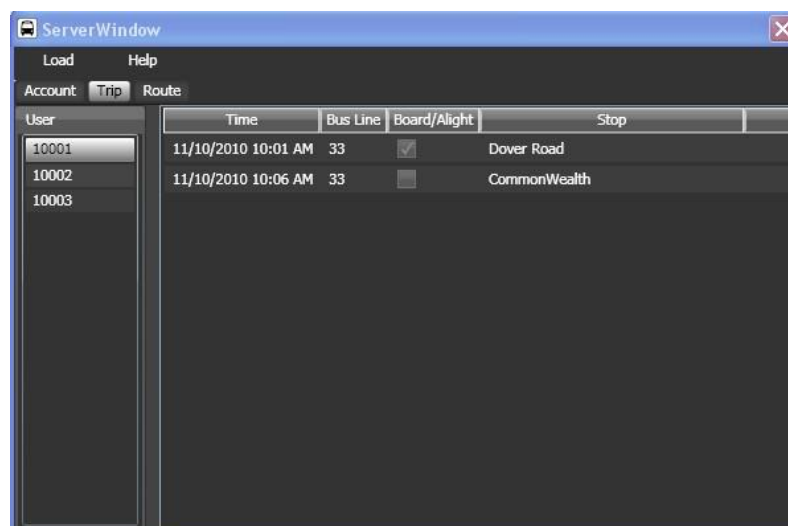
Server window has three tabs for different purposes, namely "Account", "Trip" and "Route".



User ID	Name	Phone No	Ticket	Issue Time	Balance	Pay by Credit	Credit Card No
10001	Yang Hang	83564008	<input checked="" type="checkbox"/>	12/6/2010 5:13 PM	50	<input type="checkbox"/>	5232 4344 1879 2251
10002	Li Yi	98110621	<input type="checkbox"/>	11/10/2010 8:01 AM	2	<input type="checkbox"/>	NIL
10003	Wu huanan	83661511	<input checked="" type="checkbox"/>	12/4/2010 2:24 PM	50	<input checked="" type="checkbox"/>	6278 2722 3728 1293

Figure 19 Server Window – Account Tab

Figure 19 shows the “Account” tab, in which administrators can view the account status of every subscriber. The account information shown here includes “User ID”, “Name”, “Phone No.”, “Ticket”, “Issue Time”, “Balance”, “Pay by Credit”, and “Credit Card No.”. “Ticket” indicates whether the subscriber owns a ticket and “Issue Time” indicates the time when the newest ticket is issued. “Pay by Credit” shows the payment method of the user.



User	Time	Bus Line	Board/Alight	Stop
10001	11/10/2010 10:01 AM	33	<input checked="" type="checkbox"/>	Dover Road
10002	11/10/2010 10:06 AM	33	<input type="checkbox"/>	CommonWealth

Figure 20 Server Window – Trip Tab

Figure 20 shows the “Trip” tab, in which administrators can view the trip records of all the registered users. The information shown includes “Time”, “Bus Line”, “Board/Alight” and “Stop”. “Board/Alight” indicates whether the record is boarding or alighting.

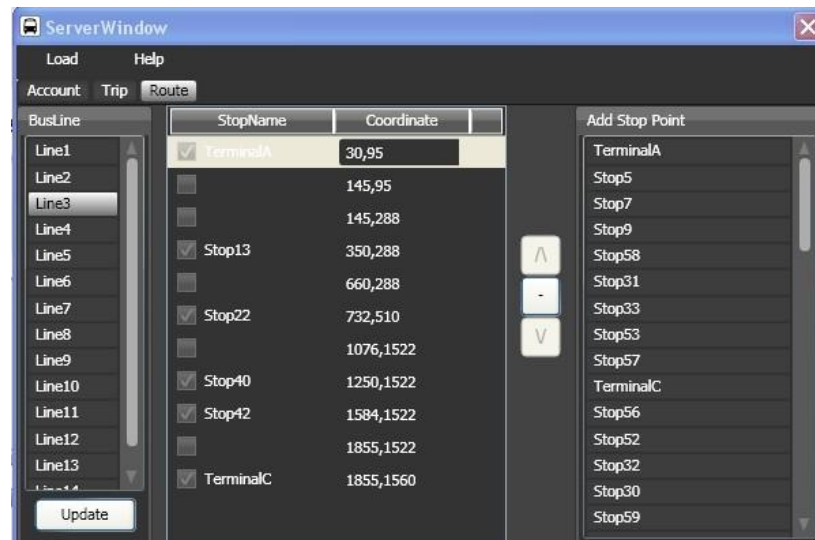


Figure 21 Server Window – Route Tab

Figure 21 shows the “Route” tab, in which administrator can modify the route for each bus line. Stops can be added, deleted, and modified. Administrator can even change the location of a particular stop.

6.3 Transport4You Simulator User Window

User window is a simulated user portal for new user registration and registered user login.



Figure 22 User Window

Figure 22 shows the design for the User Window. Unregistered user can create new account by click on the “Register” button while registered user can login by click on the “Log In” button.

6.4 User Behavior Analyst Window

User Behavior Analyst windows contains three tabs, namely “Config”, “Trip Record” and “AnalyseResult”.

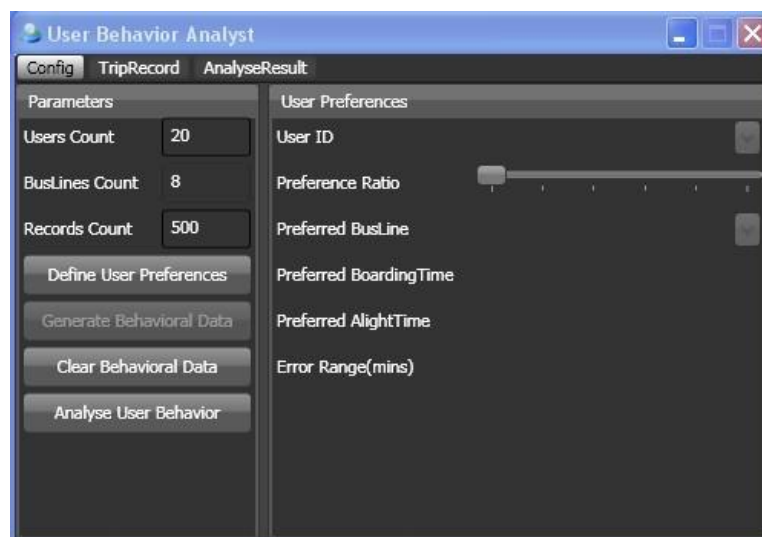


Figure 23 User Behavior Analyst – Config Tab

Figure 23 shows the interface design for “Config”. All data generation parameters can be set under this tab.

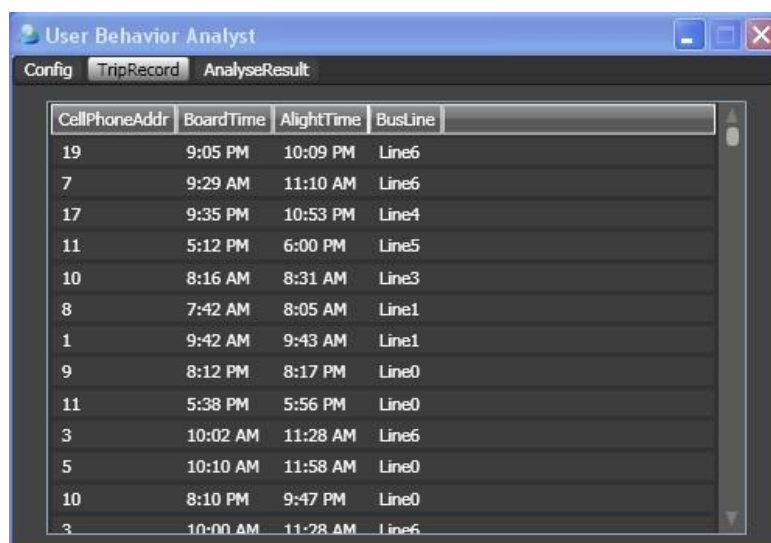


Figure 24 User Behavior Analyst – TripRecord Tab

Figure 24 shows the interface design for “TripRecord”. The generated trip record data is shown under this tab.

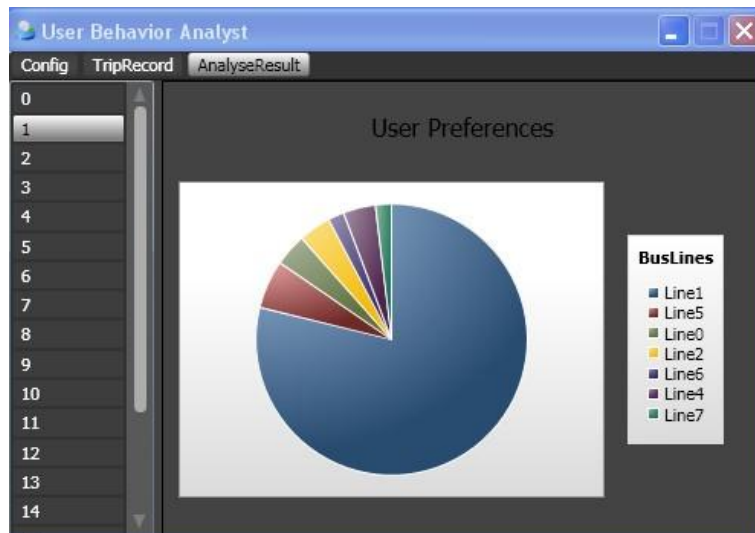


Figure 25 User Behavior Analyst – AnalyseResult Tab

Figure 25 shows the interface design for “AnalyseResult”. The pie chart illustrates the user preference analysis results.

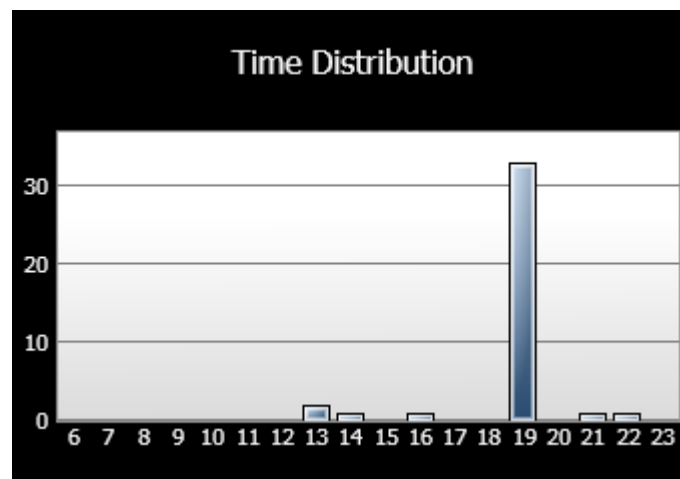


Figure 26 Time Distribution Diagram

Figure 26 shows the time distribution analysis results.

7. Implementation Consideration

This section documents the following implementation considerations:

- Strategies of Fault Detection and Correction
- Methodology of User Behavioral Analysis
- Experiments on Bluetooth Device Detection
- Using PAT as Trip Planning Service

7.1 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.

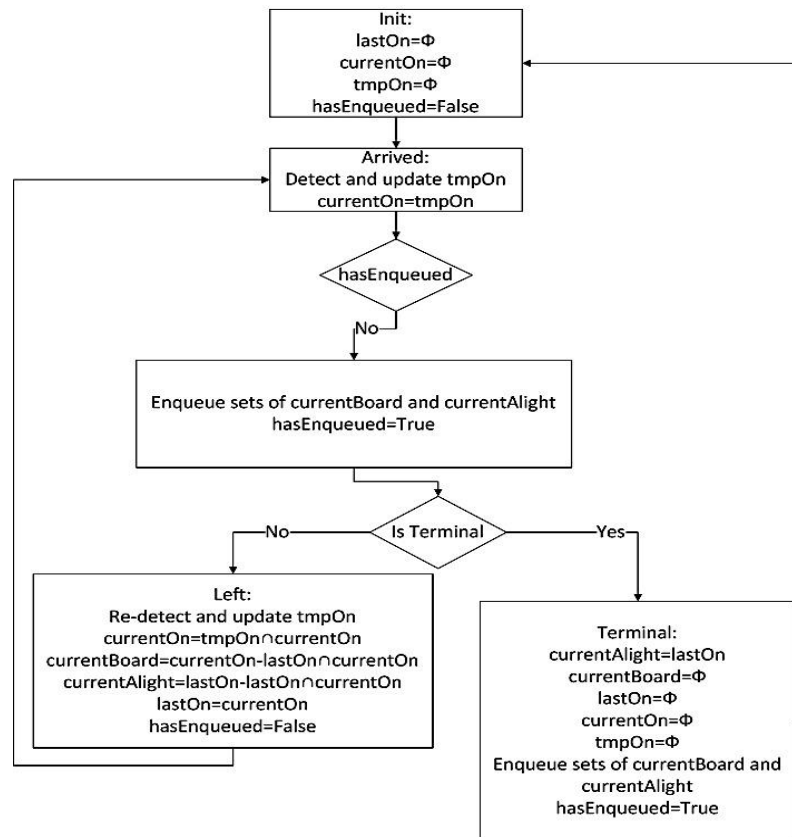


Figure 27 Repeating Detection Work Flow

Level 2 (Server Level)

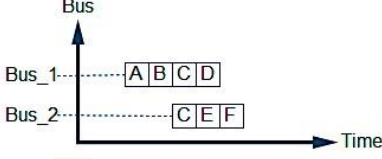
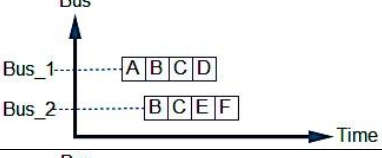
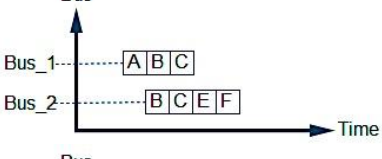
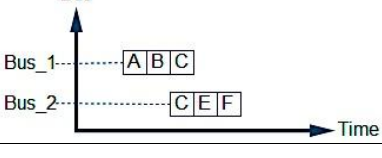
As mentioned above, situation 2 cannot be totally avoided in Level 1 correction. Thus some ambiguous data will be sent to the server. So fault correction is still needed on the server level. However, the non-determinism of the behaviors of users and buses makes the correction algorithm very complex. 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 (bus₁: A-->B-->C-->D; bus₂: B-->C-->E-->F) and only one passenger whose trip path is fixed (get on bus₁ at A, transfer to bus₂ at C, get off bus₂ 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 the test plan document: IPTM-TP.

7.2 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.

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.

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.

7.3 Experiments 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.

7.4 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.

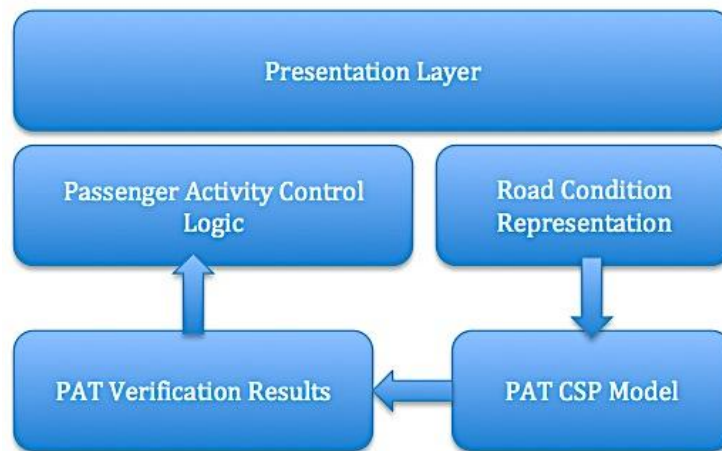


Figure 28 Simulator Architecture

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.

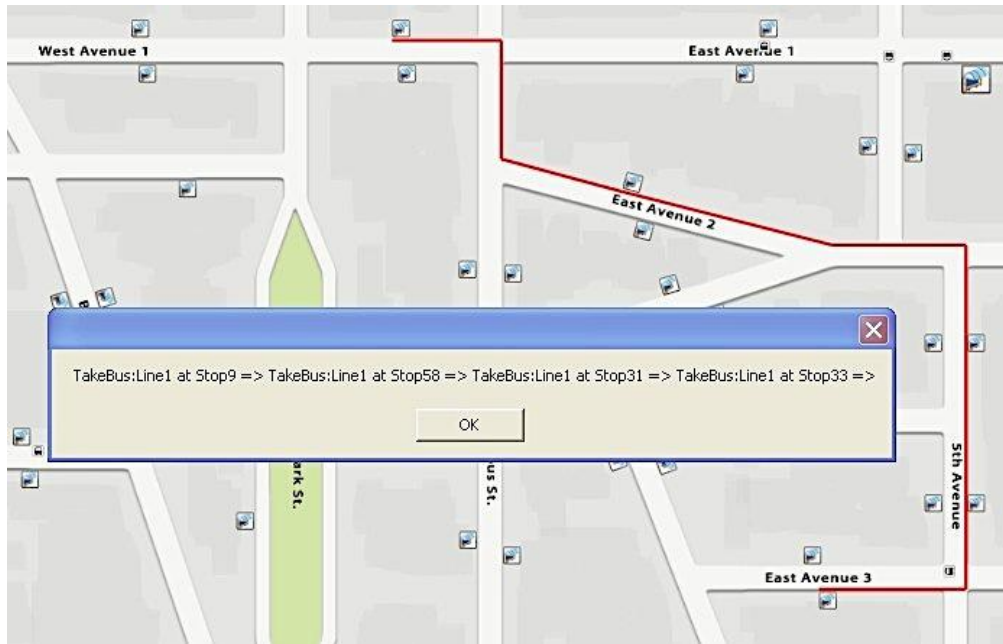


Figure 29 Route Planning Result

Appendix: CSP# model

```
enum{Initial,Chinatown,OutramPark,HarbourFront,TiongBahru,RafflesPlace,Bugis,DhobyGhaut,MarinaBay,Terminal};
enum{ArriveStop,AtTerminal,Moving};
enum{AtStop,OnBus};
enum{GetOn,GetOff};
enum{CreditCard,PrePay,OwnTicket,Warning};
enum{Deduct,Deposit,Change};

#define T_COST 2;

//channel to Bus_Info_Manager()
channel ch_bim_rec 0;
//channel to Account_Manager()
channel ch_acc_rec 0;
//channel to notify user though SMS
channel ch_SMS 10;
//*****
//*****

//Part 1: Bus Behavior
//The bus detection part is cut from the full version.
//It is because in this simplified model, there is only one user, no need to let bus
to detect.
//*****
//*****

var bus_status[2] = [AtTerminal(2)];
var bus_location[2] = [Initial(2)];

Bus_0() = start.0 (bus_status[0]=Moving)
-> arrive.0.Chinatown
{bus_status[0]=ArriveStop;bus_location[0]=Chinatown}
-> leave.0.Chinatown (bus_status[0]=Moving)
-> arrive.0.OutramPark
{bus_status[0]=ArriveStop;bus_location[0]=OutramPark}
-> leave.0.OutramPark (bus_status[0]=Moving)
-> arrive.0.HarbourFront
{bus_status[0]=AtTerminal;bus_location[0]=HarbourFront}
-> Skip;

Bus_1() = start.1 (bus_status[1]=Moving)
-> arrive.1.TiongBahru
{bus_status[1]=ArriveStop;bus_location[1]=TiongBahru}
-> leave.1.TiongBahru (bus_status[1]=Moving)
-> arrive.1.OutramPark
{bus_status[1]=ArriveStop;bus_location[1]=OutramPark}
-> leave.1.OutramPark (bus_status[1]=Moving)
-> arrive.1.RafflesPlace
{bus_status[1]=ArriveStop;bus_location[1]=RafflesPlace}
-> leave.1.RafflesPlace (bus_status[1]=Moving)
-> arrive.1.Bugis (bus_status[1]=AtTerminal;bus_location[1]=Bugis)
-> Skip;

Bus_Behavior() = Bus_0() ||| Bus_1();

//Part 2: User Behavior
//Unlike the full version, when a user gets on bus, a message will be automatically
sent to the server.
//*****
//*****

var user_status = AtStop;
var user_location = Chinatown;
var user_onbus = -1;
var user_geton_busstop = 0;

User_Geton_OP(bus,busstop) = [user_status==AtStop && user_location==busstop &&
bus_location[bus]==busstop && bus_status[bus]==ArriveStop]
```

```
        atomic{
            getonbus
{user_status=OnBus;user_onbus=bus;user_geton_busstop=busstop}
        -> ch_bim_rec!GetOn.bus.busstop //Inform the server
        -> Skip};

User_Geton() =
[]bus:{0..1}@([]busstop:{Chinatown..MarinaBay}@User_Geton_OP(bus,busstop));

User_Getoff_OP(bus,busstop) = [user_status==OnBus && user_onbus==bus &&
bus_location[bus]==busstop && bus_status[bus]==ArriveStop &&
user_geton_busstop!=busstop]
        atomic{
            getoffbus
{user_status=AtStop;user_location=busstop;user_onbus=-1}
        -> ch_bim_rec!GetOff.bus.busstop
        -> Skip};

User_Getoff() =
[]bus:{0..1}@([]busstop:{Chinatown..MarinaBay}@User_Getoff_OP(bus,busstop));
User_Behavior() = User_Geton();User_Behavior()
[]
User_Getoff();User_Behavior()
[]
Skip;

//*****
//*****
//*****

//Part 3: User Operation
//In order to reduce the states, here user can only do the operation once.
//*****
//*****
//*****
var user_pay_method = CreditCard;
var user_prepay_acc = 2;
var user_ticket = false;

User_Deposit() = ch_acc_rec!Deposit -> Skip [] Skip;
User_Change() = ch_acc_rec!Change -> Skip [] Skip;

User_Operation() = User_Deposit() ||| User_Change();
//*****
//*****
//*****

//Part 4: Bus Info Manager
//This part is to distribute coming messages from the bus(however, in this model, it's
from the user directly).
//*****
//*****
//*****
Bus_Info_Manager() = ch_bim_rec?x.y.z -> ifa(x==GetOn)
{
    ch_acc_rec!Deduct -> Bus_Info_Manager()
}
else
{
    Bus_Info_Manager()
}[]Skip;

//*****
//*****
//*****

//Part 5: Account Manager
//It's the unfixed version, which will lead to an error. (Details are in verification
part)
//*****
//*****
//*****
Account_Manager() = ch_acc_rec?x -> ifa(x==Deduct)
{
    ifa(user_pay_method==PrePay)
    {
        deduct_PP {user_prepay_acc =
user_prepay_acc-T_COST}
```

```
-> ch_SMS!PrePay
-> ifa(user_prepay_acc<=0)
{
    ch_SMS!Warning ->
}
else
{Account_Manager() }
}
else
{
    ifa(user_ticket==false) //Here is the
reason of the error. It should first check whether the user owns a ticket!
{
    deduct_CC -> getticket
{user_ticket=true} -> ch_SMS!CreditCard -> Account_Manager()
    }
    else
    {
        ch_SMS!OwnTicket ->
Account_Manager()
    }
}
}
else
{
    ifa(x==Deposit)
    {deposit {user_prepay_acc = user_prepay_acc+50}
    }
    else
    {change {user_pay_method = PrePay} ->
Account_Manager() }
} []Skip;

Server() = Bus_Info_Manager() ||| Account_Manager();
//*****
//Part 6: Fixed Account Manager
//Fixed the above part.
//*****
Account_Manager_fixed() = ch_acc_rec?x -> ifa(x==Deduct)
{
    if(user_ticket==false) //Here, we
FIRST to check whether a ticket is owned.
{
    ifa(user_pay_method==PrePay)
    {
        deduct_PP
{user_prepay_acc = user_prepay_acc-T_COST}
        -> ch_SMS!PrePay
        ->
    }
    ch_SMS!Warning -> Account_Manager_fixed()
    }
    else
    {Account_Manager_fixed() }
}
else
{
    deduct_CC ->
getticket {user_ticket=true} -> ch_SMS!CreditCard -> Account_Manager_fixed()
    }
}
else
{
    ch_SMS!OwnTicket ->
Account_Manager_fixed()
}
}
```

```

    }
    else
    {
        ifa(x==Deposit)
        {deposit {user_prepay_acc
= user_prepay_acc+50} -> Account_Manager_fixed() }
        else
        {change {user_pay_method =
PrePay} -> Account_Manager_fixed() }
    } [] Skip;

Server_fixed() = Bus_Info_Manager() ||| Account_Manager_fixed();
//*****
//*****
//*****
//Part 7: Assertions
//*****
//*****
//*****
Nature_Behavior() = User_Behavior() ||| Bus_Behavior();
System() = Nature_Behavior() ||| User_Operation() ||| Server();
System_fixed() = Nature_Behavior() ||| User_Operation() ||| Server_fixed();

//To check whether the system is deadlock-free
#assert System() deadlockfree; //YES
#assert System_fixed() deadlockfree; //YES

//Response property 1
//The user must be informed after getting on a bus
#assert System() != [] (getonbus -> <> (ch_SMS!1 || ch_SMS!0 || ch_SMS!2)); //YES
#assert System_fixed() != [] (getonbus -> <> (ch_SMS!1 || ch_SMS!0 || ch_SMS!2)); //YES

//Response property 2
//The user must be warned when the balance is insufficient
#define state_bns user_prepay_acc<=0;
#assert System() != <> (state_bns -> <> ch_SMS!3); //YES
#assert System_fixed() != <> (state_bns -> <> ch_SMS!3); //YES

//Reliability property 1
//The user can not be charged when hi/she owns a ticket
#define state_ot user_ticket == true;
#assert System() != [] !(state_ot && (deduct_PP || deduct_CC)); //NO
#assert System_fixed() != [] !(state_ot && (deduct_PP || deduct_CC)); //YES

//Reliability property 2
//Account balance will be increased when the user deposits
#define state_bd user_prepay_acc>=50;
#assert System() != [] (ch_acc_rec!Deposit -> <> state_bd); //YES
#assert System_fixed() != [] (ch_acc_rec!Deposit -> <> state_bd); //YES
```

References

- 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.
- Fischer, C., & Wehrheim, H. (1999). Model-Checking CSP-OZ Specifications with FDR. *The 1st International Conference on Integrated Formal Methods* (pp. 315-334). London: Springer-Verlag.
- Hoare, C. A. (1978). Communicating sequential processes. *Commun. ACM*, 666--677.
- 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.