Software Transactional Memory
C++14 STM

Design manual

Zoltan Fuzesi

April 1, 2018

Student ID: C00197361
Supervisor: Joseph Kehoe
Institute of Technology Carlow
Software Engineering

Institiúid Teicneolaíochta Cheatharlach
INSTITUTE of TECHNOLOGY CARLOW
At the Heart of South Leinster

Contents

1 Introduction

This document is introducing the design of the C++ STM library logic and structure. The document will describe the flow of the data within the library and the external application that uses the STM library. The use case diagram will discover the possible library API interface functionalities that need to implement in the project. With the use cases and the detailed use cases can characterize the required steps and potential alternative steps during the library execution life cycle. The domain model shows the main functionality of the library and the association between the software components, follows with the sequence diagram that helps to understand the sequence of steps required to implement lock free, object based locking atomic, transactional mechanism.

This document helps to discover the potential errors and required handlers with the use cases and the sequence diagram. However, this diagrams and use case descriptions are identify the potential issues and risks, that may can mislead the future development if not well designed. To use, or test the STM library implementation, the programmers must to write their own client application and includes the library, like any other (external) libraries in C or C++ programming design. The usage of the library will happen with the API function calls, that will be described in the use case diagram and studies thoroughly in the detailed use cases.

2 Model of the application

The flow diagram *Figure 2* introducing the dynamic relationship of the STM library. This relations interacting with the client application and the objects delegated by the STM library. In order to use the STM library, the programmers must include it in the main module, to create an instance of a transaction manager, associated with the application by the Process ID to use the API interfaces. When the instances is created then the API function will be available to the client application. The key of the library usage, the connected processes can not instantiate more then one transaction manager object of the library, because the library use singleton object instantiation, that keep track of all process accessed the transaction manager by the process ID. This way the library can register all connected process in the built in Data structure.

2.1 Library integration to the client application

The final solution will include shared and static library solution. The shared library is linked to the client application at compilation time, and the code can be shared between multiple processes. The static library code gets integrated to the client application, which is not shared with other processes. With the booth solution the header files must used as a reference to the library API functions. When the executable created, the header files and the static library file are not required any more. The shared library should be kept somewhere in the operating system.

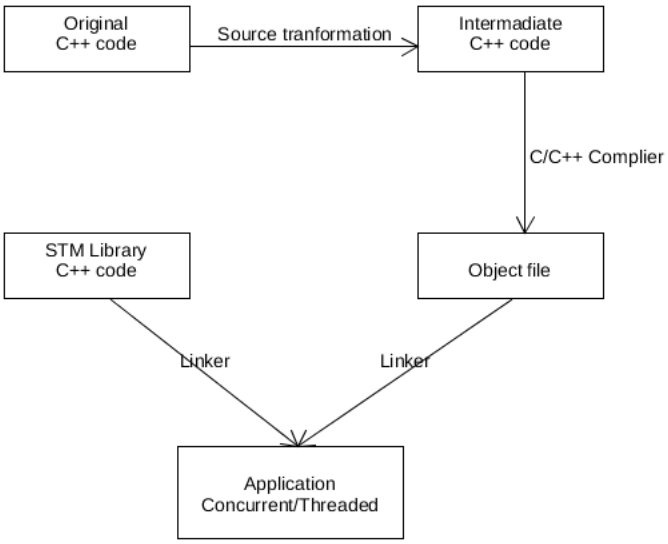

Figure 1: *STM library linked to application.*

2.2 Flow diagram

When the **create transaction** API call happens by the application, the library place a new transactional object into the built in data structure, associating by the process ID and returning back a transactional object to the given application.

The **start transaction** API call will starting the process of the Object transactional life cycle. First, place the object into the (global shared) read set, and creates a copy of it to the write set, to use it for local temporary changes, and add a version number to it, as a counter to determine the Object changes. Because more then one thread can use the same object, it must check the version number at the first step to determine the changes at early stage of the object transactional life cycle. If the version number in the object is different then the global version number, the transaction will cancel and restart. If the version number that the object hold and the global version number in the write set are same, then the copy of the object in the write set will update to the new required value. Before the original object can change from the local copy, the transaction library must check the version numbers equality again. If the version numbers are same, that means no other process accessed the same object during the updating process, and the object can get assign the new value. Otherwise, the transaction must cancel and

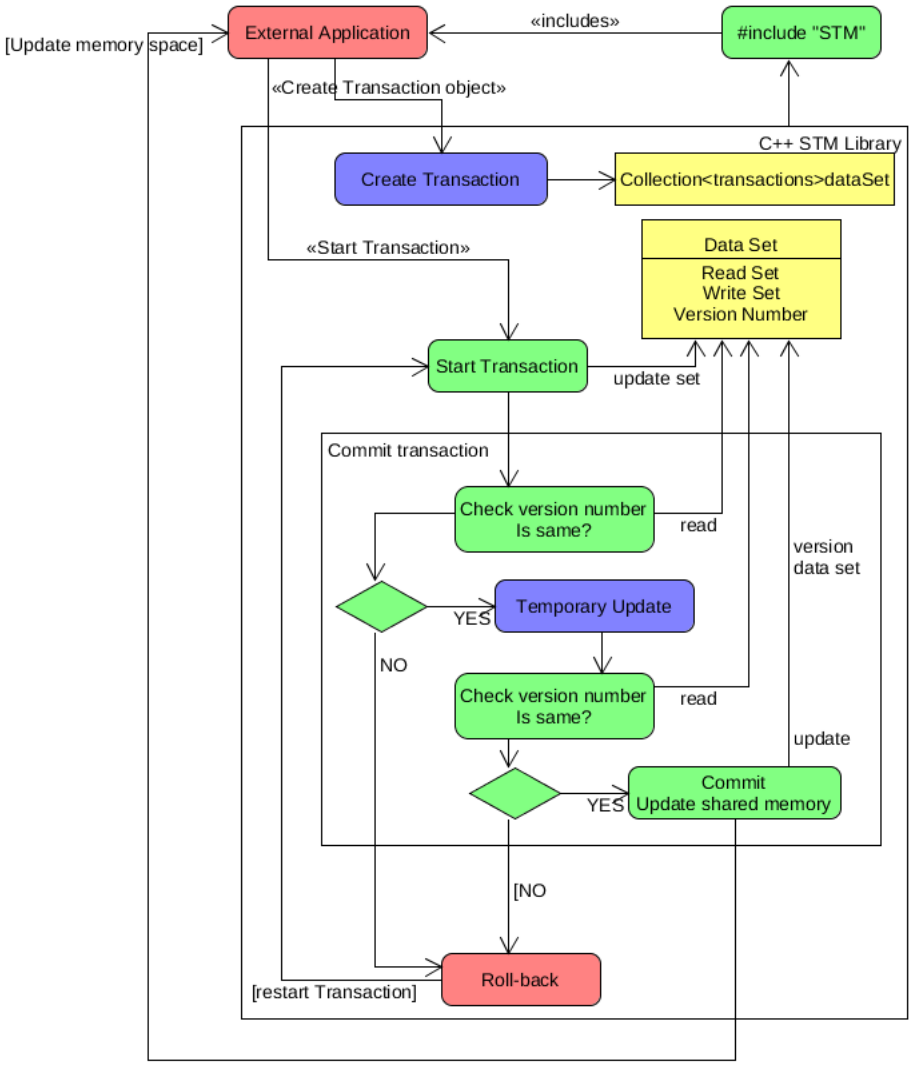

Figure 2: *The flow of the data in the library.*

the life cycle restart again.

In this phase of the program execution, the transaction marked as processable process, and the **commit method** available for execution. This API call will update the original object, update/increase the version number assigned to it and

returns back to the main process to finish it execution. If other processes or thread are still using the same object, they must detect the changes on the version number, and restart the transactional if required.

3 API usage

This part of the document will design the library API interface function(s), that will use by the external application. First, the library function (API) calls will be demonstrated with a simple test application, then the library methods.

```
#include <STM.h>
#include <sstream> //get the thread id
#include <unistd.h> //get the Application process id

//Create a Transaction manager
TM transactionManager;
//Create a transaction object
TX transaction;

class Account : public Tobj{
        private:
                        int amount;
                        int accountID;
                        string accountHolder;
                        Account(int amount, int accountID, string accountHolder):Tobj(){
                                this->amount = amount;
                                this->accountID = accountID;
                                this->accountHolder = accountHolder;
                        }
        public:
                void addAmount(double amount){
                        this->amount += amount;
                }

                void removeAmount(double amount){
                        this->amount -= amount;
                }
                double getAmount(){
                        return this->amount;
                }
                int getAccountId(){
                        return accountID;
                }

}

//Single object operation
bool updateAccount(std::shared_ptr<Account> account, double amount){

        //All threads updating account * times
        for(int i = 0; i < updateTimes; ++i)
                {


                        Account tmp = loadTransactionObject(std::this_thread::get_id());
                        tmp.addAmount(*amount);

                        transaction.storeTransaction(std::this_thread::get_id(), &tmp);
                }

                barrier->Wait();
                bool done = false
                //All threads commits their transactions
                while(!done){
                        Account tmp = loadTransactionObject(std::this_thread::get_id());
                        done = transaction.commitTransaction(std::this_thread::get_id(), &tmp);
                }
                barrier->Start();

        return true;
}


//multiple objects operations
bool exchange(double amount, std::vector<Account>*acconts){
```

```
        //Create boolean value to check the transaction state
        bool done = false;

        while(!done){
                //make changes on the copy of the objects
                acconts[0].addAmount(amount);
                acconts[1].removeAmount(amount);

                //The transaction updates the local copy of the object in the library
                transaction.storeTransaction(std::this_thread::get_id(), &acconts);

                //The transaction try to update the values in the transaction
                //if the values update then return true otherwise return false
                done = transaction.commitTransactions(*accounts);
        }
        return true;
}

int main(){
         //get the process id
         pid_t ppid;
     ppid = getppid();
         //create a thread
         std::thread threadOne;
         //Create shared poinetrs of the bank accounts
         std::shared_ptr<Account> from(100,1234,"JOE");
         std::shared_ptr<Account> to(50,2345,"BOB");

        //add the thread to the TM manager HasMap and return the transaction
        transaction = transactionManager.createTX(ppid);

        //transaction add the account object to the transaction HasMap read and write sets and return the local copy
        auto copyFrom = transaction.startTransaction(std::this_thread::get_id(), &from);
        auto copyTo = transaction.startTransaction(std::this_thread::get_id(), &to);

        //Add the local copies to the vector to store all the associated transactions
        std::vector<Accont*> accounts;
        accounts.push_back(copyFrom);
        accounts.push_back(copyTo);

        //Amount to work with
        double amountToExchange = 100.5;

        //Thread call the function and passing all the required variables to operate the transaction
        threadOne=std::thread(exchange, copyFrom, copyTo, amountToExchange, &accounts);

        //create a thread
         std::thread threadTwo, threadThree, threadFour, threadFive, threadSix;

         //All the threads are calling the same function with the same object to make changes
         threadTwo   = std::thread(updateAccount, copyFrom, 1, 5);
         threadThree = std::thread(updateAccount, copyFrom, 1, 5);
         threadFour  = std::thread(updateAccount, copyFrom, 1, 5);
         threadFive  = std::thread(updateAccount, copyFrom, 1, 5);
         threadSix   = std::thread(updateAccount, copyFrom, 1, 5);


        threadOne.join();
        threadTwo.join();
        threadThree.join();
        threadFour.join();
        threadFive.join();
        threadSix.join();



        //Otherwise destroy the manager that clean out the memory from
        //all the transactions
        transactionManager.destroy(ppid);

        return 0;
}
```

3.1 Create transaction

```
//Create transaction associated with Application
TX createTX(pid_t ppid){

        MAP::const_iterator position = transactions.find(ppid);
        TX tx;
        if (position == map.end()) {
            tx = new TX();
                transactions[ppid] = tx;
        }

        retrun Tx;
}
```

3.2 Destroy transaction

```
//Destroy transaction associated with Application
bool destroyTX(pid_t ppid){

        MAP::const_iterator position = transactions.find(id);
        if (position == map.end()) {
            return true
        }else{
                transactions[ppid] = null;
        }

        return true;

}
```

3.3 Start transaction

```
//store object in read and write set
bool startTransaction(std::thread::id id, Tobj* obj){

        MAP::const_iterator position = readSet.find(id);
        //Update data set only if not registered
        if (position == map.end()) {
                obj.setVersionNumber(0);
            //register object to the read set
                readSet[id] = boj;
                //Call the copy constructor to make a copy of the object
                Tobj objCopy = obj;
                objCopy.setVersionNumber(0);
                //Register object to the write set as a copy of the original
                Tobj object = copyObject(obj)
                writeSet[id] = object;
        }

        return true;
}
```

3.4 Commit transaction

```
//Commit Transaction , single object transaction
bool commitTransaction(std::thread::id id, Tobj* obj){
        //Compare the working write set object version number with the read set version number
        if(obj.getVersionNumber() == getVersionNumber(id)){
                //Update write set object values
                updateTransaction(*obj);
                //before change the readset checks again for version number
                if(obj.getVersionNumber() == getVersionNumber(id)){
                        updateSharedMemory(*obj);
                        obj.incrementVersionNumber();
                        return true;
                }else{
                        rollbackTransaction(id);
                        return false;
```

```
                }
        }
        //If not same return false and restart
        return false;
}
```

3.5 Commit transactions

```
//Commit Transactions , multiple objects transactions
bool commitTransactions(std::thread::id id, vector<Tobj*> & accounts){
        for (int i=0; i<accounts.size(); i++)
    {
                //get the object
                Tobj* obj = getObject(id);
        if(accounts[i].getVersionNumber())  == obj.getVersioNumber(){
                //Update write set object values
                        updateTransaction(*obj);
                        if(accounts[i].getVersionNumber())  == obj.getVersioNumber(){
                                updateSharedMemory(*accounts[i]);
                                *accounts[i].incrementVersionNumber();
                                //If all object has updated
                                if(i == accounts.size()-1){
                                        return true;
                                }
                        }else{
                                rollbackTransaction(id);
                                return false;
                        }
        }
    }


        return false;
}
```

3.6 Load Transaction Object

```
//Load transaction return object
Tobj* loadTransactionObject(std::thread::id id, Tobj* obj){

        return &writeSet[id];
}
```

3.7 Store transaction

```
//Store transaction is updating the object in the write set
bool storeTransaction(std::thread::id id, Tobj* obj){
                //Update/overwrite collection value by
                writeSet[id] = obj;

        return true;
}
```

3.8 Store transactions

```
//Store transaction is updating the object in the write set
bool storeTransactions(std::thread::id id, vector<Tobj*> & accounts){
        //Update/overwrite collection value by
        writeSet[id] = accounts;

        return true;
}
```

4 Use case

This section of the document the use case diagram will list the API actions or event steps, to define the interactions within the processes.

Figure 3: *The STM library API actions.*

4.1 Brief case

4.1.1 Create transaction

Use case : Create transaction.
Actors : Application
Description : This use case begin when Application wants instantiate an object of the STM transactional library. The library registering a new transactional object by the process Id and returning the transactional object that can use the library API interface, which ends this use case.

4.1.2 Destroy transaction

Use case : Destroy transaction.
Actors : Application

Description : This use case begin when Application wants to destroy the object of the STM transactional library associated with the application. The application calls the API function and send the process Id, that is deleting the object from the data set, which ends this use case.

4.1.3 Start transaction

Use case : Start transaction.
Actors : Application
Description : This use case begin when Application wants to add a new object to the STM library. The transactional object call the API function and send the Object and the thread id which wants to operate on it. This object will be registered in the library read data set and a copy of it in the write data set, which ends this use case.

4.1.4 Start transactions

Use case : Start transactions.
Actors : Application
Description : This use case begin when Application wants to add new objects to the STM library. The transactional object call the API function and send the collection of Objects with the thread id which wants to operate on it. This objects will be registered in the library read data set and a copy of it in the write data set, which ends this use case.

4.1.5 Commit transaction

Use case : Commit transaction.
Actors : Application
Description : This use case begin when Application wants commit or make permanent changes on the object shared value, while that is not available to the other processes. The transactional object calls the commit transaction function call. The library checking the version number associated with the thread ant the object. If the version numbers are same in the global data set and the object itself, then the STM library updates the read set and checks the write set again to the version

number. If the version number is still the same, then the library updates the shared memory space permanently and indicate the application that the object have been updated, which ends this use case.

4.1.6 commit transactions

Use case : Commit transactions.
Actors : Application
Description : This use case begin when Application wants commit or make permanent changes on more objects shared value, while those are not available to the other processes. The transactional object calls the commit transaction function call. The library goes through on all the objects in the collection set, and doing the following process on each object.First checking the version number associated with the object. If the version numbers are same in the global data set and the object itself, then the STM library updates the write set and checks the read set again to the version number. If the version number is still the same, then the library updates the shared memory space permanently and indicate the application that the object have been updated, which ends this use case.

4.1.7 Load Transaction object

Use case : Load Transaction object.
Actors : Application
Description : This use case begin when Application wants to receive the copy of the object from the STM library. The application calls the API function and passing the thread Id, that is returning back the copy of the object associated with the thread Id, which ends this use case.

4.1.8 Store transaction

Use case : Store transaction.
Actors : Application
Description : This use case begin when Application wants to store an object involved in the transaction. The transaction object calls the store transaction API function, that will save the object associated with the thread id, which ends this

use case.

4.1.9 Store transactions

Use case : Store transactions.
Actors : Application
Description : This use case begin when Application wants to store more object involved in the transaction. The transaction object calls the store transactions API function, that will goes through on the object collection and save all the objects associated with the thread id, which ends this use case.

4.2 Detailed use case

4.2.1 Create transaction

Name : Create transaction.
Actors : Application
Main Scenario.

1. The use case begins when the Application wants to create a transactional object.
2. The Application calls the create transaction function in the transaction manager.
3. The library check for the existence of the transaction.
4. The library returns the transactional object to the Application.

Alternatives
4a. The transaction is not exist the library need to create.

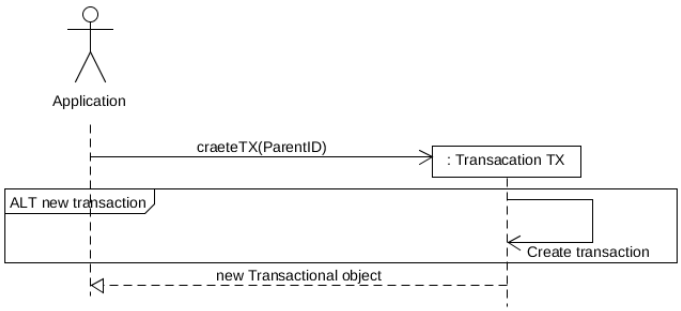

Figure 4: *Create transaction.*

4.2.2 Destroy transaction

Name : Destroy transaction.
Actors : Application
Main Scenario.

1. The use case begins when the Application wants to destroy the transaction object.
2. The Application calls the destroy transaction function in the transaction manager.
3. The library check for the existence of the transaction.
4. The library delete the transaction
5. The library returns the transactional object to the Application.

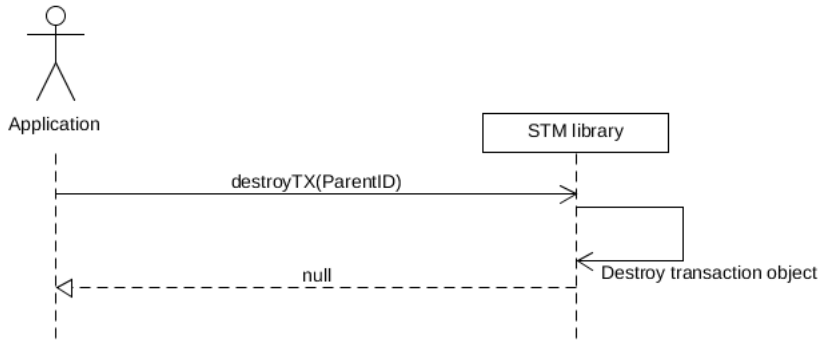

Figure 5: *Destroying transaction object.*

4.2.3 Start transaction

Name : Start transaction.
Actors : Application
Main Scenario.

1. The use case begins when Application wants add an object to the transactional library.
2. The Application call the start transaction API function and passing the object and thread Id to the library.
3. The library registering the object to the read data set and creates a copy of it to the write data set.

4. The library assign a version number to the registered object.

5. The library indicate the Application that the object registered with the returned copy of it.

Alternatives
3a. The object already in the data set don't need to be register.

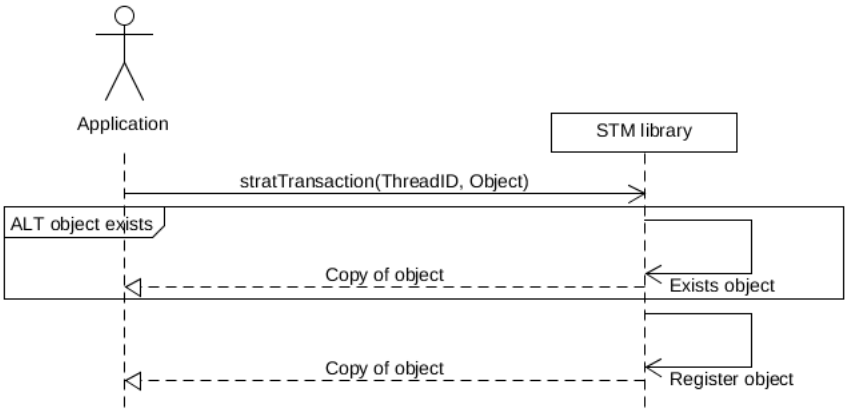

Figure 6: *Start transaction.*

4.2.4 Start transactions

Name : Start transactions.
Actors : Application
Main Scenario.

1. The use case begins when Application wants add collection of objects to the transactional library.

2. The Application call the start transaction API function and passing the object collection to the library.

3. The library goes through on the collection and registering each objects to the read data set and creates a copy of it to the write data set.

4. The library assign a version number to each registered objects.

5. The library indicate the Application that the object registered with the returned copy of the collection of objects.

Alternatives
3a. The objects already in the data set don't need to be register.

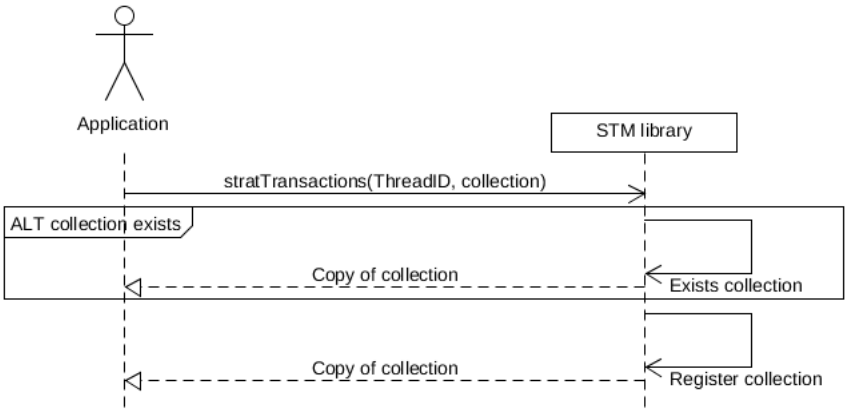

Figure 7: *Start transactions.*

4.2.5 Commit transaction

Name : Commit transaction.
Actors : Application
Main Scenario.

1. The use case begins when the Application wants to make the changes on the object value.

2. The Application calls the commit transaction function call on the API and passing the required values to exchange within the object.

3. The library checks the version number associated with the object.

4. The library make changes on the copy of the object in the local data set.

5. The library checks again the version number associated with the object.
6. The library make the permanent change on the object.
7. The library indicates the Application that the changes has made with the true boolean answer.

Alternatives
3a. The library early find the difference between the version numbers.
3b. The library cancel the transaction and indicates the Application with a false answer.
5a. The library later stage find the difference between the version numbers.
5b. The library cancel the transaction and indicates the Application with a false answer.

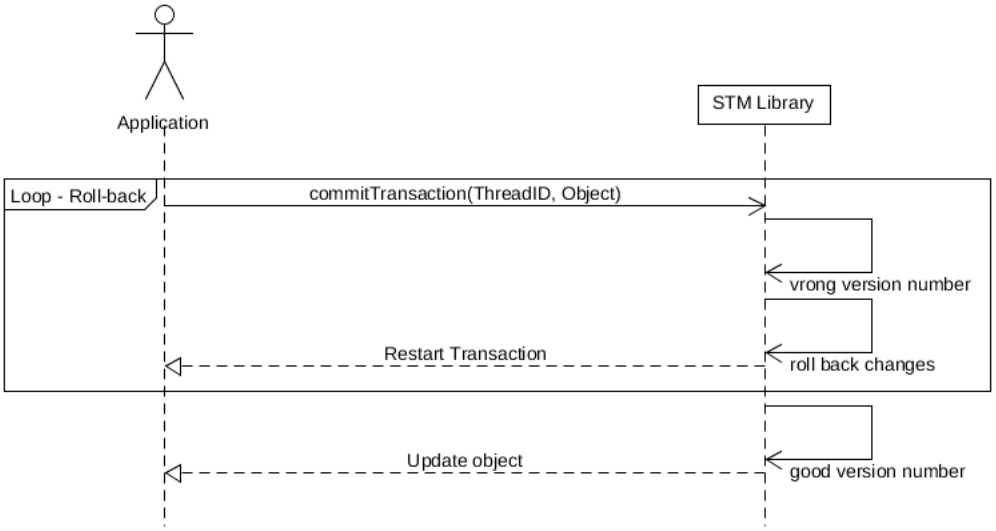

Figure 8: *Commit transaction.*

4.2.6 Commit transactions

Name : Commit transactions.
Actors : Application
Main Scenario.

1. The use case begins when the Application wants to make the changes on collection of object values.
2. The Application calls the commit transaction function call on the API and passing the required collection to exchange within the objects.
3. The library goes through every single object.
4. The library checks the version number associated with the object.
5. The library make changes on the copy of the object in the local data set.
6. The library checks again the version number associated with the object.
7. The library make the permanent change on the object.
8. The library indicates the Application that the changes has made with the true boolean answer.

Alternatives
3a. The library early find the difference between the version numbers.
3b. The library cancel all the transaction and indicates the Application with a false answer.
5a. The library later stage find the difference between the version numbers.
5b. The library cancel all the transaction and indicates the Application with a false answer.

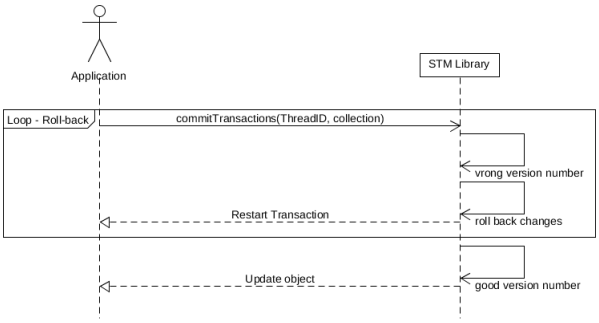

Figure 9: *Commit transactions.*

4.2.7 Load transaction

Name : Load transaction.
Actors : Application
Main Scenario.

1. The use case begins when the Application wants receive a copy of an object.
2. The Application calls the load transaction object function in the library API.
3. The library returning back the object determined by the id.

Alternatives
3a. The library can not find the Object by the id..

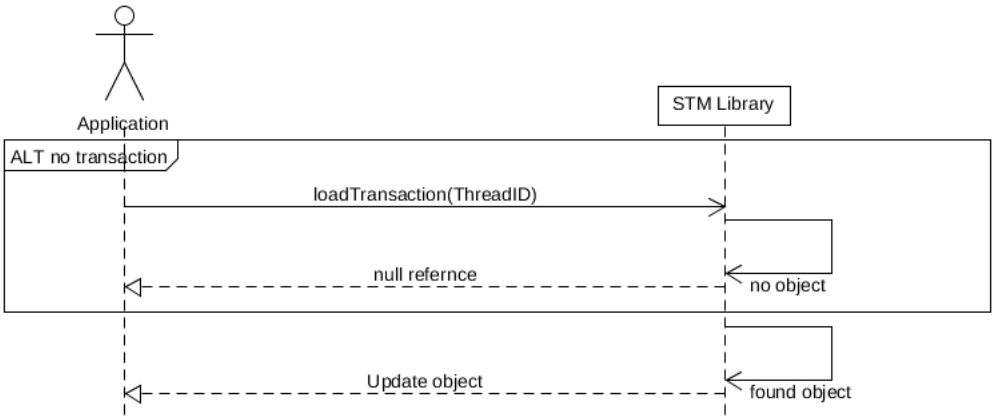

Figure 10: *Receive back object from transaction.*

4.2.8 Store transaction

Name : Store transaction.
Actors : Application
Main Scenario.

1. The use case begins when the Application wants to register a single object in the library.

2. The Application calls the store function and sending the object and the thread id to the library API interface.

3. The library registering the object by the id.

Alternatives
3a. The object is null equivalent.

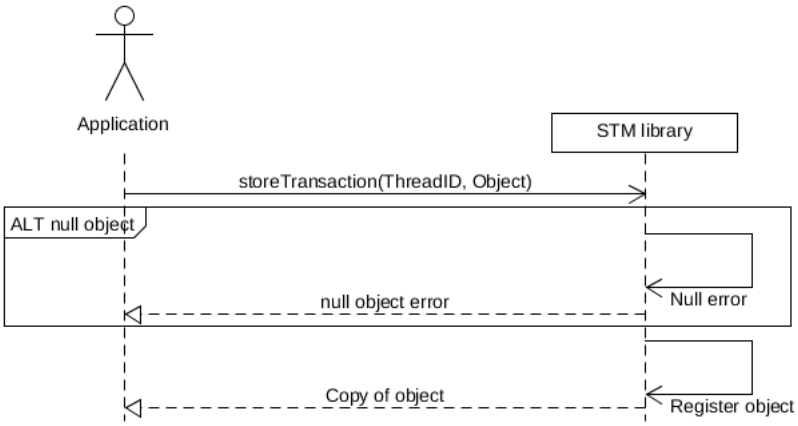

Figure 11: *Add object to transaction.*

4.2.9 Store transactions

Name : Store transactions.
Actors : Application
Main Scenario.

1. The use case begins when the Application wants register a collection of objects.

2. The Application calls the store function and sending the object collection to the library API interface.

3. The library goes through and registering each objects by the id.

Alternatives

3a. The collection contains null equivalent object.

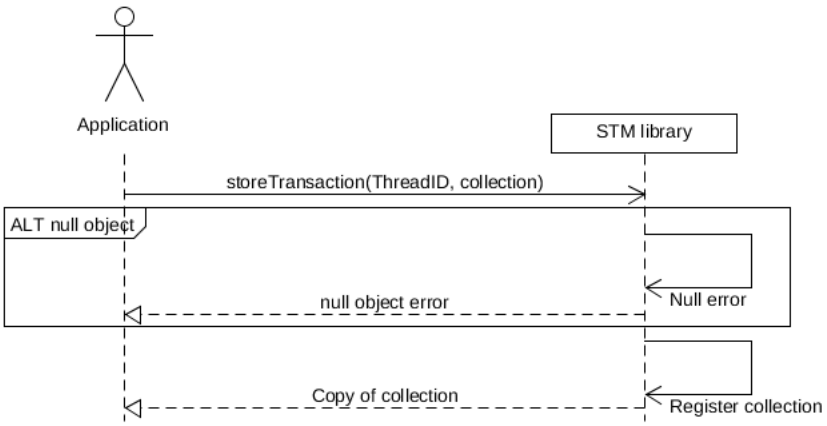

Figure 12: *Add collection to the transaction.*

5 System sequence diagram

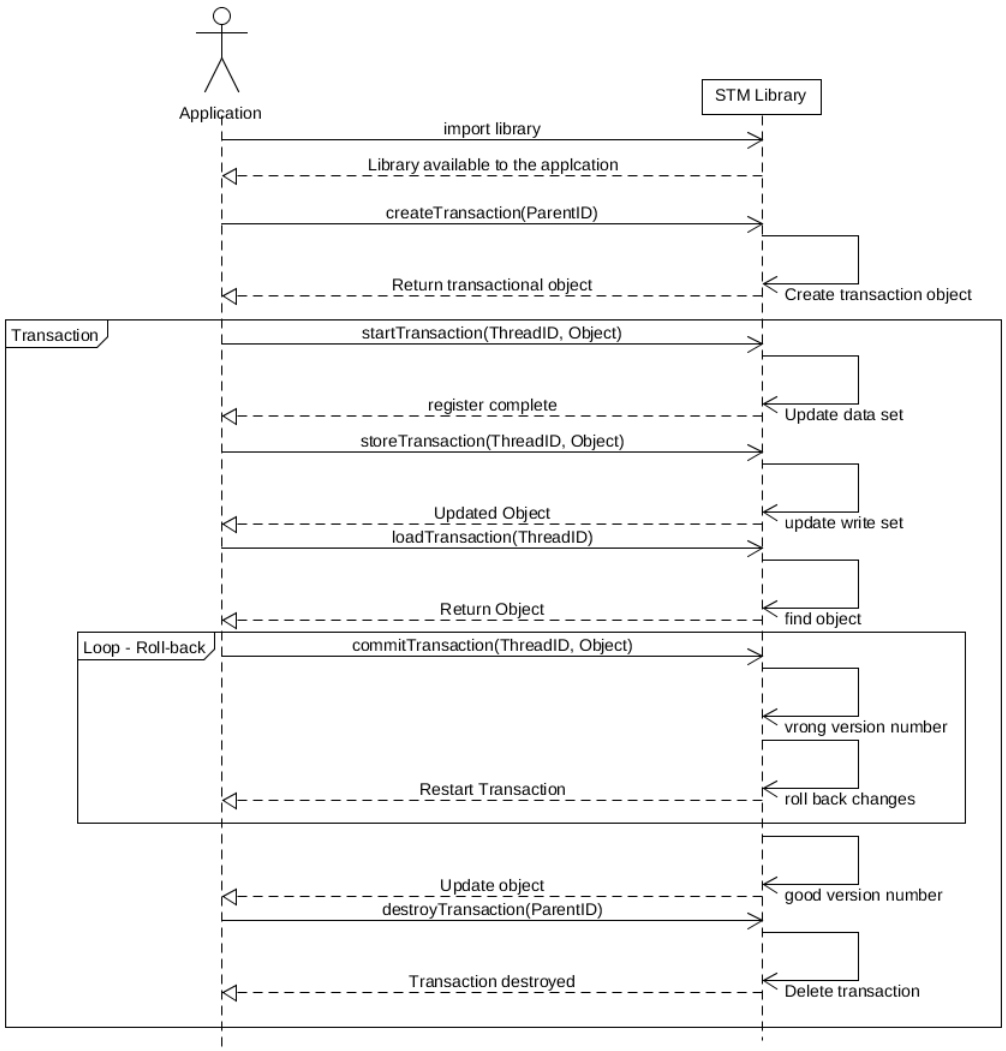

Figure 13: *The system sequence diagram.*

6 Domain model

The domain model is quiet simple since the library do not need include many classes and functions to implement the Software Transactional Memory implementation with the basic functions. Of course, many STM library has implementation of different binary trees as well, that make it more bigger and complex.

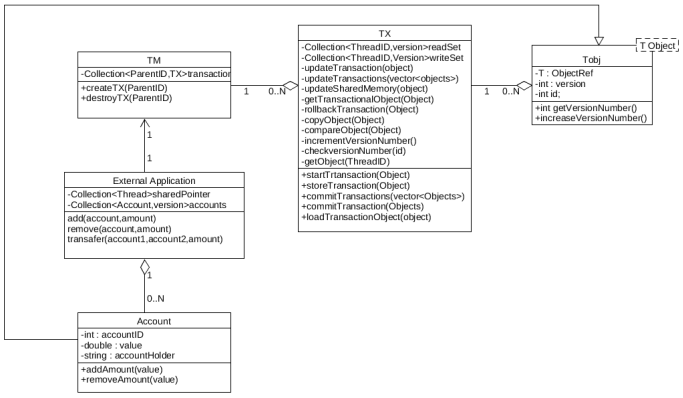

Figure 14: *The class hierarchy.*

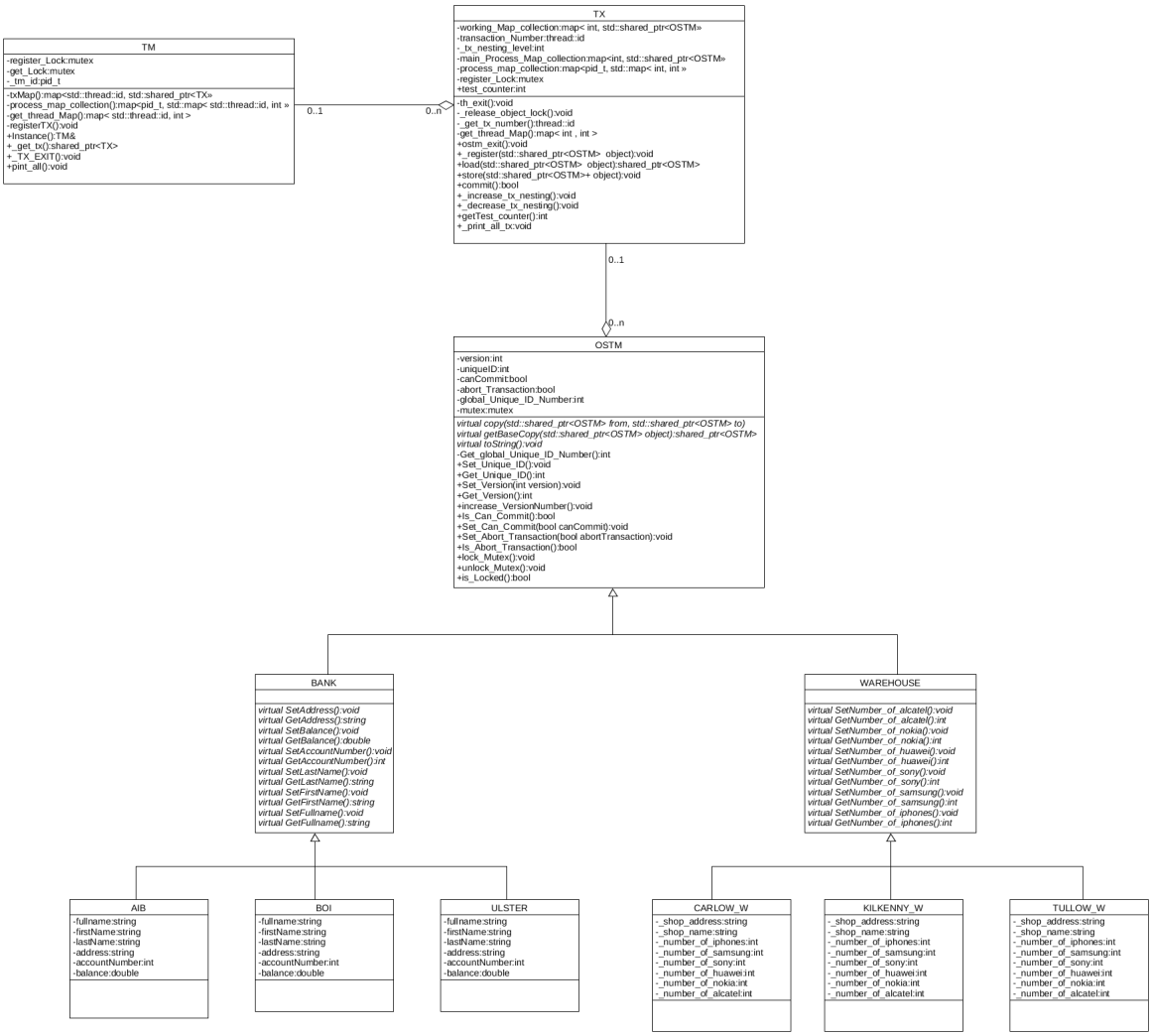

Figure 15: *Updated Domain Model.*

7 Testing the STM Library

One way to test the Software Transaction Memory library functionalities, is to create a client application and include the shared library in it. The other way is to use a C++ Testing framework such as CppUnit to test API functions and the library itself. As the library getting more functionality during the development, those functions should be tested one by one. Because the functions have return

values, it makes the testing process more easier.

The functions can be tested separately like:

1. The library available from the test application.
2. The library can be instantiated by the application.
3. The instance of the library object can access the API function.
4. The functions are working properly as they are expected.
5. The library can lock the shared variable in front of the other processes.
6. The library able to detect the version conflicts during the transaction.
7. The library returns the correct values after the transactions.

The library should able to handle single or multiple threaded environments as well.

Multi threaded multiple object exchange test : 10 threads
Multiple Threads accessing Object1 and Object2.
Object1 balance is 100 unit.
Object2 balance is 100 unit.
Every transaction transfer 5 unit from Object2 to Object1 Object1 receive 50 (10*5) unit from Object2.
After transactional process Object1 has 150 unit and Object2 has 50 unit.

Multi-threaded single Object test : 10 threads:
Object1 has 100 unit.
Every thread increase the amount on Object1 by 1 unit.
After the test Object1 has 110 units.

Single-threaded multiple object test: 1 thread
One thread accessing collection of Objects : 10 Objects.
Every Objects in the collection has 10 unit.
The thread (process) goes through on each objects 5 times and increase the amount by 1 units.
After the test all Object in the collection has 15 units.

Multi-threaded multiple Objects test : 10 threads
Ten threads are accessing a collection of 10 Objects.
Every Objects has 10 unit.
Every threads will accessing every Objects in the collection and increasing the amount by 1 unit.
After the test every Object has 20 units.

Test the library with two different running processes, that has different process id, and perhaps hold two different transactional objects. Any of the above testing method can apply for this test, using by two applications in the same time.

8 Conclusion

The design document helps to discover the functionality and the association between system components. Give a clearer picture of the project functions, sequence of the processes and relation between the application and the library.

Because this project developed in an Agile way, the described functions, source code and system components my will change during the future development in the project life cycle. The designed Software Transactional Library is operating on inherited objects instead of primitive type of variables. So, the programmers must inherit from the Tobj in order to use the STM library. However, if it will make major difficulties during the development, it may will change back to operate on primitive types, that cause to redesign the domain of the software design.