Test Plan

For

Transport4You

Version 1.0 approved

Prepared by Li Yi, Yang Hang and Wu Huanan

National University of Singapore

Created on 18/12/2010

Revision History

Date	Description	Author	Comments
18/12/2010	Version 0.9	Yang Hang	First Draft
27/02/2011	Version 1.0	Yang Hang	Final Draft

Document Approval

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

Signature	Name	Title	Date
	Wu Huanan	Test Manager	28/02/2011

Table of Contents

1 Introduction

1.1 Purpose

The purpose of this Test Plan (TP) is to achieve the following:

- Define testing strategies for each area and sub-area to include all the functional and quality requirements.
- Divide Design Specifications into testable areas and sub-areas. Identify and include areas that are to be omitted (not tested) also.
- Identify required resources and related information.
- Provide testing Schedule.
- Ensure the Fault Correction Module achieve satisfactory percentage of correct results.
- Ensure the Route Plan Module give the most preferred route.
- Guarantee the software usability meet the requirement (mentioned in section 2.5).

1.2 Intended Audiences

The intended audiences of stakeholders for this test plan of the IPTM include:

- Transport4You IPTM system development team:
  - Project Manager.
  - Developers, who have to write part of the test cases and do unit and integration test according to the TP.
  - Testers, who must validate the system according to the TP.
- Transport4You IPTM system project stakeholders (clients):
  - Mangers.
  - Customer Representatives, who must approve it.

1.3 Document Organization

This document is organized into the following sections:

- Introduction, which introduces the test plan for the IPTM system to its readers.
- Testing Methodology
- Testing Report
- Testing Environment
- Testing Schedule
- Appendices

2. Testing Methodology

This section introduces the testing methodologies that we used in the IPTM system testing.

2.1 PAT Guided Test Overview

PAT supports user defined C# library. Namely, an object or method defined in PAT's C# library can be invoked as a part of the CSP# model. This creates a way of linking events of the interface models with codes expected to be tested (Sun, Liu, & Cheng, 2010).

PAT guided test is beyond the scope of traditional model checking, as it guides the validation of not only the design but also the implementation. PAT guided test is superior to traditional testing especially for concurrent systems. Usually concurrent systems are very difficult for testing, as the generation of test cases requires a large amount of efforts. However, PAT is able to automatically generate behavioral traces for concurrent systems like our user-bus and bus-server systems. Therefore, a large portion of our testing is done using PAT guided test.

There are two ways to test using PAT. One is using self-defined C# library to invoke the subject functions to be tested, while the other one is more general which uses C# library to create processes containing functional codes and all related resources. The further illustration is as follows.

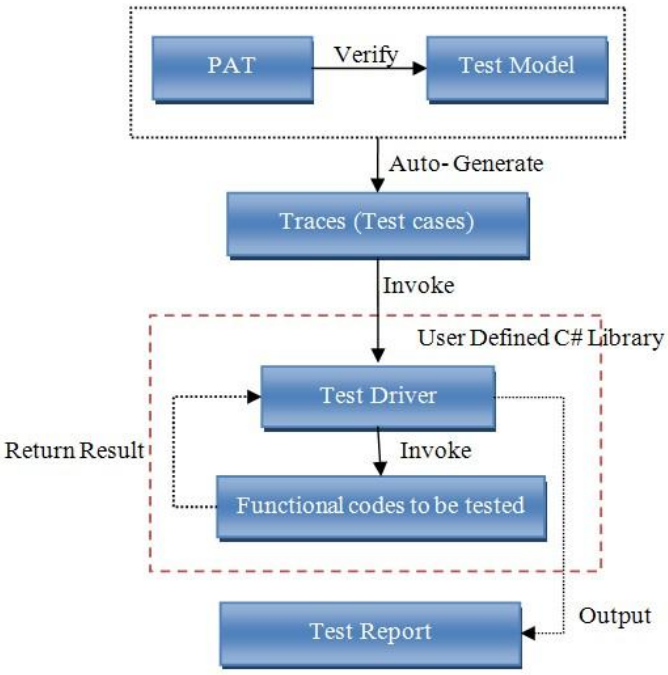

Figure 1 PAT Guided Test 1

In the first case, functions that need to be tested are extracted from the programs and complied with PAT interface library code into a .dll file. The .dll file acts as a test driver and is invoked by PAT when the related event appears in the model checking traces. The model checking traces are automatically generated by a test model which is written in CSP# by testers. The test model describes all the possible inputs and behaviors of the environment, such that the test cases generated have a broad coverage.

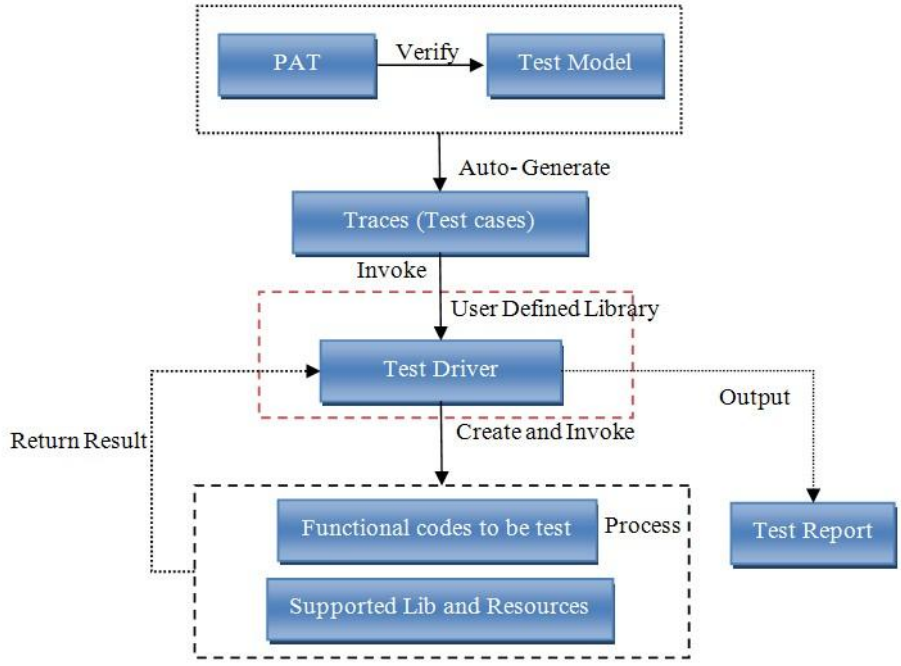

Figure 2 PAT Guided Test 2

The main difference of the second way is that the test subjects become executables which will be running as processes. Furthermore, unlike the first way, the pre-compiled .dll file does not contain functional codes. Instead, the functional codes are compiled into executables.

The second PAT guided test approach is more flexible than the first one. It is because that in first case, subject codes to be tested cannot reference to other resources except for PAT predefined libraries. However, the first approach is easier in the sense that except for the CSP# test model, no extra efforts are needed while in the second case, subject codes have to be compiled into executables beforehand.

Code Contracts which validate important properties during runtime are also used to further ensure the reliability of the system in our testing. Pre-conditions, post-conditions and invariants are added to some critical functions. Therefore, when PAT evokes a certain function with Code Contracts, all violations of the contracts will be reported to the testers. An

example of the use of Code Contracts in the *ModifyUser* function is show below:

```
[ContractInvariantMethod]
void AccountInvariant()
{
        //invariant
        Contract.Invariant(ServerConfig.GetT_COST() > 0 &&
ServerConfig.GetT_VAL() > 0, "T_COST AND T_VAL is illegal!");
}

public static void ModifyUser(UserInfo userInfo)
{
        //pre-conditions
        Contract.Requires(userInfo != null, "user info is null!");
        Contract.Requires(userInfo.name.Length > 0, "user name length is
        zero!");
        Contract.Requires(userInfo.cellPhoneNumber.Length > 0 , "user
        cellphone number length is zero!");
        Contract.Requires(ContainsUser(userInfo.cellPhoneAddr), "try to access
        a unregistered user!");

        //post-conditions
        Contract.Ensures(userBase[userInfo.cellPhoneAddr] == userInfo,
        "modification   failed!");

        if (ContainsUser(userInfo.cellPhoneAddr))
        {
                lock (userBase)
                {
                        userBase[userInfo.cellPhoneAddr] = userInfo;
                        saveUserBase();
                }
        }
}
```

The pre-conditions in the above example ensure that the *userInfo* is valid before the modification. The invariant ensures that *T_COST* and *T_VAL* values are legal. The post-conditions ensure that after modification, the new value is updated to the input value.

2.2 Test Fault Correction Module

To test the Fault Correction module, the first approach is adopted. We design a CSP# test model that describes the natural behaviors of buses and users. Each user can choose any starting point, destination point, and any bus to get on. Buses may travel in parallel such that a user can be detected by multiple buses at the same time. The fault correction strategy is able to reduce all the situations to the case where only two buses detect the same user. The test model covers all the possible cases for two bus lines, and therefore it covers all the bus detection patterns.

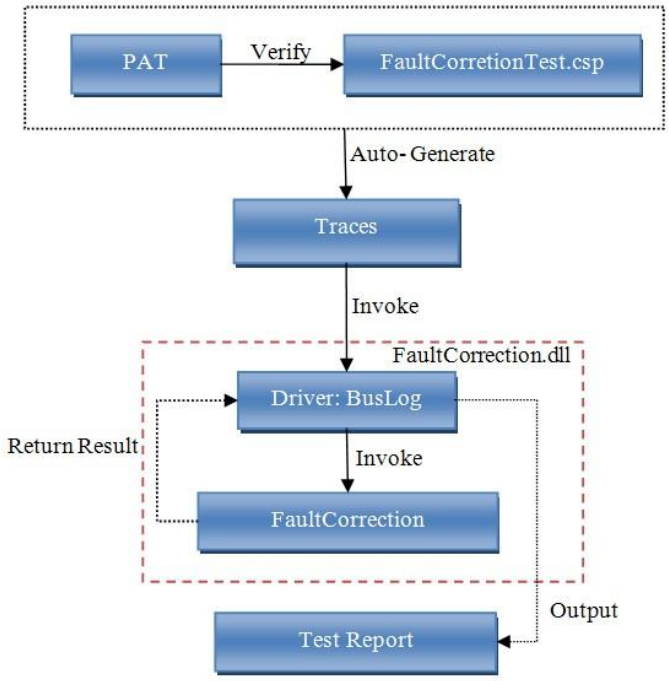

Figure 3 Fault Correction Testing Flow

There are three important processes in the test model, *FalutCorrectionTest.csp*: *Bus_2*, *Bus_3*, and *User*. *Bus_2* travels through *StopA*, *StopB*, *StopC*, *StopD* and *Bus_3* travels through *StopB*, *StopC*, *StopE*, *StopF*. User randomly arranges his or her trip between *StopA* to *StopF*.

Part of the test model is shown below:

```
1 #import "FaultCorrectionTest";
2 #define TOTAL_STATES 14;
3 enum{initial,StopA,StopB,StopC,StopD,StopE,StopF };
4 enum{ini,AtStop,Moving};
5 var bus_location[4]=[0(4)];
6 var bus_status[4]=[0(4)];
7 var useron_flag[4]=[0(4)];
8 var user_location=StopA;
9 var onbus=0;
10 var<BusLog> log = new BusLog(TOTAL_STATES);
11 var trace[0];
12 var useron_stop;
13 var useron_bus[4]=[0(4)];
```

- Line 1 imports the user defined DLL "*FaultCorrectionTest.dll*" into the model.
- Line2 defines the total number of bus states. For each bus stop, there are two states, namely "arriving" and "leaving". Thus, there are 14 states in total.

- Line 10 creates an instance named log of the user defined DLL class, which makes test driver class *BusLog* accessible to the test model.

```
14 Bus_2() = get.2.StopA
15              {
16                      trace=log.SetEvent(1200+StopA);
17                      bus_location[2]=StopA;
18                      bus_status[2]=AtStop;
19              } ->
20              leave.2.StopA
21              {
22                       trace=log.SetEvent(2200+StopA);
23                       bus_status[2]=Moving;
24                       if(bus_location[2]==bus_location[onbus] &&
bus_status[onbus]==Moving)
25                       {
26                              if(useron_flag[2]==0)
27                              {useron_flag[2]=1;log.AddRecord(2,StopA,1)}
28                       }
29                       else
30                       {
31                                if(useron_flag[2]==1)
32                                {useron_flag[2]=0;log.AddRecord(2,StopA,0)}
33                       }
34              }
```

The part above describes the bus moving behaviors. Line 14 to 19 define the actions of Bus_2 arriving at *StopA*. Line 20 to 34 defines actions of *Bus_2* leaving from *StopA*.

```
184 User_GetOn() =
[]bus:{2..3}@([]busstop:{StopA..StopF}@User_GetOn_OP(bus,busstop));
185 User_GetOff() =
[]bus:{2..3}@([]busstop:{StopA..StopF}@User_GetOff_OP(bus,busstop));
186 User() = (User_GetOn();User_GetOff();User())[]Skip;
187 Sys() = Bus_2() ||| Bus_3() ||| User();
188 #assert Sys() deadlockfree;
```

- Line 184 to 186 defines the user behaviors.
- Line 187 defines the system behaviors as the interleaving of two buses and one user.
- Line 188 is a deadlock free assertion which traverses all the traces and automatically generates test cases.

Descriptions of the functions in the user defined C# library *"FaultCorrection.dll"* are as follows:

Function	Descriptions
SetEvent()	Test model uses this function to inform the Test Driver what the current event is.
AddFlagRecord()	Log the boarding and alighting behaviors of user.
GetClone()	PAT needs this function to clone the current state for model checking purpose.
FlushEventLeft()	When a trace comes to the terminal state, this function outputs the testing report for the current trace.

2.3 Test Route Plan Module

The testing for the route plan module uses the second PAT guided testing approach, as external library is used in the testing. In the first place, an open source .NET graph algorithm library *QuickGraph* is used to calculate the length of the shortest path in the given map. Then this length is compared with the length of the suggested path given by the Route Plan module developed by ourselves.

First the test model generates all pairs of start point and end point. Then the test driver "*PatServiceTest*" is invoked by PAT for each pair. The test driver creates two processes, both calculating the path length. The test report is then produced after the comparison of their result.

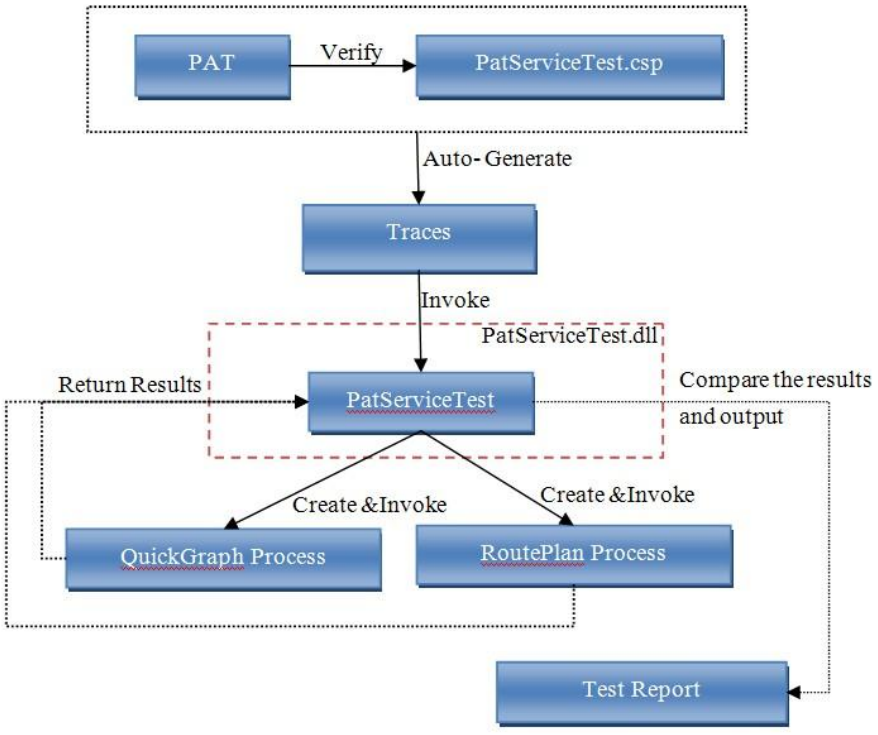

Figure 4 Route Plan Testing Flow

Part of the test model is given as follows:

```
1 #import "PatServiceTest";
2 var service = new PatServiceTest();

9 StartPoint(i) = {startStop = i} -> Skip;
10 EndPoint(i) = {endStop = i;} ->ifa(startStop == endStop){Skip}
else{trip.startStop.endStop{service.TestMethod(startStop,endStop)}->Skip};

11 System() = StartProcess(); EndProcess();
12 #assert System() deadlockfree;
```

- Line 1 imports user defined DLL *"PatServiceTest.dll"*.
- Line 2 creates an instance of the *PatServiceTest* class, which makes the test driver accessible to the model.
- Line 9 to 10 generates test cases.
- Line 12 is a deadlock free assertion which traverses all the traces and automatically generates test cases.

Descriptions of the functions in the user defined C# library *"PatServiceTest.dll"* are as follows:

Function	Descriptions
TestMethod()	Invokes *CreatePorcess()*, do comparison and output report.
CreateProcess()	Create new processes to be tested.

2.4 Unit Testing

Unit testing is carried out continuously to ensure that the software is bug-free and meets the requirement specifications. Unit testing is conducted by developers during code development process to ensure that proper functionality and code coverage have been achieved by each developer. Below shows the major functions tested by unit testing:

Important Functions	Expected Result	Test Result
SeverConfig:: ModifyT_COST	Read/Modify System parameter T_COST and T_VAL Correctly	OK
ServerConfig::ModifyT_VAL		
ServerConfig:: GetT_COST		
ServerConfig:: GetT_VAL		
AccountManager::AddUser	Can add new user info into System	OK
AccountManager::DeleteUser	Can delete existing user info from System	OK
AccountManager::QueryUser	Can get existing user info from System	OK
AccountManager::ModifyUser	Can modify user info of System	OK

AccountManager::ChargeUser	Can deduct account balance of user	OK
ServerNetComm::Listen	Server communicates	OK
ServerNetComm::Send		
BusReportManger::ManageReport	Server receives reports and charges user correctly when there is no need to do fault correction	OK
TripManger::AddRecord	Server can log user trip info correctly	OK
BustNetComm:: SendMsg	Bus component sends report to server correctly	OK

2.5 Usability Testing

Development will typically create a non-functioning prototype of the UI components to evaluate the proposed design. Usability testing can be coordinated by testers, but actual testing must be performed by non-testers. For the IPTM system, we invited some students from our school to actually use the software. After that, we summarized the suggestions and made modifications according to the reviews.

The user portal of the IPTM system enables at least 90% of a statistically valid sample of representative experienced users to:

- Register as a subscriber within 2.5 minutes.
- Modify his or her personal particulars within 1 minute.
- Modify his or her account settings within 1 minute.
- Add value to his or her account within 1.5 minutes.

3. Testing Report

3.1 Test Cases

3.1.1 Fault Correction Test Cases

The bus line configurations for test case generation are shown below:

Bus Line 1: StopA->StopB->StopC->StopD
Bus Line 2: StopB->StopC->StopE->StopF

The test coverage is 100% for the case where two buses travel in between six bus stops. The set of test cases is also representative for more buses and more bus stops based on the explanation in subsection 2.2. 51350 test cases are automatically 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*" indicate the detailed execution sequence of the test model for reference purpose. 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 "*StopA*. "*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

3.1.2 Route Plan Test Cases

The test cases covers all start and end bus stop combinations. Therefore the test coverage is 100% for the map configuration in the Transport4You simulator. The results should also be representative for other map configurations. 3660 test cases are automatically generated. Meanwhile, all the test results are acquired. The following is one detailed test log.

7: pass; From TerminalA to Stop53; PAT RoutePlan Length: 7; Dijkstra Aloghrithm Length: 7; PAT actionList: TakeBus:Line1 at TerminalA => TakeBus:Line1 at Stop5 => TakeBus:Line1 at Stop7 => TakeBus:Line1 at Stop9 => TakeBus:Line1 at Stop58 => TakeBus:Line1 at Stop31 => TakeBus:Line1 at Stop33 => ; Dijkstra Aloghrithm actionList:TerminalA=>Stop5=>Stop7=>Stop9=>Stop58=>Stop31=>Stop33=>Stop53

In the above test log, "*7*" is the test case ID. "*pass*" means this test case has passed. It is followed by the comparison of the results from *RoutePlan* and the *Dijkstra Shortest Path Algorithm* (*QuickGraph* library).

3.2 Test Results

3.2.1 Fault Correction Test Results

Statistics of the test results are shown below:

Correct fare cases	Count	Percentage
Correct route and correct fare	48728	94.9%
Incorrect route and correct fare	1428	2.8%
Incorrect fare cases	Count	Percentage
Incorrect route and less fare	1194	2.3%
Incorrect route and more fare	0	0%
Summary	Count	Percentage
Total Correct fare cases	50156	97.7%
All cases	51350	100%

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

3.2.2 Route Plan Test Results

Statistics of the test results are shown below:

	Count	Percentage
Pass	3509	95.9%
Fail	151	4.1%

For the failed test cases, *RoutePlan* module returns a longer path than that is returned by the *Dijkstra Shortest Path Algorithm*. The average difference in path length is 1.72. However, *RoutePlan* module not only aims for the shortest path, but also takes the number of bus changes into consideration, which is the reason for the failed cases. The planning algorithm can be further improved in the future to produce optimal suggested paths.

4. Testing Environment

4.1 Tools and External Libraries

Below shows the tools and external libraries used in the testing.

Tool	Usage
PAT	PAT guided testing
Visual Studio	Unit testing
Library	Usage
QuickGraph	Provide shortest path for comparison with Route Plan

4.2 System Configurations

Below shows the system configurations of the testing machine:

CPU	Intel core2 Duo T5450 @ 1.66GHZ
RAM	2.5G DDRII
OS	WINDOWS 7 Professional 32bit

5. Testing Schedule

Below shows the testing schedule:

Round	TimeLine	Tasks
1	After 1st iteration	Unit testing
2	After 2nd iteration	PAT guided testing and usability testing

6. Appendices

This section documents the following appendices:

- CSP# test model: *FaultCorrectionTest.csp*
- CSP# test model: *PatServiceTest.csp*
- Test driver implementation: *FaultCorrectionTest.cs*
- Test driver implementation: *PatServiceTest.cs*

6.1 CSP# test model: FaultCorrectionTest.csp

```
#import "FaultCorrectionTest";

#define TOTAL_STATES 14;

enum{initial,StopA,StopB,StopC,StopD,StopE,StopF,StopG,StopH};
enum{ini,AtStop,Moving};

var bus_location[4]=[0(4)];
var bus_status[4]=[0(4)];
var useron_flag[4]=[0(4)];
var user_location=StopA;
var onbus=0;
var<BusLog> log = new BusLog(TOTAL_STATES);
var trace[0];
var useron_stop;
var useron_bus[4]=[0(4)];

Bus_2() = get.2.StopA
                {
                        trace=log.SetEvent(1200+StopA);
                    bus_location[2]=StopA;
                    bus_status[2]=AtStop;
                } ->
                leave.2.StopA
                {
                        trace=log.SetEvent(2200+StopA);
                        bus_status[2]=Moving;
                        if(bus_location[2]==bus_location[onbus] && bus_status[onbus]==Moving)
                        {
                            if(useron_flag[2]==0)
                            {useron_flag[2]=1;log.AddRecord(2,StopA,1)}
                        }
                        else
                        {
                                if(useron_flag[2]==1)
                                {useron_flag[2]=0;log.AddRecord(2,StopA,0)}
                        }
                } ->
                get.2.StopB
                {
                        trace=log.SetEvent(1200+StopB);
                    bus_location[2]=StopB;
                    bus_status[2]=AtStop;
                } ->
                leave.2.StopB
                {
                        trace=log.SetEvent(2200+StopA);
                        bus_status[2]=Moving;
                        if(bus_location[2]==bus_location[onbus] && bus_status[onbus]==Moving)
                        {
```

```
                        if(useron_flag[2]==0)
                        {useron_flag[2]=1;log.AddRecord(2,StopB,1)}
                    }
                    else
                    {
                            if(useron_flag[2]==1)
                            {useron_flag[2]=0;log.AddRecord(2,StopB,0)}
                    }
                } ->
                get.2.StopC
                {
                        trace=log.SetEvent(1200+StopC);
                    bus_location[2]=StopC;
                    bus_status[2]=AtStop;
                } ->
                leave.2.StopC
                {
                    trace=log.SetEvent(2200+StopC);
                    bus_status[2]=Moving;
                    if(bus_location[2]==bus_location[onbus] && bus_status[onbus]==Moving)
                            {
                                if(useron_flag[2]==0)
                                {useron_flag[2]=1;log.AddRecord(2,StopC,1)}
                            }
                            else
                            {
                                    if(useron_flag[2]==1)
                                    {useron_flag[2]=0;log.AddRecord(2,StopC,0)}
                            }
                } ->
                get.2.StopD
                {
                        trace=log.SetEvent(1200+StopD);
                    bus_location[2]=StopD;
                    bus_status[2]=AtStop;
                    if(useron_flag[2]==1)
                    {useron_flag[2]=0;log.AddRecord(2,StopD,0)};
                    log.FlushEventLeft();
                } ->
                Skip;

Bus_3() = get.3.StopB
                {
                    trace=log.SetEvent(1300+StopB);
                    bus_location[3]=StopB;
                    bus_status[3]=AtStop;
                } ->
                leave.3.StopB
                {
                    trace=log.SetEvent(2300+StopB);
                    bus_status[3]=Moving;
                    if(bus_location[3]==bus_location[onbus] && bus_status[onbus]==Moving)
                            {
                                if(useron_flag[3]==0)
                                {useron_flag[3]=1;log.AddRecord(3,StopB,1)}
                            }
                            else
                            {
                                    if(useron_flag[3]==1)
                                    {useron_flag[3]=0;log.AddRecord(3,StopB,0)}
                            }
                } ->
                get.3.StopC
                {
                    trace=log.SetEvent(1300+StopC);
                    bus_location[3]=StopC;
```

```
        bus_status[3]=AtStop;
    } ->
    leave.3.StopC
    {
        trace=log.SetEvent(2300+StopC);
        bus_status[3]=Moving;
        if(bus_location[3]==bus_location[onbus] && bus_status[onbus]==Moving)
                {
                    if(useron_flag[3]==0)
                    {useron_flag[3]=1;log.AddRecord(3,StopC,1)}
                }
                else
                {
                        if(useron_flag[3]==1)
                        {useron_flag[3]=0;log.AddRecord(3,StopC,0)}
                }
    } ->
    get.3.StopE
    {
        trace=log.SetEvent(1300+StopE);
        bus_location[3]=StopE;
        bus_status[3]=AtStop;
    } ->
    leave.3.StopE
    {
        trace=log.SetEvent(2300+StopE);
        bus_status[3]=Moving;
        if(bus_location[3]==bus_location[onbus] && bus_status[onbus]==Moving)
                {
                    if(useron_flag[3]==0)
                    {useron_flag[3]=1;log.AddRecord(3,StopE,1)}
                }
                else
                {
                     if(useron_flag[3]==1)
                     {useron_flag[3]=0;log.AddRecord(3,StopE,0)}
                }
    } ->
    get.3.StopF
    {
        trace=log.SetEvent(1300+StopF);
        bus_location[3]=StopF;
        bus_status[3]=AtStop;
        if(useron_flag[3]==1)
        {useron_flag[3]=0;log.AddRecord(3,StopF,0)};
        log.FlushEventLeft();
    } ->
    Skip;

User_GetOn_OP(bus,busstop) = [onbus==0 && user_location==busstop &&
bus_location[bus]==busstop && useron_bus[bus]==0 && bus_status[bus]==AtStop]
                                                        geton.bus.busstop
                                                        {

trace=log.SetEvent(8000+bus*100+busstop);
                                                            onbus=bus;
                                                            user_location=0;
                                                            useron_stop=busstop;
                                                            useron_bus[bus]=1;
                                                            log.AddFlagRecord(bus,busstop,1);
                                                            log.FlushEventLeft();
                                                        } -> Skip;

User_GetOff_OP(bus,busstop) = [onbus==bus && useron_stop!=busstop &&
bus_location[bus]==busstop && bus_status[bus]==AtStop]
                                                         getoff.bus.busstop
```

```
                                                    {

trace=log.SetEvent(9000+bus*100+busstop);
                                                        onbus=0;
                                                        user_location=busstop;
                                                        log.AddFlagRecord(bus,busstop,0);
                                                        log.FlushEventLeft();
                                                    } -> Skip;

User_GetOn() = []bus:{2..3}@([]busstop:{StopA..StopH}@User_GetOn_OP(bus,busstop));
User_GetOff() = []bus:{2..3}@([]busstop:{StopA..StopH}@User_GetOff_OP(bus,busstop));

User() = (User_GetOn();User_GetOff();User())[]Skip;

Sys() = Bus_2() ||| Bus_3() ||| User();

#assert Sys() deadlockfree;
```

6.2 CSP# test model: PatServiceTest.csp

```
#import "PatServiceTest";

var service = new PatServiceTest();

#define stopN 61;

var startStop = 0;
var endStop = 0;

StartProcess() = []x:{0..(stopN-1)}@StartPoint(x);
EndProcess() = []y:{0..(stopN-1)}@EndPoint(y);

StartPoint(i) = {startStop = i} -> Skip;

EndPoint(i) = {endStop = i;} ->ifa(startStop == endStop){Skip}
else{trip.startStop.endStop{service.TestMethod(startStop,endStop)}->Skip};

System() = StartProcess(); EndProcess();

#assert System() deadlockfree;
```

6.3 Test driver implementation: FaultCorrectionTest.cs

The source code file is located under the root folder of the submission package.

6.4 Test driver implementation: PatServiceTest.cs

The source code file is located under the root folder of the submission package.

Works Cited

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.