CS3211 Project Progress Report
Group 5

Ong Jit Sheng, Jonathan A0073924B
Tran Minh Hoang A0088452X
Le Vu Nguyen Chuong A0088552W

Contents

I. CSP/PAT model 3
AllATM process 3
AllCPU process 3
Database process 4
The unreliable network 4
The dealock version 4
The Incorrect calculation version 5
II. Assertions 5
III. Java simulator 7

I. CSP/PAT model

The model consists of three main processes: AllATM(), AllCPU and Database.

AllATM process

AllATM() process is the parallel combination of N (a constant) process ATM(i). Each of these process will go to Authenticate.

- The Authenticate process sends the authentication request to the corresponding CPU, then waits for the response (success or fail) or a timeout signal (through shared action). If the authentication is successful, then the ATM process will go to either WIthdraw process or CheckBalance process. If it fails or timeout, the Authenticate process will go back to its start.
- The Withdraw process sends the amount of money need to withdraw to the corresponding CPU, then waits for the response (success or fail) or a timeout signal (through shared action). If the withdraw is success (the account balance is enough to perform the withdraw), money will be sent out. If a timeout signal is received, the process will go back to its start to send again. If the Withdraw is failed, the process will go to ATM process.
- The CheckBalance process sends a check balance request to the corresponding CPU, then wait for the response (success or fail) or a timeout signal (through shared action). If the CheckBalance process is successful, the current balance will be showed to the user. If one of the two ways communication fails, the CheckBalance process will try to send again by going back to its start.

AllCPU process

AllCPU() process is the parallel combination of N (a constant) process CPU(i). Each of these process will go to either CPUAuthen, CPUWithdraw or CPUCheckBalance, depend on the request from the ATM.

- The CPUAuthen is the process in CPU to perform the authenticate action. CPU process will go to the CPUAuthen if it received the request from the ATM through the shared action sendAuthenSuccess. In its turn, the CPUAuthen will send an authentication request to the Database then wait for the response (success or fail) or a timeout signal. If the response is received, the result will be sent back to the ATM through a shared action. If one of the two ways communication fails, the CPUAuthen process will try to send again by going back to its start.

- The CPUWithdraw is the process in CPU to perform the withdraw action. CPU will go to the CPUWithdraw if it received the request from the ATM through the shared action sendAmountSuccess. In its turn, the CPUWithdraw will send a withdraw request to the Databse then wait for the response (success or fail) or a timeout signal. If the response is received, the result will be sent back to the ATM through a shared action. If one of the two ways communication fails, the CPUWithdraw process will try to send again by going back to its start.
- The CPUCheckbalance is the process in CPU to perform the check balance process. CPU will go to CPUCheckBalance if it received the request from the ATM through the shared action sendCBSuccess. In its turn, the CPUCheckBalance will send a check balance request to the Database then wait for the response (success or fail) or a timeout signal. If the response is received, the result will be sent back to the ATM through a shared action. If one of the two-way communication fails, the CPUCheckBalance process will try to send again by going back to its start.

Database process

Similar to the CPU received the request from the ATM through shared actions, the Database also received request from the CPU then go to one of the three process: DBAuthen, DBWithdraw or DBCheckBalance.

- The DBAuthen will be either success or fail, then send the result to the CPU
- The DBWithdraw will check if the account balance is sufficient for the withdrawal action, then return the result to the CPU.
- The DBCheckBalance will simply return the current balance to the CPU

The unreliable network

The unreliable network is modeled by giving an option of fail in every communication link. Each time a communication link fail, the sender will receive a send-fail signal and the device that is listening to that information will received a timeout signal. Then the sender will try to resend or go to its main process, depends on the situation.

The dealock version

In this version, the deadlock can be introduced if the system doesn't handle the case that the communication protocol has errors. For instance, when the cloud processing unit connects to the database for authenticating purpose, if the message sent to the database is lost and the cloud processing unit doesn't sent again but wait for a response instead, there will be a deadlock. The cloud processing unit is waiting for a response from the database while the database is waiting for a message from the cloud processing unit (as the previous database is lost).

Here is the screen shot to see that the assertion that the system is deadlockfree is INVALID:

```
********Verification Result********
The Assertion (BankingSystem() deadlockfree) is NOT valid.
The following trace leads to a deadlock situation.
<init -> authen.1 -> sendAuthenSuccess.1 -> sendCheckUserDB.1 -> sendCheckUserDBSuccess.1 -> sendCheckUserDBResult.1 -> sendCheckUserDBResultFail.1
-> authen.0 -> sendAuthenSuccess.0 -> sendCheckUserDB.0 -> sendCheckUserDBSuccess.0 -> sendCheckUserDBResult.0 -> sendCheckUserDBResultFail.0>
```

The Incorrect calculation version

The incorrect calculation because of the shared temperature variable between different processes when updating the account balance after a money-withdraw is performed. The account balance is copied to variable temp, then the variable temp is changed because of the money withdraw before the value is copied back to the account balance. Because the temp variable is shared, the incorrect calculation will occur if the temp is used by a second process before the first process finished using it.

The problem can be eliminated by eliminate the using of temp. The account balance will be changed directly.

Here is the screen shot of the line that gives out incorrect calculation:

```
DBUpdate(i) = withdrawTotalupdate.i {totalWithdraw = totalWithdraw + amount }  ->update1.i {temp = accBalances[i]}
-> update2.i {temp = temp - amount} -> update3.i {accBalances[i] = temp} -> Database(i);
```

We do an assertion with two accounts trying to withdraw money and proving through invalid that the database doesn't update the account correctly:

```
********Verification Result********
The Assertion (AllDatabase() |= [](update3.0-> X SameAmount)) is NOT valid.
A counterexample is presented as follows.
```

We will elaborate more about the assertion to check the incorrect calculation version in the Assertions part below:

II. Assertions

The assertions can either be valid or invalid but is all to check if the system is working

correctly:
1/ #assert BankingSystem() deadlockfree;
Mean that the system cannot deadlock. For the true version, it will be VALID and INVALID in the version that introduce deadlock

2/ #assert BankingSystem()|= [](sendAmountSuccess.0 && withdrawSuccess.0 && sendChangeBalanceResultSuccess.0 -> <>moneyOut.0);
This assertion should return VALID in the right version. It says that if check that if user successfully send command to withdraw money and the process is successfully handled, then money will come out

3/ #assert BankingSystem()|= []!((sendAuthenFail.0 || authenFail.0 || getAuthenTimeout.0) && moneyOut.0))
The assertion should return VALID. If there is an error in authentication process then money will not come out.

4/ #assert BankingSystem()|= [](authenSuccess.0 -> <> moneyOut.0);
The assertion will return INVALID. It is possible that no communication might take place after authentication

5/ #assert BankingSystem()|= []!(authenSuccess.0 && authenFail.0);
The assertion will return VALID. Any account can be authenticated successfully or authenticate fail but it can never be in both states

6/ #assert BankingSystem()|= [](authen.0 -> <> (authenFail.0 || authenSuccess.0))
The assertion will return INVALID. Although the result for authentication can only be either authenFail or authenSuccess, but if there is an error in communication that makes the system continously send authentication message and resend but not being able to produce a right result

7/ #assert BankingSystem()|= [](!sendCheckBalance.0 -> <> returnBalance.0);
Assertion should return INVALID. If the user doesn’t send command to check balance, the system won’t return the balance to check.

8/ This assertion test a successful scenario for a user:
Test1() = authen.0 -> sendAuthenSuccess.0 -> getAuthenRespond.0 -> authenSuccess.0 -> sendAmount.0 -> sendAmountSuccess.0 -> getWithdrawRespond.0 -> withdrawSuccess.0 -> moneyOut.0 -> Test1();
#assert Test1() refines AllATM();
The assertion will return VALID

9/ #assert BankingSystem()|= [](authen.0 -> <> (authenFail.0));
The assertion will return INVALID. When we authenticate, it doesn’t always return authenticationFail

10/ #assert BankingSystem()|=[] !(sendCheckBalance.0 && moneyOut.0);
The assertion will return valid. If the command is to check the current balance, then the ATM will not give out the money for the user (it is not a withdraw money command)

11/ #assert AllATM() |= [](sendAmountSuccess.0 -> ((<> getWithdrawRespond.0) || (<> getWithdrawTimeOut.0)));
The assertion will return invalid. Even if the sendAmount to give the amount of money user wants to withdraw is successful. The system may be stuck in a live lock if the cloud processing unit is trapped in a loop of another ATM continuously fail to authenticate.

12/ This assertion check that if a user withdraw some money, then the current account balance will have less money than the initial amount:
#define lessMoney (accBalances[0] < InitMoney);
#assert BankingSystem() |= [](withdrawSuccess.0 -> lessMoney);

For the incorrect calculation version:

We do the assertion with a system with two ATM (or 2 users) each trying to withdraw money. The totalWithdraw variable is to hold the total money that the two users withdraw. We define:

#define SameAmount (accBalances[0] + accBalances[1] + totalWithdraw == N*100);

As initially, each account has 100, N*100 = 200, (N = 2). The sum of the two accounts and the total withdraw money has to be 200.
We will write an assertion that after the database update an account (here we use the first account (account 0), after the event update3.0 which indicating the database finishes updating) the current balance because of a withdraw money command, the total amount has to be the same:

#assert AllDatabase() |= [](update3.0 -> X SameAmount);

The assertion will return INVALID for the incorrect calculation version.

III. Java simulator