CS3211 Project Report
Group 5

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

Contents

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) processes ATM(i). Each of these processes 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 the Withdraw process or CheckBalance process. If it fails or times out, the Authenticate process will go back to its start.
- The Withdraw process sends the amount of money needed 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 a 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 the ATM process.
- The CheckBalance process sends a check balance request to the corresponding CPU, then waits for the response (success or fail) or a timeout signal (through shared action). If the CheckBalance process is successful, the current balance will be shown 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 processes will go to either CPUAuthen, CPUWithdraw or CPUCheckBalance, depending on the request from the ATM.

- The CPUAuthen is the process in CPU to perform the authenticate action. CPU process will go to CPUAuthen if it receives 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 CPUWithdraw if it receives the request from the ATM through

the shared action sendAmountSuccess. In its turn, the CPUWithdraw will send a withdraw 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 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 action. CPU will go to CPUCheckBalance if it receives 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 receiving requests from the ATM through shared actions, the Database also receives requests from the CPU before going to one of the three processes: DBAuthen, DBWithdraw or DBCheckBalance.

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

The unreliable network

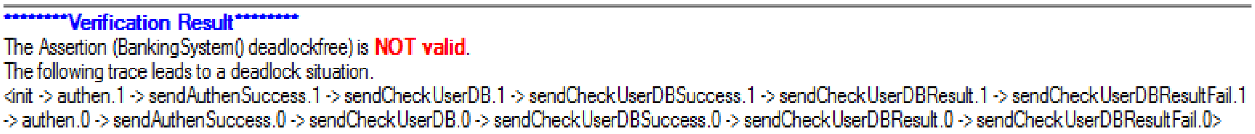
The unreliable network is modeled by giving an option of fail in every communication link. Each time a communication link fails, the sender will receive a send-fail signal and the device that is listening to that information will receive a timeout signal. Then the sender will try to resend or go to its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

The deadlock version

In this version, the deadlock can be introduced if the system does not handle the case that the communication protocol has errors. For instance, when the cloud processing unit connects to the database for authentication, if the message sent to the database is lost and the cloud processing unit doesn't send again but waits 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 screenshot to show that the assertion that the system is deadlockfree is

INVALID:



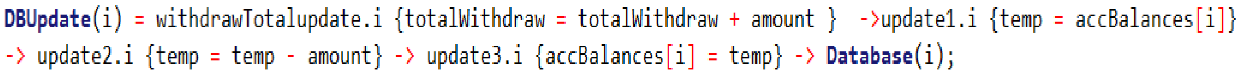
```
*********Verification Result*********
The Assertion (BankingSystem() deadlockfree) is NOT valid.
The following trace leads to a deadlock situation.
<init -> authen.1 -> sendAuthenSuccess.1 -> sendCheckUserDB.1 -> sendCheckUserDBSuccess.1 -> sendCheckUserDBResult.1 -> sendCheckUserDBResultFail.1
->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

The incorrect calculation version

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

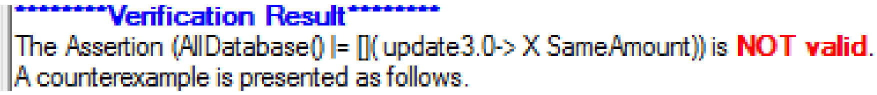
The problem can be eliminated by removing the use of *temp*. The account balance will be changed directly.

Here is the screenshot of the line that gives an 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 prove through invalid that the database does not 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 all of them check if the system is working correctly:

1) `#assert BankingSystem() deadlockfree;`
This means that the system does not reach deadlock. For the correct version, it will be VALID, and INVALID in the version that introduces deadlock.

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

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

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 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, if there is an error in communication, it may cause the system to continuously send authentication messages and resend and not produce a right result.

7) `#assert BankingSystem()|= [](!sendCheckBalance.0 -> <> returnBalance.0);`
The assertion should return INVALID. If the user does not send a command to check balance, the system will not return the balance to check.

8) This assertion tests 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 does not 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 event to withdraw money 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 failing to authenticate.

12) This assertion checks that if a user withdraws 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 holds the total money that the two users withdraw. We define:

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

Initially, each account has 100, N*100 = 200, (N = 2), so the sum of the two accounts and the total withdraw money has to be 200.
We will write an assertion that after the database updates an account (here we use the first account 'account 0'), after the event *update3.0*, which indicates the database finished 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

Overview

The Java Simulator is an implementation of the models developed in CSP/PAT, using a message-passing paradigm reflecting the events specified in the models. All communication between the processes occur through the passing of messages between them with the use of Java's *BlockingQueue* structures. Unreliable network conditions are simulated by using a random number generator to determine if a message is sent successfully. Timeouts are detected by using the *BlockingQueue*'s *take* method, which takes in an argument specifying a timeout period. The messages used can be found in the *TransactionMessage* class. They are almost in a one-to-one mapping with the events in the CSP models, with some name changes.

The simulator takes in a series of instructions and sets up a system comprising a given number of ATM, CloudProcessor and Database threads. In this implementation, only one Database thread is used. The ATMs then perform their scheduled actions (checking of balance and withdrawing of money). Each ATM action is preceded by an authentication call. Each element of this system corresponds to the major processes found in the CSP models. The sub-processes found in the CSP models are implemented as methods in each of the classes above.

Running the simulator

The simulator is a single Java program which takes in a file containing a series of instructions to perform. If no filename is given, the instructions will be taken from standard input. The simulator can be invoked via:

```
    java Simulator <instruction_file>
```

The simulator is capable of running in three different modes:

- proper, which is the correct version;
- deadlock, which may become deadlocked; and
- incorrect, which may perform incorrect calculations.

The commands that the simulator accepts specify the initial conditions of the system, and also a set of actions each ATM will perform once the simulator is started. The following commands are available:

Command	Description
addrecord <balance>	Adds a record and set its current balance to `balance` cents.
checkbalance <atmID> <recordID>	Add a checkbalance action to the ATM `atmID` on the record `recordID`
withdraw <atmID> <recordID> <amount>	Add a withdraw action to the ATM `atmID` on record `recordID` and withdraw `amount` cents.
numatm <num_of_atm>	Create `num_of_atm` ATMs and CloudProcessors.
type [proper|deadlock|incorrect]	Choose the type of simulator to run: proper; with deadlock; or with incorrect calculations.
run	Run the system.
stop	Stop the system.
exit	Quit the simulator.
help	Display this message.

A typical run of the simulator can be found in one of the demo files provided. At the start of a simulation run (using the *run* command), the simulator will display the initial balances. Upon termination, the simulator will display the final record balances.

Demo files

A set of demo files are packaged together with the simulator:

- *proper-demo1.txt*: contains a proper run of the system, including an authentication that will fail due to accessing a non-existent record. A correct run without timeouts will result in all records having 0 balance.
- *proper-demo2.txt*: contains a proper run of the system, with many ATMs performing withdrawals on the same record. A correct run without timeouts will result in Record 0 having 0 balance.
- *deadlock-demo.txt*: is just like proper-demo1.txt, but in deadlock mode. The run will encounter a deadlock due to an ATM trying to access a non-existent record.
- *incorrect-demo.txt*: is just like proper-demo2.txt, but in incorrect mode. The run may produce an incorrect balance in Record 0 depending on the way the threads interleave.

Note that timeouts may cause some inconsistency between runs.