INFORMATICS
INSTITUTE OF
TECHNOLOGY

Informatics Institute of Technology

in Collaboration with

University of Westminster

BSc. (Hons) in Computer Science

Concurrent Programming 2019/2020

Coursework Report for

By

H.K Dulana Hansisi-W1654550-2016342

Contents

1. FSP FORMS.

1.1 BANKACCOUNT

6SENG002W Concurrent Programming

FSP Process Analysis & Design Form

Name	H.K Dulana Hansisi
Student ID	UOW ID: W1654550 | IIT ID: 2016342
Date	1/11/2019

1. FSP Process Attributes

Attribute	Value
Name	BANKACCOUNT
Description	Actions perform by BANKACCOUNT.
Alphabet	{calculateAccBalance, readAccBalance[5], updateAccBalance}
Number of States	2
Deadlocks (yes/no)	N/A
Deadlock Trace(s)	N/A

2. FSP Process Code

FSP Process:

```
range BALANCE = 5..5
range TRANSACTION = 1..1
range ALLTRANSACTION = 1..1
range BALANCENEW = 3..7

BANKACCOUNT = EXISTINGAMOUNT,
EXISTINGAMOUNT = (readAccBalance[balance:BALANCE]->EXISTINGAMOUNT
|calculateAccBalance->NEWBALANCE),
NEWBALANCE = (updateAccBalance->EXISTINGAMOUNT).
```

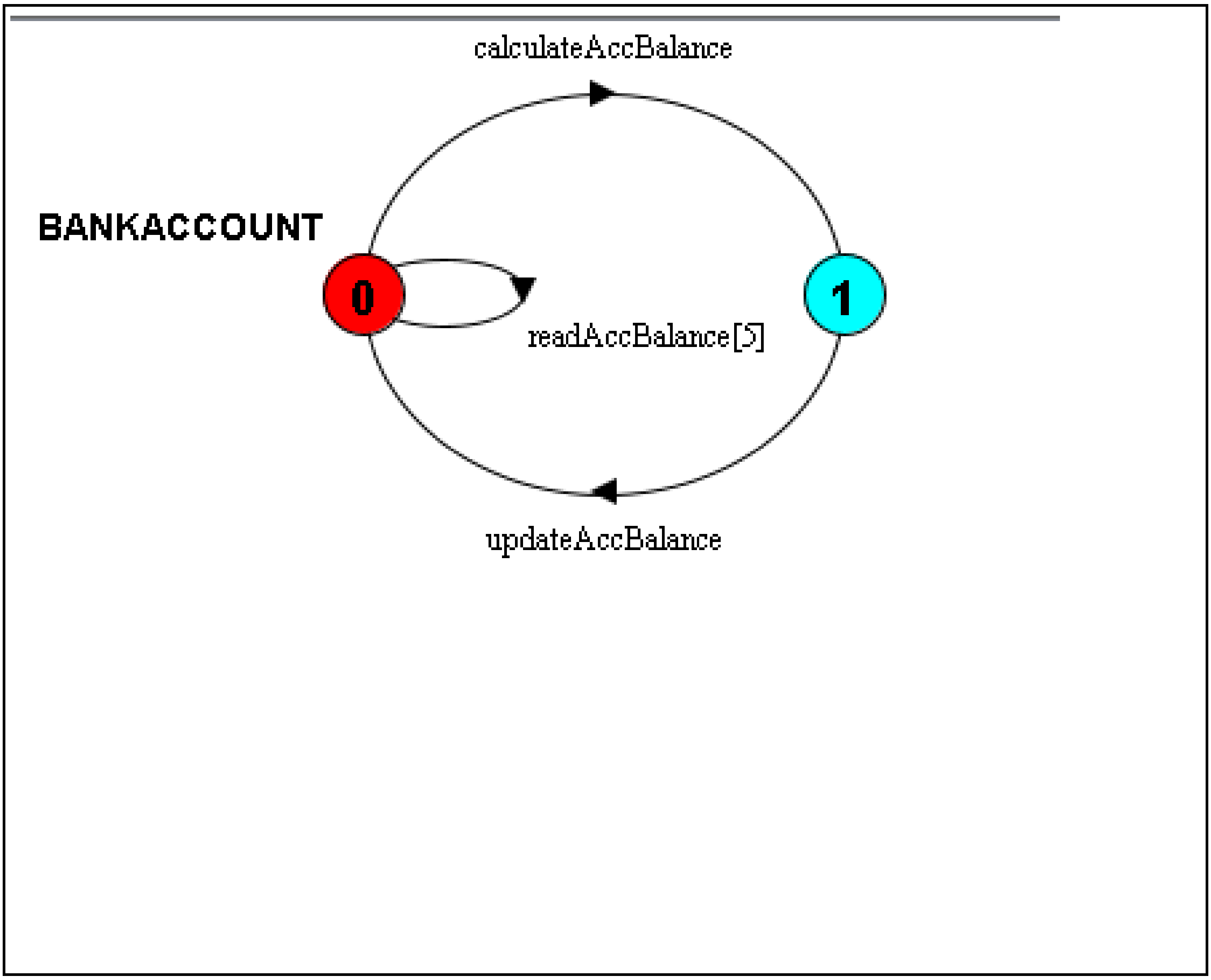
3. Actions Description

A description of what each of the FSP process' actions represents, i.e. is modelling. In addition, indicate if the action is intended to be synchronised (shared) with another process or asynchronous (not shared). (Add rows as necessary.)

Actions	Represents	Synchronous or Asynchronous
calculateAccBalance	Changers EXISTINGMONEY to NEWBALANCE state.	Asynchronous
readAccBalance[balance:BALANCE]	EXISTINGMONEY state comes back to EXISTINGMONEY state	Synchronous
updateAccBalance	NEWBALANCE state goes to EXISTINGMONEY state.	Asynchronous

4. FSM/LTS Diagrams of FSP Process

Note that if there are too many states, more than 64, then the LTSA tool will not be able to draw the diagram. In this case draw small diagrams of the most important parts of the complete diagram.



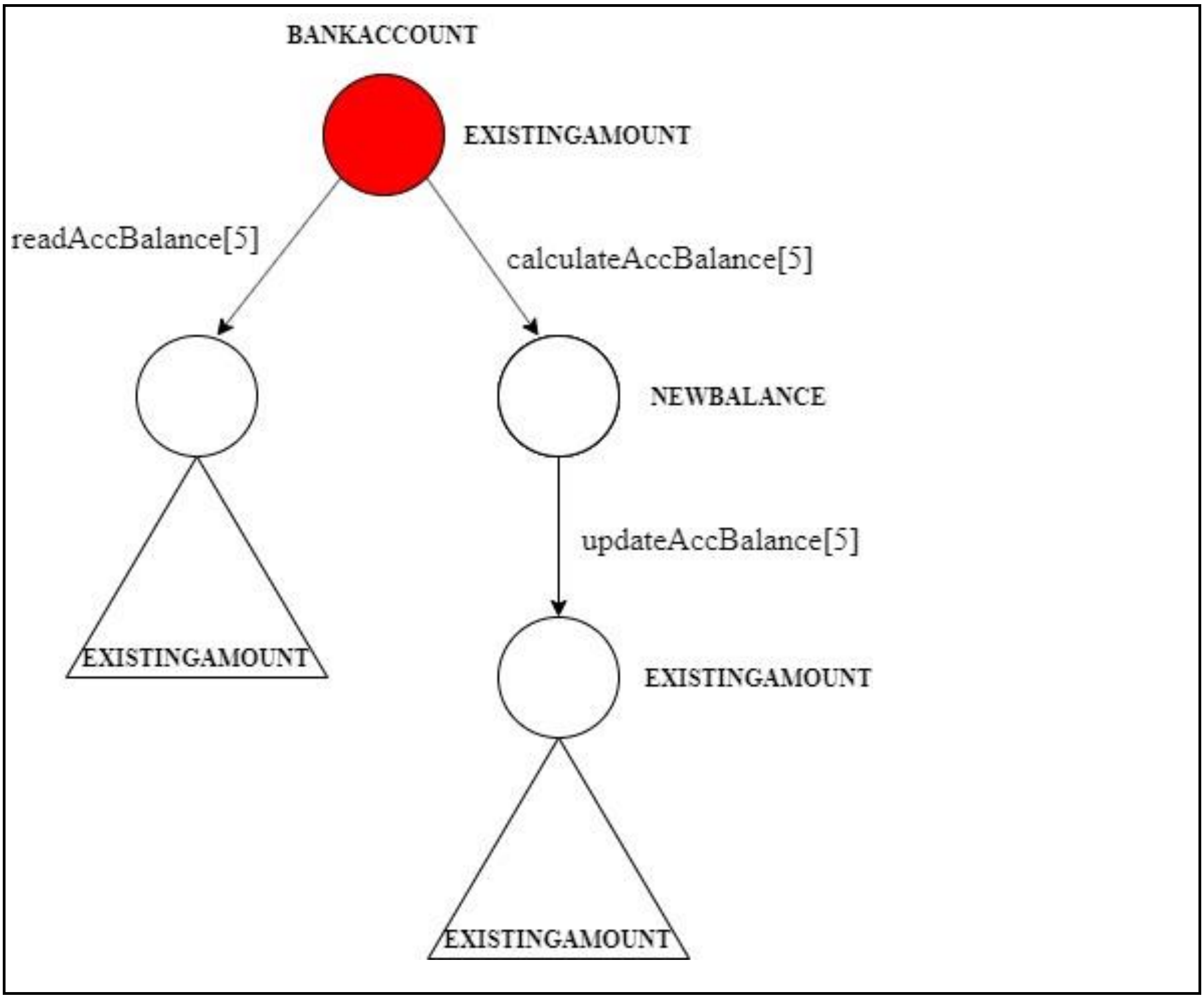
5. LTS States

A description of what each of the FSP process' states represents, i.e. is modelling. If there are a large number of states, then you can group similar states together &/or only include the most important ones. For example, identify any states related to mutual exclusion (ME) & the associated critical section (CS), e.g. waiting to enter the CS state, in the CS state(s), left the CS state. (Add rows as necessary.)

States	Represents
Q0	Initial state. After invoking readAccBalance[5]. After invoking updateAccBalance
Q1	After invoking calculateAccBalance.

6. Trace Tree for FSP Process

The trace tree for the process. Use the conventions given in the lecture notes.



1.2 GRANDMOTHER

6SENG002W Concurrent Programming

FSP Process Analysis & Design Form

Name	H.K Dulana Hansisi
Student ID	UOW ID: W1654550 | IIT ID: 2016342
Date	1/11/2019

1. FSP Process Attributes

Attribute	Value
Name	GRANDMOTHER
Description	Actions which can be performed by grandmother and its states.
Alphabet	{addingBirthdayMoney[5][1], calculateBalance[6], depositMoney[5][1], readBalance[5], sendBDayCard, updateAccount[3..7]}
Number of States	4
Deadlocks (yes/no)	N/A
Deadlock Trace(s)	N/A

2. FSP Process Code

FSP Process:

```
range BALANCE = 5..5
range TRANSACTION = 1..1
range ALLTRANSACTION = 1..1
range BALANCENEW = 3..7
range DEPOSIT = 1..1

GRANDMOTHER = EXISTINGAMOUNT,
EXISTINGAMOUNT = (readBalance[balance:BALANCE]
->EXISTINGAMOUNT|addingBirthdayMoney[balance:BALANCE][amount:DEPO
SIT]->calculateBalance[balance+amount] ->NEWBALANCE[balance+amount]
|depositMoney[balance:BALANCE][amount:DEPOSIT]->calculateBalance[balance+
amount]->NEWBALANCE[balance+amount]),
NEWBALANCE[balance:BALANCENEW] =
(updateAccount[balance]->EXISTINGAMOUNT)+{sendBDayCard}.
```

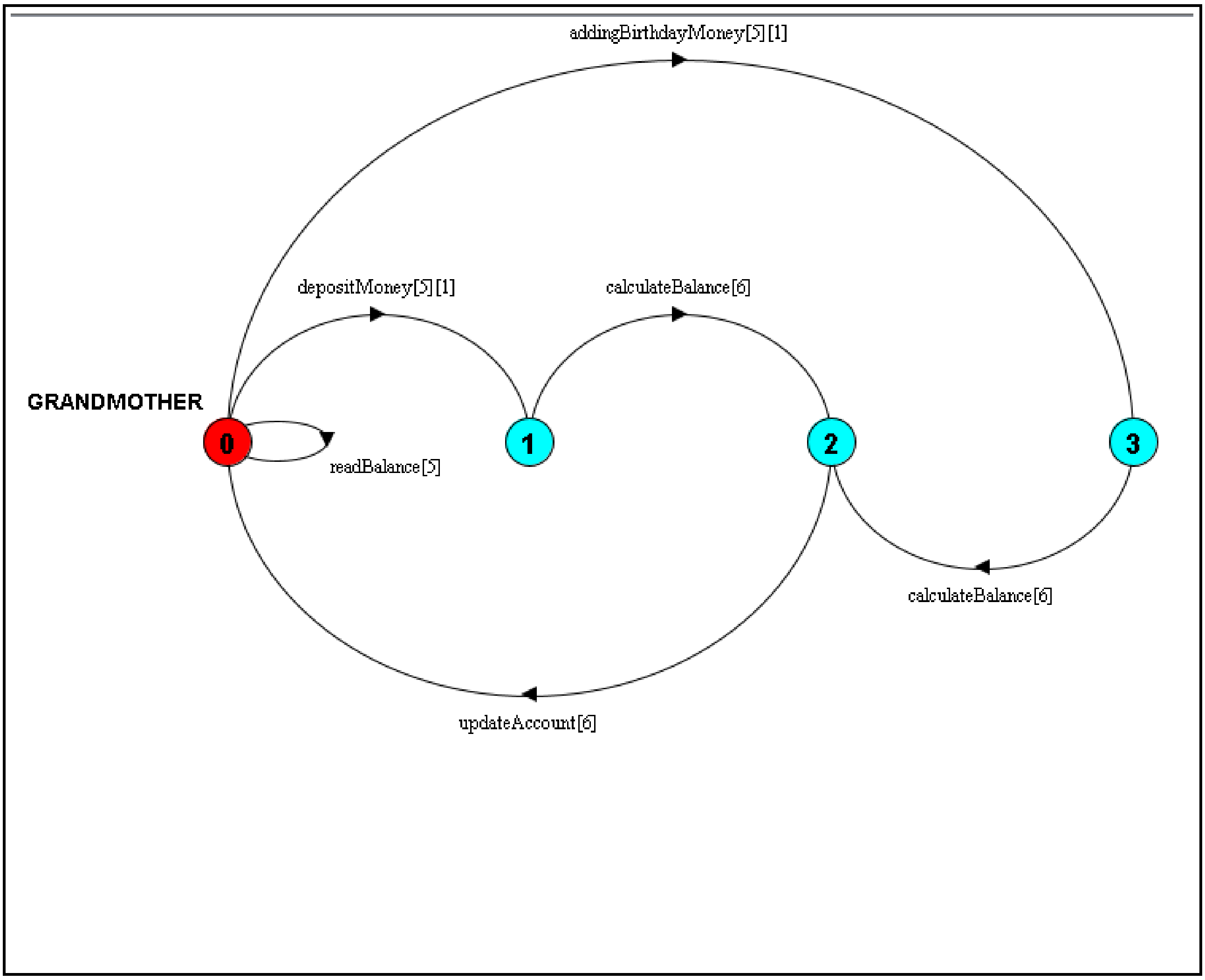
3. Actions Description

A description of what each of the FSP process' actions represents, i.e. is modelling. In addition, indicate if the action is intended to be synchronised (shared) with another process or asynchronous (not shared). (Add rows as necessary.)

Actions	Represents	Synchronous or Asynchronous
addingBirthdayMoney[balance:BALANCE][amount:DEPOSIT]	Initial state changes into state 4.	Asynchronous
calculateBalance[balance+amount]	State 3 changes into NEWBALANCE state. State 1 changes into NEWBALANCE.	Asynchronous
depositMoney[balance:BALANCE][amount:DEPOSIT]	Initial state changes into state 2	Synchronous
readBalance[balance:BALANCE]	EXISTINGMONEY state comes back to EXISTINGMONEY state	Synchronous
sendBDayCard	Extends alphabet	Asynchronous
updateAccount[balance]	NEWBALANCE state goes to EXISTINGMONEY state.	Asynchronous

4. FSM/LTS Diagrams of FSP Process

Note that if there are too many states, more than 64, then the LTSA tool will not be able to draw the diagram. In this case draw small diagrams of the most important parts of the complete diagram.



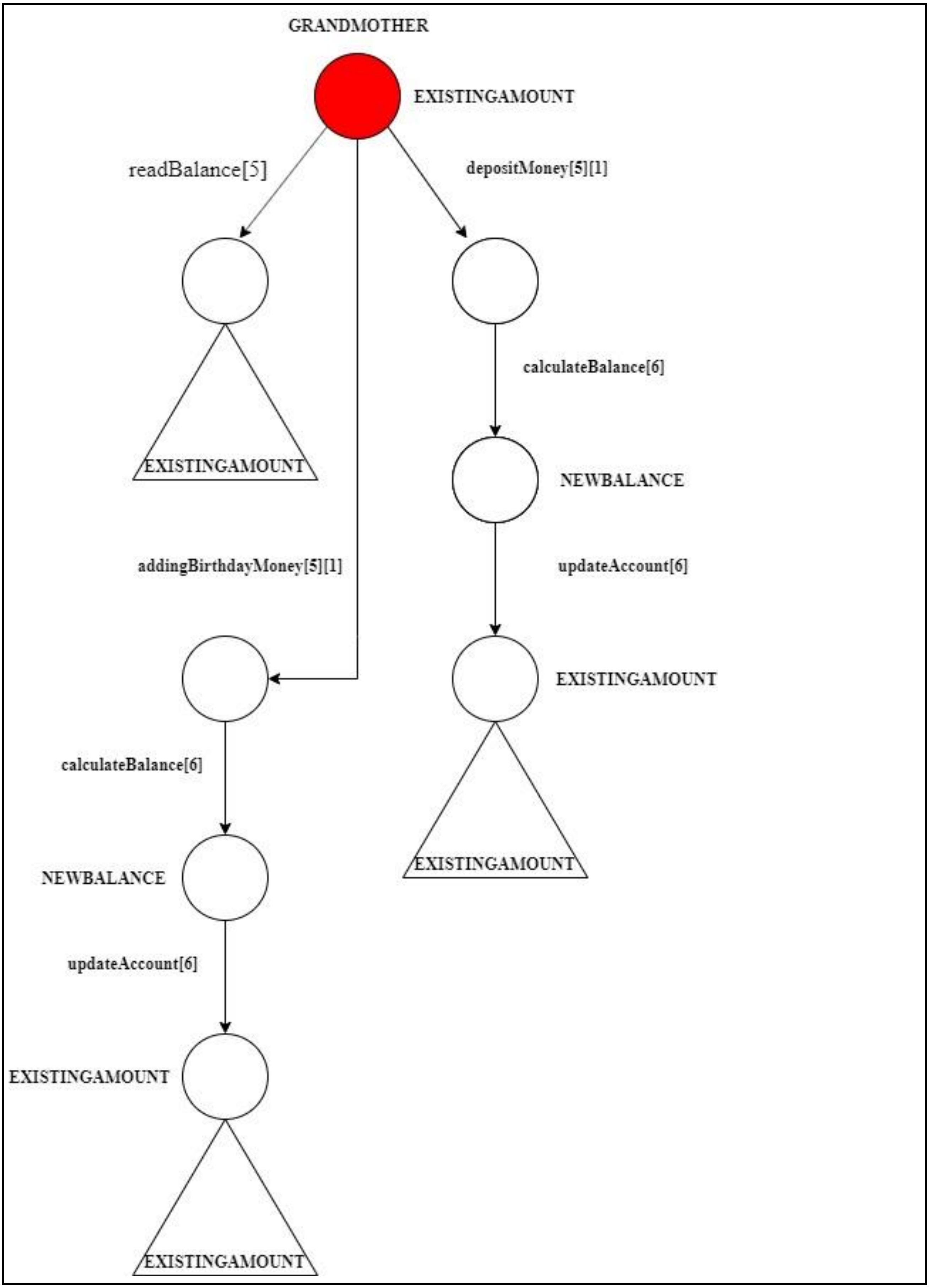
5. LTS States

A description of what each of the FSP process' states represents, i.e. is modelling. If there are a large number of states, then you can group similar states together &/or only include the most important ones. For example, identify any states related to mutual exclusion (ME) & the associated critical section (CS), e.g. waiting to enter the CS state, in the CS state(s), left the CS state. (Add rows as necessary.)

States	Represents
Q0	Initial state. After invoking readBalance [5]. After invoking updateAccount [6]
Q1	After invoking depositMoney[5][1].
Q2	After invoking calculateBalance[6] on Q3. After invoking calculateBalance[6] on Q1.
Q3	After invoking addingBirthdayMoney[5][1].
Q4	After invoking addingBirthdayMoney.

6. Trace Tree for FSP Process

The trace tree for the process. Use the conventions given in the lecture notes.


GRANDMOTHER
EXISTINGAMOUNT
readBalance[5]
depositMoney[5][1]
EXISTINGAMOUNT
calculateBalance[6]
NEWBALANCE
addingBirthdayMoney[5][1]
updateAccount[6]
EXISTINGAMOUNT
EXISTINGAMOUNT
calculateBalance[6]
NEWBALANCE
updateAccount[6]
EXISTINGAMOUNT
EXISTINGAMOUNT

1.3 LOANCOMPANY

6SENG002W Concurrent Programming

FSP Process Analysis & Design Form

Name	H.K Dulana Hansisi
Student ID	UOW ID: W1654550 | IIT ID: 2016342
Date	1/11/2019

1. FSP Process Attributes

Attribute	Value
Name	LOANCOMPANY
Description	Actions performs by LOANCOMPANY
Alphabet	{calculateBalance[6], depositMoney[5][1], readBalance[5], updateAccount[3..7]}
Number of States	3
Deadlocks (yes/no)	N/A
Deadlock Trace(s)	N/A

2. FSP Process Code

FSP Process:

```
range BALANCE = 5..5
range TRANSACTION = 1..1
range ALLTRANSACTION = 1..1
range BALANCENEW = 3..7
range DEPOSIT = 1..1

LOANCOMPANY = EXISTINGAMOUNT,
EXISTINGAMOUNT = (readBalance[balance:BALANCE]->EXISTINGAMOUNT |
depositMoney[balance:BALANCE][amount:DEPOSIT]->calculateBalance[balance+amount]->NEWBALANCE[balance+amount]),
NEWBALANCE[balance:BALANCENEW] = (updateAccount[balance]->EXISTINGAMOUNT).
```

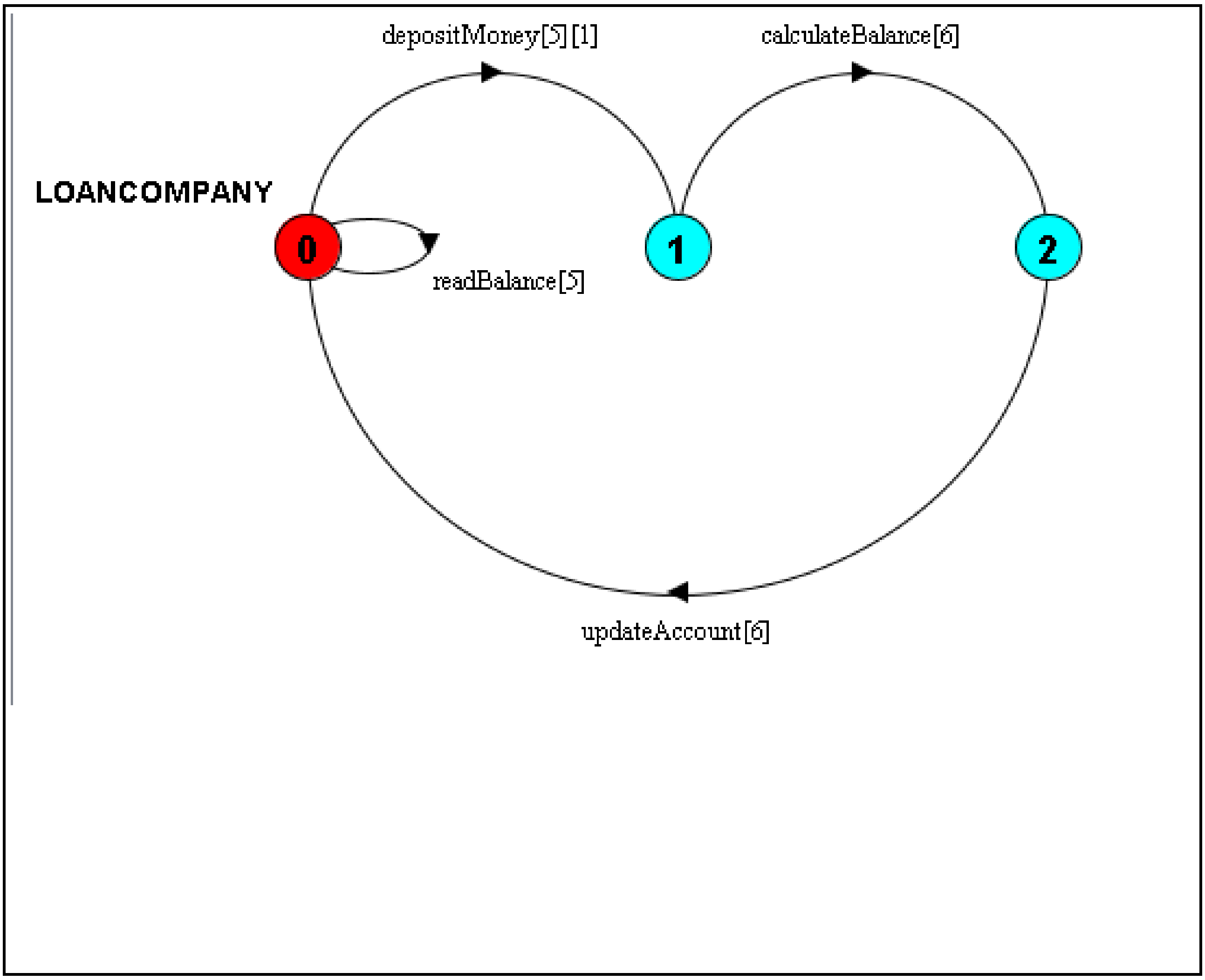
3. Actions Description

A description of what each of the FSP process' actions represents, i.e. is modelling. In addition, indicate if the action is intended to be synchronised (shared) with another process or asynchronous (not shared). (Add rows as necessary.)

Actions	Represents	Synchronous or Asynchronous
calculateBalance[balance+amount]	State 1 changes into NEWBALANCE state.	Asynchronous
depositMoney[balance:BALANCE][amount:DEPOSIT]	Initial state changes into state 1	Synchronous
readBalance[balance:BALANCE]	EXISTINGMONEY state comes back to EXISTINGMONEY state	Synchronous
updateAccount[balance]	NEWBALANCE state goes to EXISTINGMONEY state.	Asynchronous

4. FSM/LTS Diagrams of FSP Process

Note that if there are too many states, more than 64, then the LTSA tool will not be able to draw the diagram. In this case draw small diagrams of the most important parts of the complete diagram.



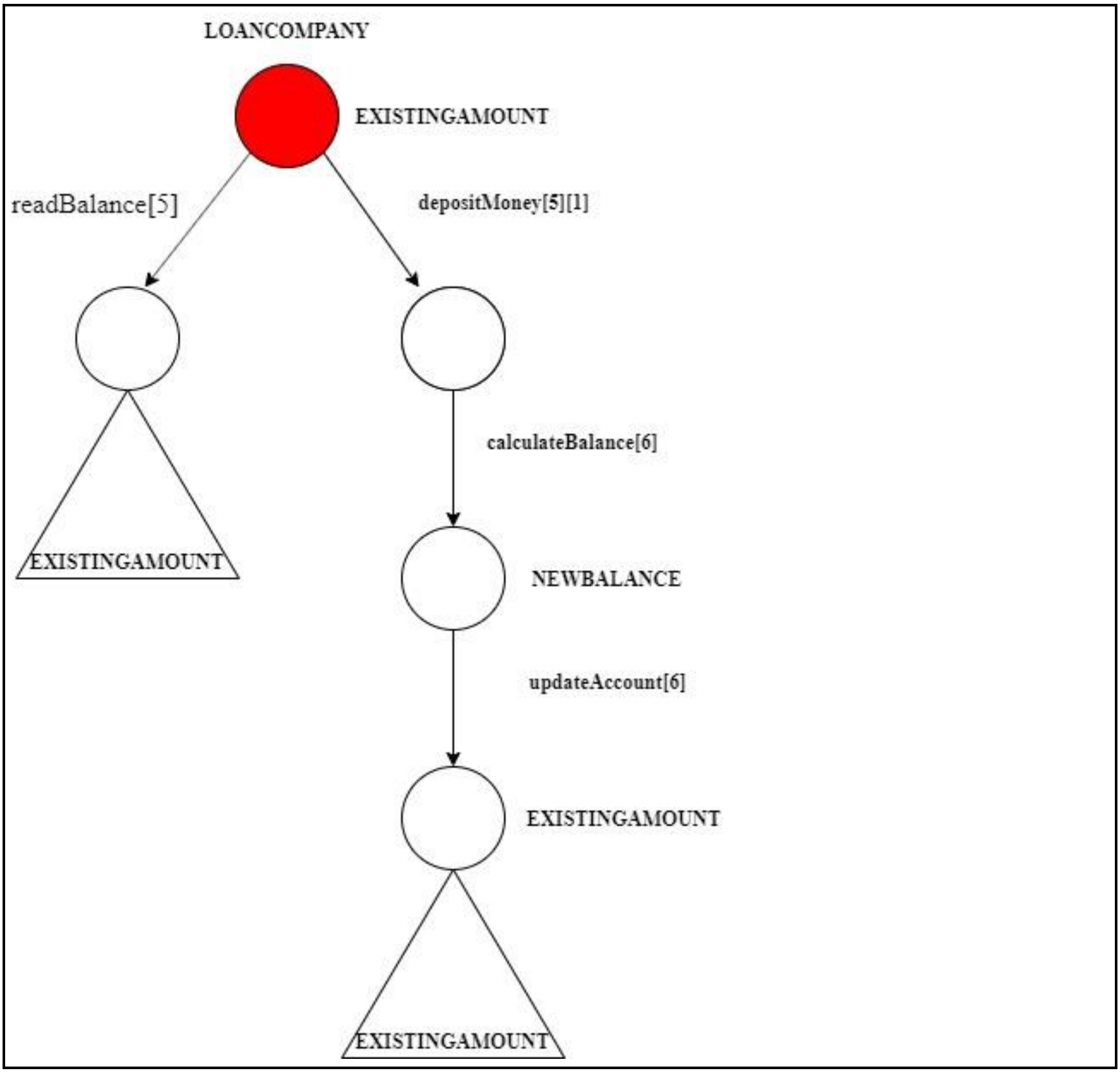
5. LTS States

A description of what each of the FSP process' states represents, i.e. is modelling. If there are a large number of states, then you can group similar states together &/or only include the most important ones. For example, identify any states related to mutual exclusion (ME) & the associated critical section (CS), e.g. waiting to enter the CS state, in the CS state(s), left the CS state. (Add rows as necessary.)

States	Represents
Q0	Initial state. After invoking readBalance[5]. After invoking updateAccount[6]
Q1	After invoking depositMoney[5][1].
Q2	After invoking calculateBalance[6].

6. Trace Tree for FSP Process

The trace tree for the process. Use the conventions given in the lecture notes.



1.4 STUDENT

6SENG002W Concurrent Programming

FSP Process Analysis & Design Form

Name	H.K Dulana Hansisi
Student ID	UOW ID: W1654550 | IIT ID: 2016342
Date	1/11/2019

1. FSP Process Attributes

Attribute	Value
Name	STUDENT
Description	Actions which can be performed by a Student and its states.
Alphabet	{calculateBalance[4], readBalance[5], subtractMoney[1], updateAccount[3..7], withdrawMoney[5][1]}
Number of States	7
Deadlocks (yes/no)	No deadlocks/errors
Deadlock Trace(s)	N/A

2. FSP Process Code

FSP Process:

```
range BALANCE = 5..5
range TRANSACTION = 1..1
range ALLTRANSACTION = 1..1
range BALANCENEW = 3..7
range DEPOSIT = 1..1

STUDENT = EXISTINGAMOUNT,
EXISTINGAMOUNT = (readBalance[balance:BALANCE]->EXISTINGAMOUNT
|
withdrawMoney[balance:BALANCE][transaction:TRANSACTION]->subtractMoney[transaction]-
>calculateBalance[balance-transaction]->NEWBALANCE[balance-
transaction]|withdrawMoney[balance:BALANCE][transaction:TRANSACTION]
->buyNewPhone->subtractMoney[transaction]->calculateBalance[balance-transaction]
->NEWBALANCE[balance-transaction]),
NEWBALANCE[balance:BALANCENEW] = (updateAccount[balance]->EXISTINGAMOUNT)
\{buyNewPhone}.
```

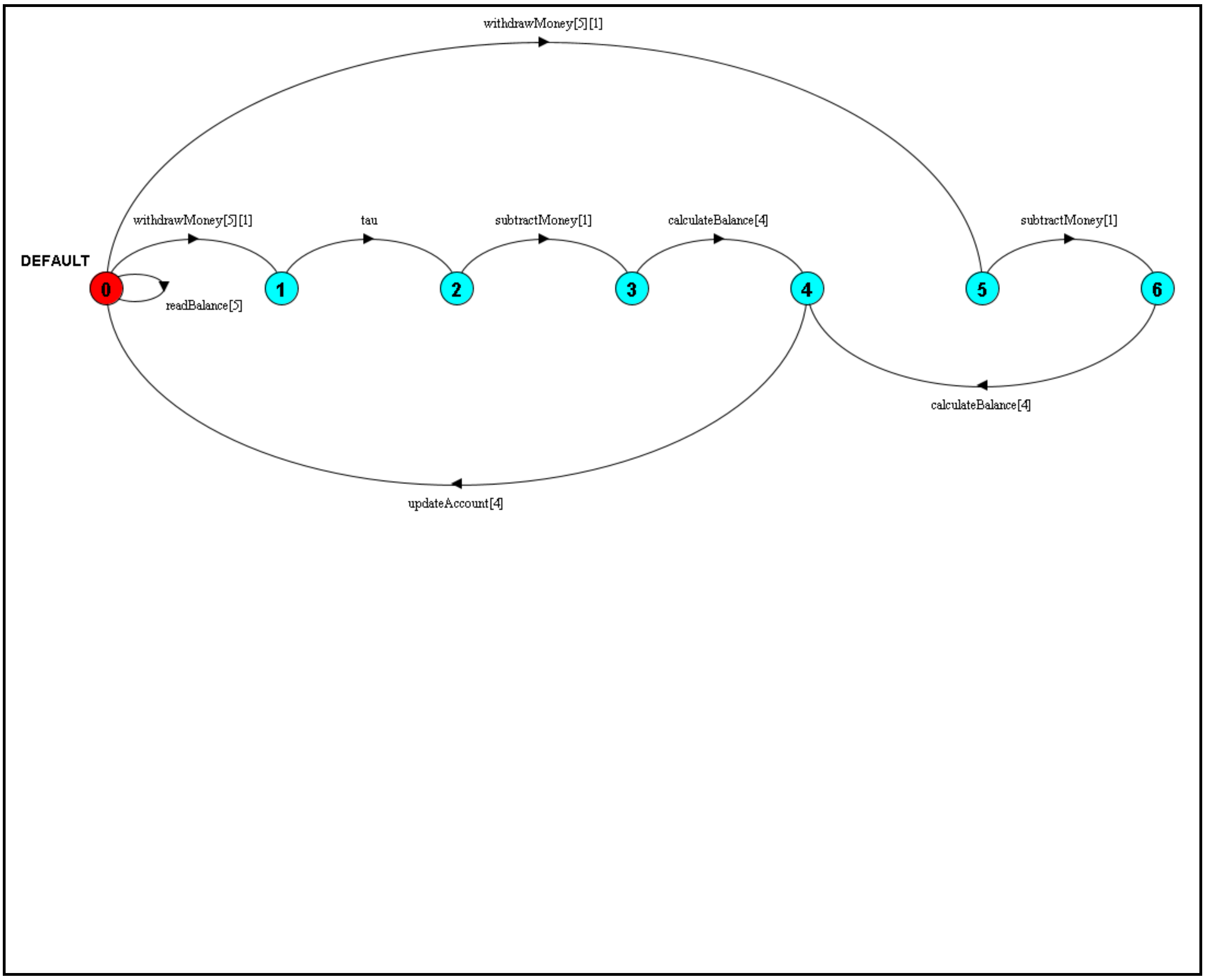
3. Actions Description

A description of what each of the FSP process' actions represents, i.e. is modelling. In addition, indicate if the action is intended to be synchronised (shared) with another process or asynchronous (not shared). (Add rows as necessary.)

Actions	Represents	Synchronous or Asynchronous
readBalance[balance:BALANCE]	EXISTINGAMOUNT state will be met again because of the recursion.	Synchronous
withdrawMoney[balance:BALANCE][transaction:TRANSACTION]	EXISTINGAMOUNT state changes into State 1. EXISTINGAMOUNT state changes into State 5.	Synchronous
subtractMoney[transaction]	State 5 changes into state 6. State 2 changes into state 3.	Asynchronous
calculateBalance[balance-transaction]	State 3 changes into NEWBALANCE state. State 6 changes into NEWBALANCE state.	Asynchronous
updateAccount[balance]	NEWBALANCE state changes into EXISTINGAMOUNT	Asynchronous
buyNewPhone	State 1 changes into state 2	Asynchronous

4. FSM/LTS Diagrams of FSP Process

Note that if there are too many states, more than 64, then the LTSA tool will not be able to draw the diagram. In this case draw small diagrams of the most important parts of the complete diagram.



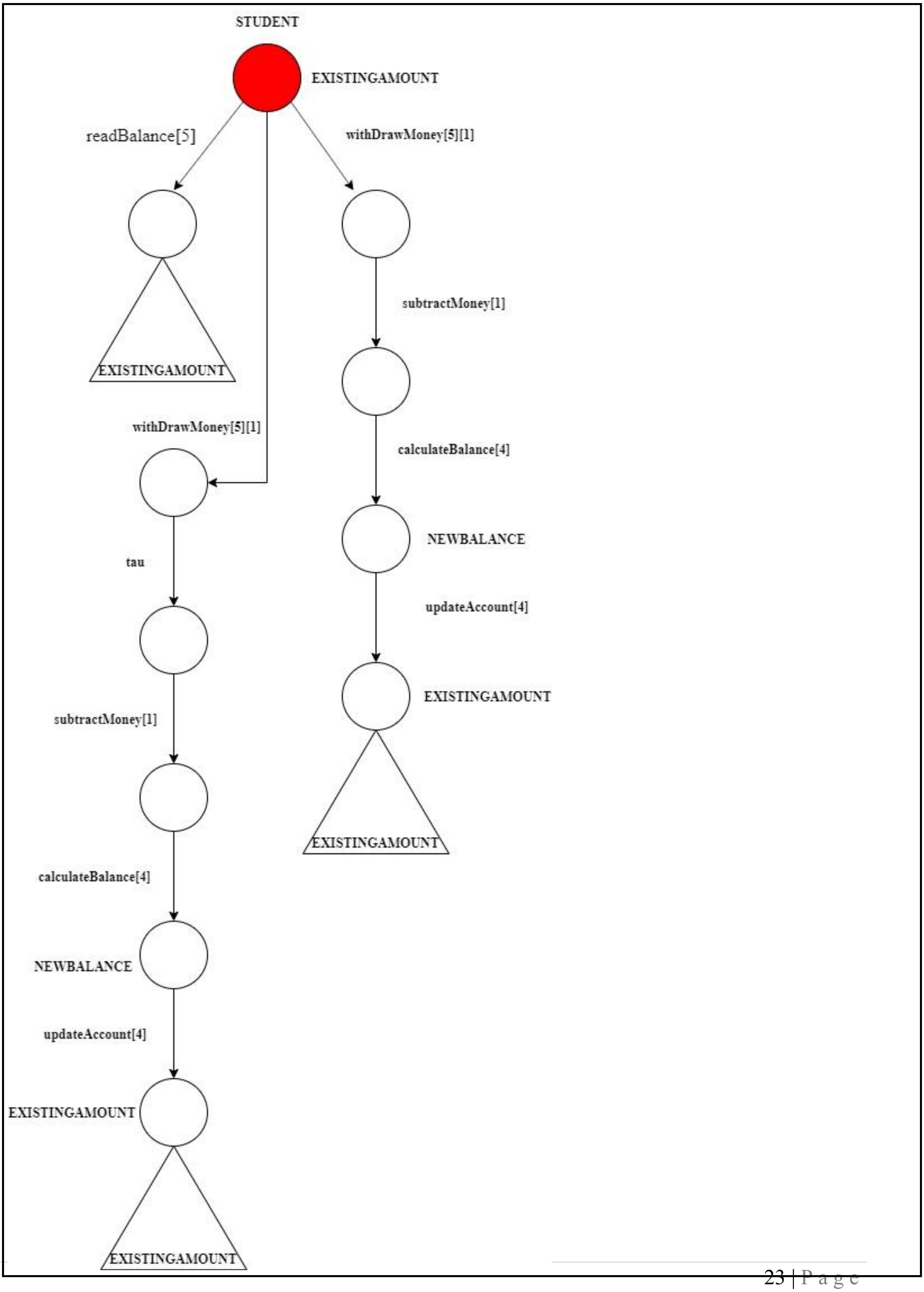
5. LTS States

A description of what each of the FSP process' states represents, i.e. is modelling. If there are a large number of states then you can group similar states together &/or only include the most important ones. For example, identify any states related to mutual exclusion (ME) & the associated critical section (CS), e.g. waiting to enter the CS state, in the CS state(s), left the CS state. (Add rows as necessary.)

States	Represents
Q0	Initial state. After invoking readBalance[5]. After invoking updateAccount[4] on Q4.
Q1	After invoking withdrawMoney[5][1] on Q0.
Q2	After invoking tau.
Q3	After invoking subtractMoney[1].
Q4	After invoking calculateBalance[4] on Q3. After invoking calculateBalance[4] on Q6
Q5	After invoking withdrawMoney[5][1] on Q0.
Q6	After invoking subtractMoney[1] on Q5.

6. Trace Tree for FSP Process

The trace tree for the process. Use the conventions given in the lecture notes.


STUDENT
EXISTINGAMOUNT
readBalance[5]
withDrawMoney[5][1]
EXISTINGAMOUNT
subtractMoney[1]
withDrawMoney[5][1]
calculateBalance[4]
NEWBALANCE
tau
updateAccount[4]
EXISTINGAMOUNT
subtractMoney[1]
EXISTINGAMOUNT
calculateBalance[4]
NEWBALANCE
updateAccount[4]
EXISTINGAMOUNT
EXISTINGAMOUNT

1.5 UNIVERSITY

6SENG002W Concurrent Programming

FSP Process Analysis & Design Form

Name	H.K Dulana Hansisi
Student ID	UOW ID: W1654550 | IIT ID: 2016342
Date	1/11/2019

1. FSP Process Attributes

Attribute	Value
Name	UNIVERSITY
Description	Actions perform by UNIVERSITY.
Alphabet	{calculateBalance[4], giveADiscount, readBalance[5], subtractMoney[1], updateAccount[3..7], withdrawMoney[5][1]}
Number of States	4
Deadlocks (yes/no)	N/A
Deadlock Trace(s)	N/A

2. FSP Process Code

FSP Process:

```
range BALANCE = 5..5
range TRANSACTION = 1..1
range ALLTRANSACTION = 1..1
range BALANCENEW = 3..7
range DEPOSIT = 1..1

UNIVERSITY = EXISTINGAMOUNT,
EXISTINGAMOUNT = (readBalance[balance:BALANCE] ->EXISTINGAMOUNT|
withdrawMoney[balance:BALANCE][transaction:TRANSACTION]
->subtractMoney[transaction]->calculateBalance[balance-transaction]->
NEWBALANCE[balance-transaction]),
NEWBALANCE[balance:BALANCENEW] = (updateAccount [balance] ->
EXISTINGAMOUNT)+{giveADiscount}.
```

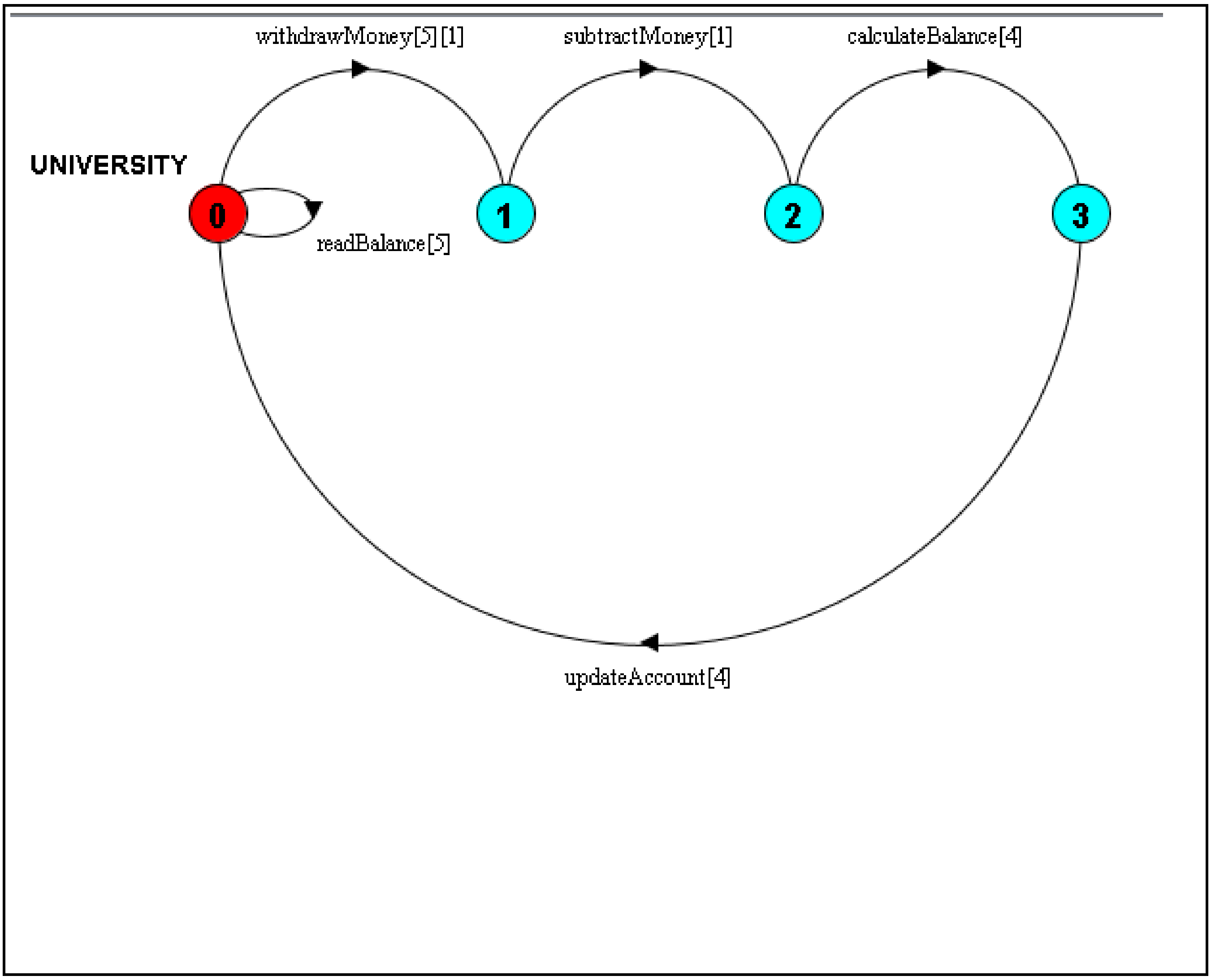
3. Actions Description

A description of what each of the FSP process' actions represents, i.e. is modelling. In addition, indicate if the action is intended to be synchronised (shared) with another process or asynchronous (not shared). (Add rows as necessary.)

Actions	Represents	Synchronous or Asynchronous
calculateBalance[balance-transaction]	State 2 changes into NEWBALANCE state.	Asynchronous
readBalance[balance:BALANCE]	EXISTINGMONEY state comes back to EXISTINGMONEY state	Synchronous
updateAccount [balance]	NEWBALANCE state goes to EXISTINGMONEY state.	Asynchronous
subtractMoney[transaction]	Changes state 1 into state 2.	Asynchronous
withdrawMoney[balance:BALANCE][transaction:TRANSACTION]	Changes initial state into state 1.	Synchronous

4. FSM/LTS Diagrams of FSP Process

Note that if there are too many states, more than 64, then the LTSA tool will not be able to draw the diagram. In this case draw small diagrams of the most important parts of the complete diagram.



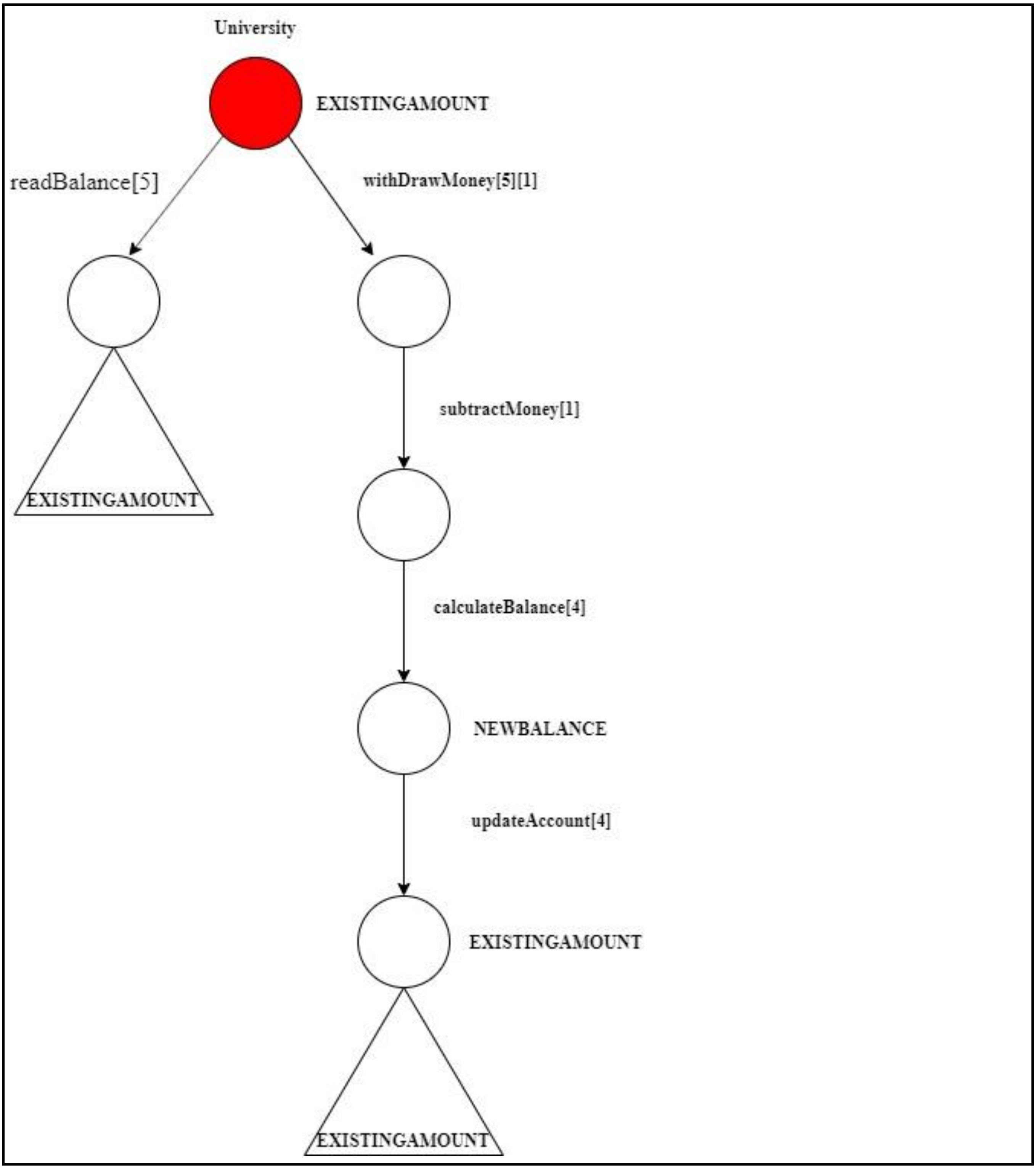
5. LTS States

A description of what each of the FSP process' states represents, i.e. is modelling. If there are a large number of states, then you can group similar states together &/or only include the most important ones. For example, identify any states related to mutual exclusion (ME) & the associated critical section (CS), e.g. waiting to enter the CS state, in the CS state(s), left the CS state. (Add rows as necessary.)

States	Represents
Q0	Initial state. After invoking readBalance[5]. After invoking updateAccount[4].
Q1	After invoking withdrawMoney[5][1]
Q2	After invoking subtractMoney[1]
Q3	After invoking calculateBalance[4]

6. Trace Tree for FSP Process

The trace tree for the process. Use the conventions given in the lecture notes.



1.6 BANKINGSYSTEM

6SENG002W Concurrent Programming

FSP Process Composition Analysis & Design Form

Name	H.K Dulana Hansisi
Student ID	IID ID: 2016342 UOW ID: W1654550
Date	16/12/2019

1. FSP Composition Process Attributes

Attribute	Value
Name	BANKINGSYSTEM
Description	This represents a banking system which is used by 4 users.
Alphabet (Use LTSA's compressed notation, if alphabet is large.)	{a.{calculateAccBalance, readAccBalance[5], subtractMoney[1], updateAccBalance, withdrawMoney[5][1]}, b.{addingBirthdayMoney[5][1], calculateAccBalance, depositMoney[5][1], readAccBalance[5], {sendBDayCard, updateAccBalance}}, c.{calculateAccBalance, depositMoney[5][1], readAccBalance[5], updateAccBalance}, calculateAccBalance.{[4], [6]}, d.{{calculateAccBalance, giveADiscount}, readAccBalance[5], subtractMoney[1], updateAccBalance, withdrawMoney[5][1]}, readAccBalance[5], updateAccBalance[3..7]}
Sub-processes (List them.)	STUDENT, GRANDMOTHER, LOANCOMPANY, BANCACCOUNT, UNIVERSITY
Number of States	1344
Deadlocks (yes/no)	N/A
Deadlock Trace(s)	N/A

2. FSP "main" Program Code

The code for the parallel composition of all of the sub-processes and the definitions of any constants, ranges & process labelling sets used. (Do not include the code for the sub-processes.)

FSP Program:

```
range BALANCE = 5..5
range TRANSACTION = 1..1
range BALANCENEW = 3..7
range DEPOSIT = 1..1

UNIVERSITY = EXISTINGAMOUNT,
EXISTINGAMOUNT = (readBalance[balance:BALANCE] ->EXISTINGAMOUNT|
withdrawMoney[balance:BALANCE][transaction:TRANSACTION]
->subtractMoney[transaction]->calculateBalance[balance-transaction]->
NEWBALANCE[balance-transaction]),
NEWBALANCE[balance:BALANCENEW] = (updateAccount [balance] ->
EXISTINGAMOUNT)+{giveADiscount}.

STUDENT = EXISTINGAMOUNT,
EXISTINGAMOUNT = (readBalance[balance:BALANCE]->EXISTINGAMOUNT
|
withdrawMoney[balance:BALANCE][transaction:TRANSACTION]->subtractMoney[transaction]-
>calculateBalance[balance-transaction]->NEWBALANCE[balance-
transaction]|withdrawMoney[balance:BALANCE][transaction:TRANSACTION]
->buyNewPhone->subtractMoney[transaction]->calculateBalance[balance-transaction]
->NEWBALANCE[balance-transaction]),
NEWBALANCE[balance:BALANCENEW] =
(updateAccount[balance]->EXISTINGAMOUNT)\{buyNewPhone}.


LOANCOMPANY = EXISTINGAMOUNT,
EXISTINGAMOUNT = (readBalance[balance:BALANCE]->EXISTINGAMOUNT |
depositMoney[balance:BALANCE][amount:DEPOSIT]->calculateBalance[balance+amount]->NE
WBALANCE[balance+amount]),
NEWBALANCE[balance:BALANCENEW] = (updateAccount[balance]->EXISTINGAMOUNT).

GRANDMOTHER = EXISTINGAMOUNT,
EXISTINGAMOUNT = (readBalance[balance:BALANCE]
->EXISTINGAMOUNT|addingBirthdayMoney[balance:BALANCE][amount:DEPOSIT]->calculat
eBalance[balance+amount] ->NEWBALANCE[balance+amount]
|depositMoney[balance:BALANCE][amount:DEPOSIT]->calculateBalance[balance+amount]->NE
WBALANCE[balance+amount]),
NEWBALANCE[balance:BALANCENEW] =
(updateAccount[balance]->EXISTINGAMOUNT)+{sendBDayCard}.

BANKACCOUNT = EXISTINGAMOUNT,
EXISTINGAMOUNT = (readAccBalance[balance:BALANCE]->EXISTINGAMOUNT
|withdrawAccMoney->calculateAccBalance->NEWBALANCE|depositAccMoney->calculateAccB
alance->NEWBALANCE),
NEWBALANCE = (updateAccBalance->EXISTINGAMOUNT).


||BANKINGSYSTEM =
(a:STUDENT/{withdrawAccMoney/a.withdrawMoney,readAccBalance/a.readBalance}
|| b:GRANDMOTHER/{depositAccMoney/b.depositMoney,readAccBalance/b.readBalance}
|| c:LOANCOMPANY/{depositAccMoney/c.depositMoney,readAccBalance/c.readBalance}
|| d:UNIVERSITY/{withdrawAccMoney/d.withdrawMoney,readAccBalance/d.readBalance}
||{a,b,c,d}::BANKACCOUNT)@{read,write,deposit}.
```

3. Combined Sub-processes

(Add rows as necessary.)

Process	Description
a:STUDENT	Represents a student (owner of the bank account).
b:GRANDMOTHER	Represents grandmother of student.
c:LOANCOMPANY	Represents loan company which pays for student’s education
d:UNIVERSITY	Represents University where student study
{a,b,c,d}::BANKACCOUNT)	Represents a bank account which holds money.

4. Analysis of Combined Process Actions

- **Synchronous** actions are performed by at least two sub-process in the combination.
- **Blocked Synchronous** actions cannot be performed, since at least one of the sub-processes cannot preform them, because they were added to their alphabet using alphabet extension.
- **Asynchronous** actions are preformed independently by a single sub-process.

(Add rows as necessary.)

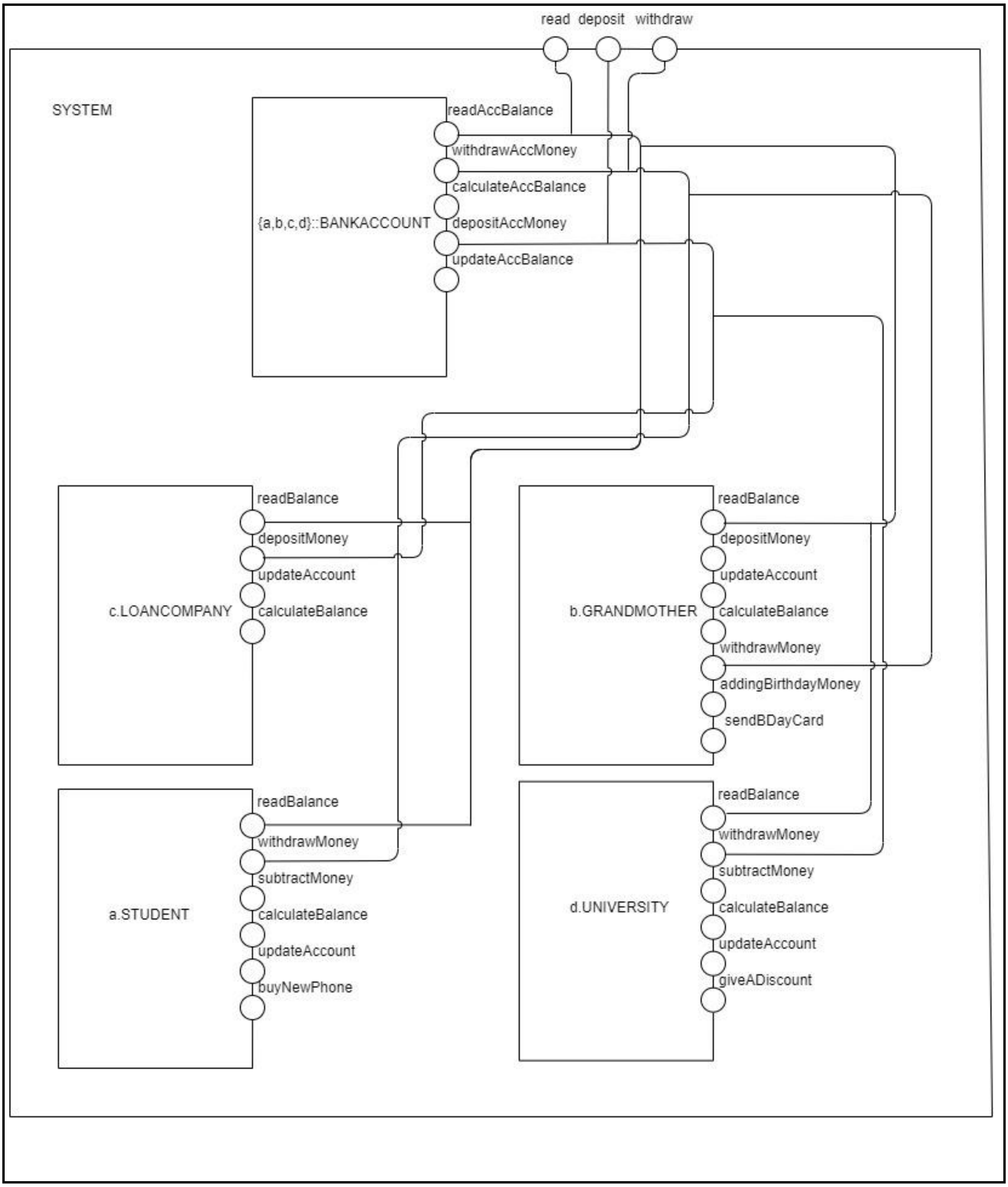
Synchronous Actions	Synchronised by Sub-Processes (List)
a.withdrawMoney, a.readBalance	STUDENT, BANKACCOUNT
b.depositMoney, b.readBalance	GRANDMOTHER,BANKACCOUNT
c.depositMoney, c.readBalance	LOANCOMPANY,BANKACCOUNT
d.withdrawMoney, d.readBalance	UNIVERSITY,BANKACCOUNT

Sub-Process	Asynchronous Actions (List)
STUDENT	buyNewPhone
GRANDMOTHER	N/A
LOANCOMPANY	N/A
UNIVERSITY	N/A

Blocked Synchronising Actions	Synchronising Sub-Processes	Blocking Sub-Processes
d. giveADiscount	UNIVERSITY, BANKACCOUNT	UNIVERSITY
b. sendBDayCard	GRANDMOTHER, BANKACCOUNT	GRANDMOTHER

5. Parallel Composition Structure Diagram

The structure diagram for the parallel composition.



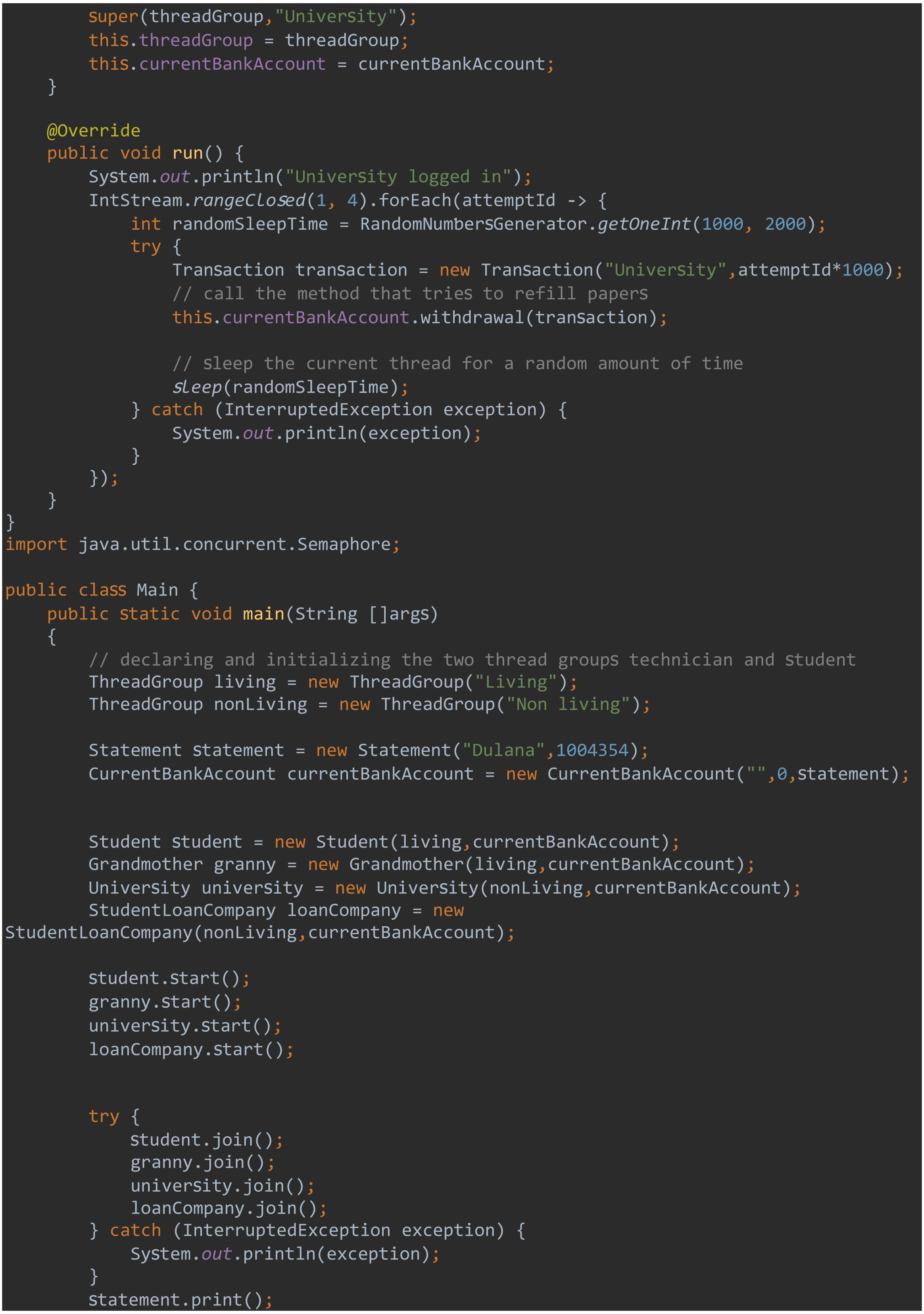
2. LTSA code

```
range BALANCE = 5..5
range TRANSACTION = 1..1
range BALANCENEW = 3..7
range DEPOSIT = 1..1

UNIVERSITY = EXISTINGAMOUNT,
EXISTINGAMOUNT = (readBalance[balance:BALANCE] ->EXISTINGAMOUNT|
withdrawMoney[balance:BALANCE][transaction:TRANSACTION]
->subtractMoney[transaction]->calculateBalance[balance-transaction]->
NEWBALANCE[balance-transaction]),
NEWBALANCE[balance:BALANCENEW] = (updateAccount [balance] ->
EXISTINGAMOUNT)+{giveADiscount}.

STUDENT = EXISTINGAMOUNT,
EXISTINGAMOUNT = (readBalance[balance:BALANCE]->EXISTINGAMOUNT
|
withdrawMoney[balance:BALANCE][transaction:TRANSACTION]->subtractMoney[transaction]-
>calculateBalance[balance-transaction]->NEWBALANCE[balance-
transaction]|withdrawMoney[balance:BALANCE][transaction:TRANSACTION]
->buyNewPhone->subtractMoney[transaction]->calculateBalance[balance-transaction]
->NEWBALANCE[balance-transaction]),
NEWBALANCE[balance:BALANCENEW] =
(updateAccount[balance]->EXISTINGAMOUNT)\{buyNewPhone}.


LOANCOMPANY = EXISTINGAMOUNT,
EXISTINGAMOUNT = (readBalance[balance:BALANCE]->EXISTINGAMOUNT |
depositMoney[balance:BALANCE][amount:DEPOSIT]->calculateBalance[balance+amount]->NE
WBALANCE[balance+amount]),
NEWBALANCE[balance:BALANCENEW] = (updateAccount[balance]->EXISTINGAMOUNT).

GRANDMOTHER = EXISTINGAMOUNT,
EXISTINGAMOUNT = (readBalance[balance:BALANCE]
->EXISTINGAMOUNT|addingBirthdayMoney[balance:BALANCE][amount:DEPOSIT]->calculat
eBalance[balance+amount] ->NEWBALANCE[balance+amount]
|depositMoney[balance:BALANCE][amount:DEPOSIT]->calculateBalance[balance+amount]->NE
WBALANCE[balance+amount]),
NEWBALANCE[balance:BALANCENEW] =
(updateAccount[balance]->EXISTINGAMOUNT)+{sendBDayCard}.

BANKACCOUNT = EXISTINGAMOUNT,
EXISTINGAMOUNT = (readAccBalance[balance:BALANCE]->EXISTINGAMOUNT
|withdrawAccMoney->calculateAccBalance->NEWBALANCE|depositAccMoney->calculateAccB
alance->NEWBALANCE),
NEWBALANCE = (updateAccBalance->EXISTINGAMOUNT).
```

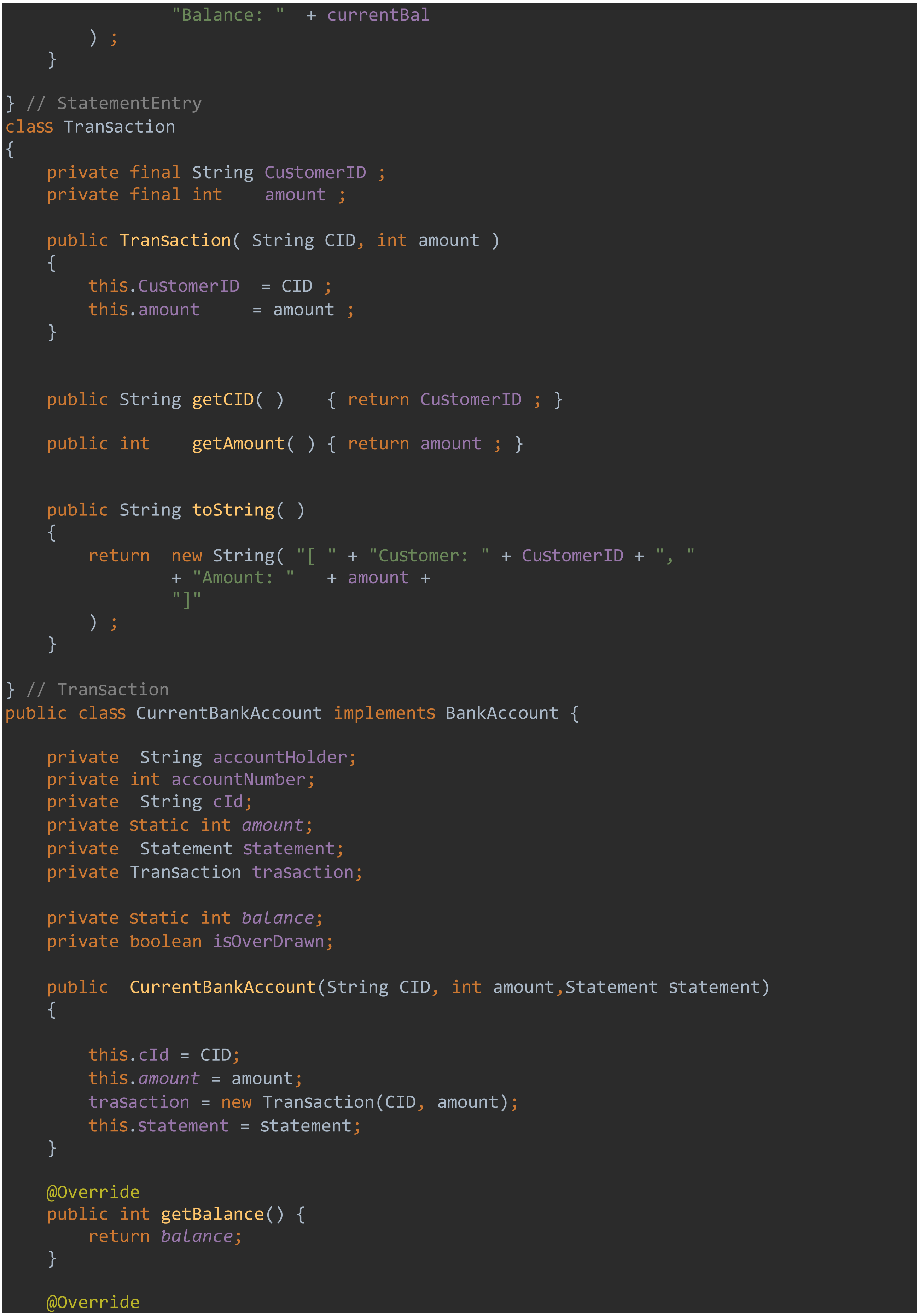
||BANKINGSYSTEM =
(a:STUDENT/{withdrawAccMoney/a.withdrawMoney,readAccBalance/a.readBalance}
|| b:GRANDMOTHER/{depositAccMoney/b.depositMoney,readAccBalance/b.readBalance}
|| c:LOANCOMPANY/{depositAccMoney/c.depositMoney,readAccBalance/c.readBalance}
|| d:UNIVERSITY/{withdrawAccMoney/d.withdrawMoney,readAccBalance/d.readBalance}
||{a,b,c,d}::BANKACCOUNT)@{read,deposit,withdraw}.

3. Java code

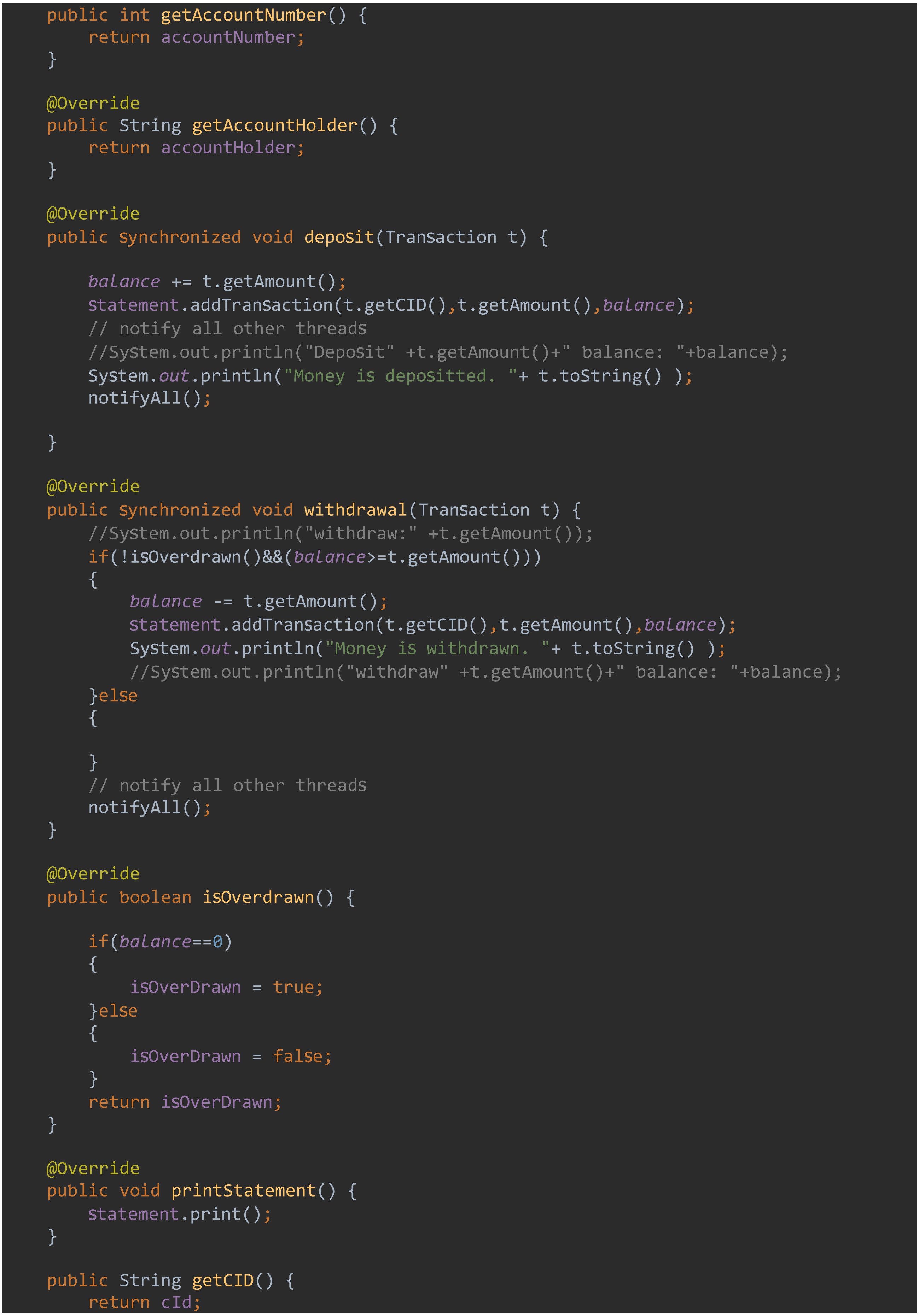
```
import java.util.stream.IntStream;

public class Grandmother extends Thread {

    private ThreadGroup threadGroup;
    private CurrentBankAccount currentBankAccount;

    public Grandmother(ThreadGroup threadGroup, CurrentBankAccount currentBankAccount)
    {
        super(threadGroup,"Granny");
        this.threadGroup = threadGroup;
        this.currentBankAccount = currentBankAccount;
    }

    @Override
    public void run() {
        System.out.println("Grandmother logged in");
        IntStream.rangeClosed(1, 3).forEach(attemptId -> {
            int randomSleepTime = RandomNumbersGenerator.getOneInt(1000, 2000);
            try {
                Transaction transaction = new Transaction("GrandMother",attemptId*1000);
                // call the method that tries to refill papers
                this.currentBankAccount.deposit(transaction);

                // sleep the current thread for a random amount of time
                sleep(randomSleepTime);
            } catch (InterruptedException exception) {
                System.out.println(exception);
            }
        });
    }
}
import java.util.stream.IntStream;

public class Student extends Thread {

    private ThreadGroup threadGroup;
    private CurrentBankAccount currentBankAccount;

    public Student(ThreadGroup threadGroup, CurrentBankAccount currentBankAccount)
    {
        super(threadGroup,"Student");
        this.threadGroup = threadGroup;
        this.currentBankAccount = currentBankAccount;
    }
    @Override
    public void run() {
        System.out.println("Student logged in");
        IntStream.rangeClosed(1, 5).forEach(attemptId -> {
            int randomSleepTime = RandomNumbersGenerator.getOneInt(1000, 2000);
            try {
                Transaction transaction = new Transaction("Student",attemptId*1000);
                this.currentBankAccount.withdrawal(transaction);
                // sleep the current thread for a random amount of time
                sleep(randomSleepTime);
            } catch (InterruptedException exception) {
                System.out.println(exception);
            }
        });
    }
}
import java.util.stream.IntStream;

public class StudentLoanCompany extends Thread {
    private ThreadGroup threadGroup;
    private CurrentBankAccount currentBankAccount;

    public StudentLoanCompany(ThreadGroup threadGroup, CurrentBankAccount
currentBankAccount)
    {
        super(threadGroup,"loanCompany");
        this.threadGroup = threadGroup;
        this.currentBankAccount = currentBankAccount;
    }

    @Override
    public void run() {
        System.out.println("Loan company logged in");
        IntStream.rangeClosed(1, 3).forEach(attemptId -> {
            int randomSleepTime = RandomNumbersGenerator.getOneInt(1000, 2000);
            try {
                Transaction transaction = new Transaction("LoanCompany",attemptId*1000);
                // call the method that tries to refill papers
                this.currentBankAccount.deposit(transaction);

                // sleep the current thread for a random amount of time
                sleep(randomSleepTime);
            } catch (InterruptedException exception) {
                System.out.println(exception);
            }
        });

    }
}
import java.util.stream.IntStream;

public class University extends Thread {
    private ThreadGroup threadGroup;
    private CurrentBankAccount currentBankAccount;

    public University(ThreadGroup threadGroup, CurrentBankAccount currentBankAccount)
    {
```



```
        super(threadGroup,"University");
        this.threadGroup = threadGroup;
        this.currentBankAccount = currentBankAccount;
    }

    @Override
    public void run() {
        System.out.println("University logged in");
        IntStream.rangeClosed(1, 4).forEach(attemptId -> {
            int randomSleepTime = RandomNumbersGenerator.getOneInt(1000, 2000);
            try {
                Transaction transaction = new Transaction("University",attemptId*1000);
                // call the method that tries to refill papers
                this.currentBankAccount.withdrawal(transaction);

                // sleep the current thread for a random amount of time
                sleep(randomSleepTime);
            } catch (InterruptedException exception) {
                System.out.println(exception);
            }
        });
    }
}
import java.util.concurrent.Semaphore;

public class Main {
    public static void main(String []args)
    {
        // declaring and initializing the two thread groups technician and student
        ThreadGroup living = new ThreadGroup("Living");
        ThreadGroup nonLiving = new ThreadGroup("Non living");

        Statement statement = new Statement("Dulana",1004354);
        CurrentBankAccount currentBankAccount = new CurrentBankAccount("",0,statement);


        Student student = new Student(living,currentBankAccount);
        Grandmother granny = new Grandmother(living,currentBankAccount);
        University university = new University(nonLiving,currentBankAccount);
        StudentLoanCompany loanCompany = new
StudentLoanCompany(nonLiving,currentBankAccount);

        student.start();
        granny.start();
        university.start();
        loanCompany.start();


        try {
            student.join();
            granny.join();
            university.join();
            loanCompany.join();
        } catch (InterruptedException exception) {
            System.out.println(exception);
        }
        statement.print();
    }
}
public interface BankAccount {

    int    getBalance( ) ;               // returns the current balance

    int    getAccountNumber( ) ;         // returns the Account number

    String getAccountHolder( ) ;         // returns the Account holder

    void deposit( Transaction t ) ;     // perform a deposit transaction on the bank
account

    void withdrawal( Transaction t ) ;  // perform a withdrawal transaction on the bank
account

    boolean isOverdrawn( ) ;            // returns true if overdrawn; false otherwise

    void printStatement( ) ;            // prints out the transactions performed so far
}
public class  Statement
{
    /********* private Instance Variables *********/

    private final char TAB       = '\t' ;
    private final int  MAX_TRANS = 20 ;

    private final StatementEntry[] statement = new StatementEntry[ MAX_TRANS ] ;

    private final String accountHolder ;
    private final int    accountNumber ;

    private int transactionCount = 0 ;


    /********* public Constructor Method *********/

    public Statement ( String accountHolder, int accountNumber )
    {
        this.accountHolder = accountHolder ;
        this.accountNumber = accountNumber ;
    }


    /********* public Modifier Methods *********/

    public void addTransaction( String CID, int amount, int balance )
    {
        // Create a new statement entry & add to the statement

        statement[ transactionCount ] = new StatementEntry( CID, amount, balance ) ;

        transactionCount++ ;
    }


    public  void print ( )
    {
        System.out.println( ) ;

        System.out.println( "Statement for "  +  accountHolder  +
                "'s Account: "    +  accountNumber    ) ;

        System.out.println( "================================================"        )
;
        System.out.format(  "%1$-20s %2$10s  %3$13s", "Customer", "Amount", "Balance" )
;
        System.out.println() ;
        System.out.println( "================================================"        )
;

        for ( int tid = 0 ; tid < transactionCount ; tid++ )
        {
            //     System.out.println( statement[ tid ] ) ;
            System.out.format( "%1$-20s %2$10d  %3$10d",
                    statement[ tid ].getCustomer(),
                    statement[ tid ].getAmount(),
                    statement[ tid ].getCurrentBalance()
            ) ;
            System.out.println() ;
        }
        System.out.println( "================================================" ) ;
        System.out.println( ) ;

    }

} // Statement
public class StatementEntry
{

    private final char TAB = '\t' ;

    private final String CustomerID ;
    private final int    amount ;
    private final int    currentBal ;


    public StatementEntry( String CID, int amount, int currentBal )
    {
        this.CustomerID = CID ;
        this.amount     = amount ;
        this.currentBal = currentBal ;
    }


    public String getCustomer()      { return CustomerID ; }

    public int    getAmount()        { return amount ;     }

    public int    getCurrentBalance(){ return currentBal ; }


    public String toStringOLD( )
    {
        return  new String(  "Customer: " + CustomerID + ","  + TAB +
                "Amount: "   + amount     + ", " + TAB +
```



```
                "Balance: "  + currentBal
        ) ;
    }

} // StatementEntry
class Transaction
{
    private final String CustomerID ;
    private final int    amount ;

    public Transaction( String CID, int amount )
    {
        this.CustomerID  = CID ;
        this.amount     = amount ;
    }


    public String getCID( )    { return CustomerID ; }

    public int    getAmount( ) { return amount ; }


    public String toString( )
    {
        return  new String( "[ " + "Customer: " + CustomerID + ", "
                + "Amount: "   + amount +
                "]"
        ) ;
    }

} // Transaction
public class CurrentBankAccount implements BankAccount {

    private  String accountHolder;
    private int accountNumber;
    private  String cId;
    private static int amount;
    private  Statement statement;
    private Transaction trasaction;

    private static int balance;
    private boolean isOverDrawn;

    public  CurrentBankAccount(String CID, int amount,Statement statement)
    {

        this.cId = CID;
        this.amount = amount;
        trasaction = new Transaction(CID, amount);
        this.statement = statement;
    }

    @Override
    public int getBalance() {
        return balance;
    }

    @Override
```



```java
    public int getAccountNumber() {
        return accountNumber;
    }

    @Override
    public String getAccountHolder() {
        return accountHolder;
    }

    @Override
    public synchronized void deposit(Transaction t) {

        balance += t.getAmount();
        statement.addTransaction(t.getCID(),t.getAmount(),balance);
        // notify all other threads
        //System.out.println("Deposit" +t.getAmount()+" balance: "+balance);
        System.out.println("Money is depositted. "+ t.toString() );
        notifyAll();

    }

    @Override
    public synchronized void withdrawal(Transaction t) {
        //System.out.println("withdraw:" +t.getAmount());
        if(!isOverdrawn()&&(balance>=t.getAmount()))
        {
            balance -= t.getAmount();
            statement.addTransaction(t.getCID(),t.getAmount(),balance);
            System.out.println("Money is withdrawn. "+ t.toString() );
            //System.out.println("withdraw" +t.getAmount()+" balance: "+balance);
        }else
        {

        }
        // notify all other threads
        notifyAll();
    }

    @Override
    public boolean isOverdrawn() {

        if(balance==0)
        {
            isOverDrawn = true;
        }else
        {
            isOverDrawn = false;
        }
        return isOverDrawn;
    }

    @Override
    public void printStatement() {
        statement.print();
    }

    public String getCID() {
        return cId;
    }

    public void  logMessage()
    {


    }

}
```