AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

# Architecture and Design of Embedded Real-Time Systems

## Journal on Assignment 4

## Group 5

Authors:
Daniel Ejnar Larsen (201406535)
Christian Lillelund (201408354)

Supervisor:
Jalil Boudjadar

## Contents

### Revision History

| Revision | Date/Authors | Description |
|---|---|---|
| 1.0 | 04.12.2018/All | Document created, written and finished. |
| | | |
| | | |
| | | |

# 1 Introduction

This assignment is a refinement of assignment 3 where we will refine the implementation to incorporate a concurrent state machine and use the active object pattern**.**

## 1.1 Intro to requirements for the exercises

Assignment 3 needs to be extended with the implementation of AND-States in the State Pattern, implementing an active object.

The execution of the AND-states shall be concurrent using threads

## 1.2 Patterns used in the solution

- Singleton
- State Pattern
- Active Object Pattern
- Command Pattern

The new pattern in this assignment compared to assignment 3 are the active object pattern.

# 2 Solution

## 2.1 Introduction to architecture and decisions

The difference in architecture to assignment 3 is mostly the AND-states. These is an addition to the last assignment where the RealTimeLoop is now decomposed as ApplicationModeSetting, which is basically the same as the RealTimeLoop were in assignment 3, and SimulateRealTimeState, which is running in parallel with ApplicationModeSetting and is implemented as an active object.

The command pattern is used when executing the commands sent to SimulateRealTimeState.

## 2.2 Logical View

### 2.2.1 Use Case View

From the assignment description, a user interacts with the EmbeddedSystemX class and calls events on this class.

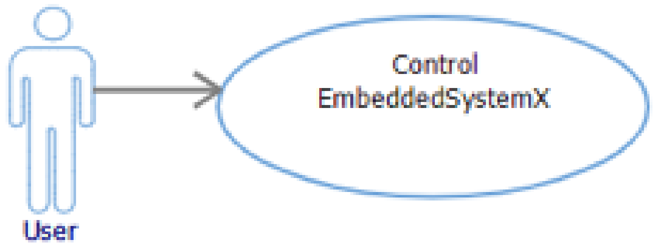

Figure 1. Use case diagram [Assignment3.pdf]

### 2.2.2 Class diagram(s)

The class diagram of the system can be seen below. Notice the introduction of the SimProxy class. This object will run in a thread serving requests from the client (EmbeddedSystemX) calling RealTimeExecution. When Simulate() is called, it will create new Simulation objects (commands) and put them on a queue and the SimProxy will be responsible for executing the them. SimProxy has a FIFO queue which the client has access to via RealTimeExecution.

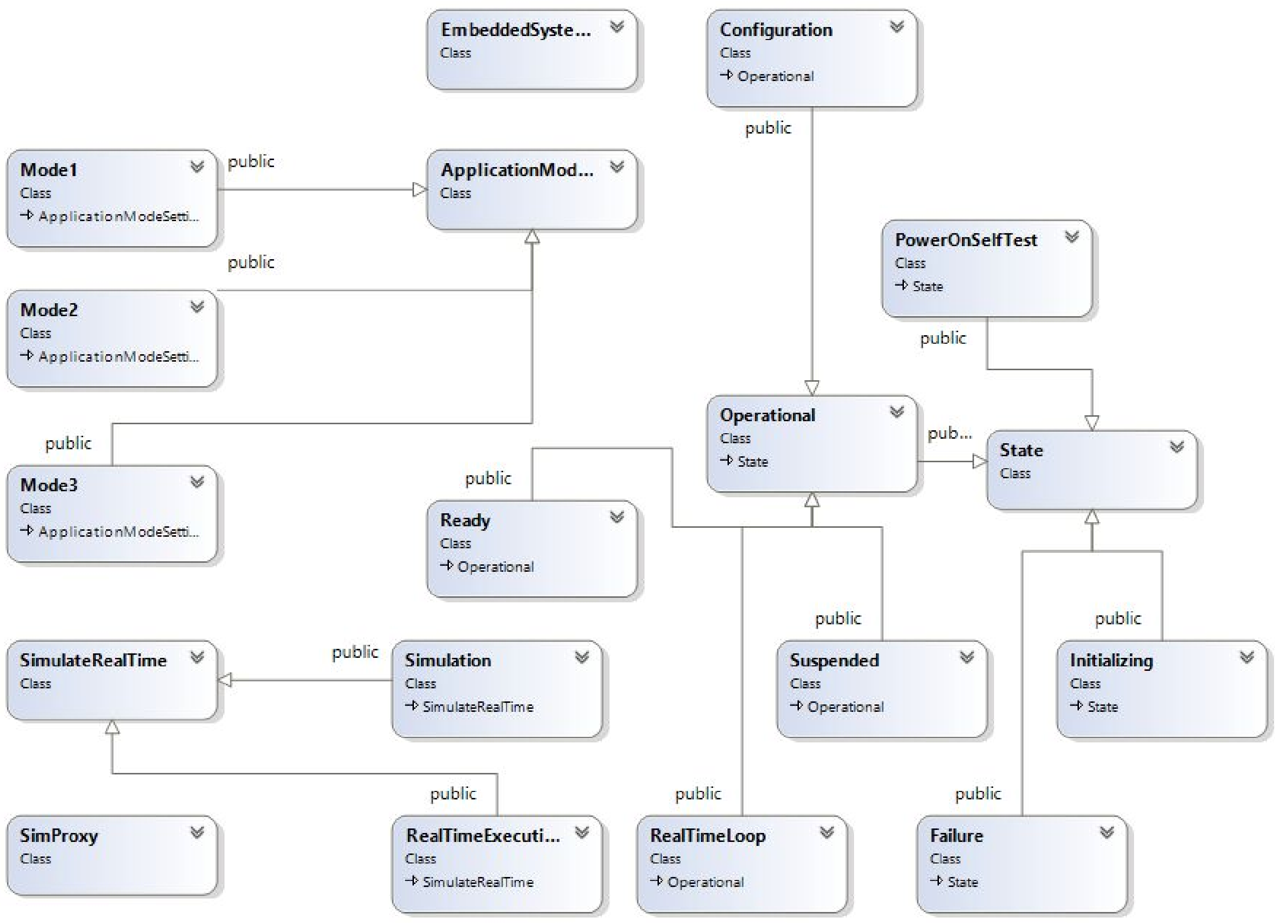
EmbeddedSyste... Class
Configuration Class → Operational
public
Mode1 Class → ApplicationModeSetti...
public
ApplicationMod... Class
public
PowerOnSelfTest Class → State
Mode2 Class → ApplicationModeSetti...
public
Operational Class → State
pub...
State Class
public
Mode3 Class → ApplicationModeSetti...
public
Ready Class → Operational
public
public
SimulateRealTime Class
public
Simulation Class → SimulateRealTime
Suspended Class → Operational
Initializing Class → State
public
public
public
SimProxy Class
RealTimeExecuti... Class → SimulateRealTime
RealTimeLoop Class → Operational
Failure Class → State

Figure 2. Class diagram of the solution.

### 2.2.3 Sequence diagram(s)

The sequence diagram for the active object can be seen below. This is the only part changed majorly and therefore the only one shown here.

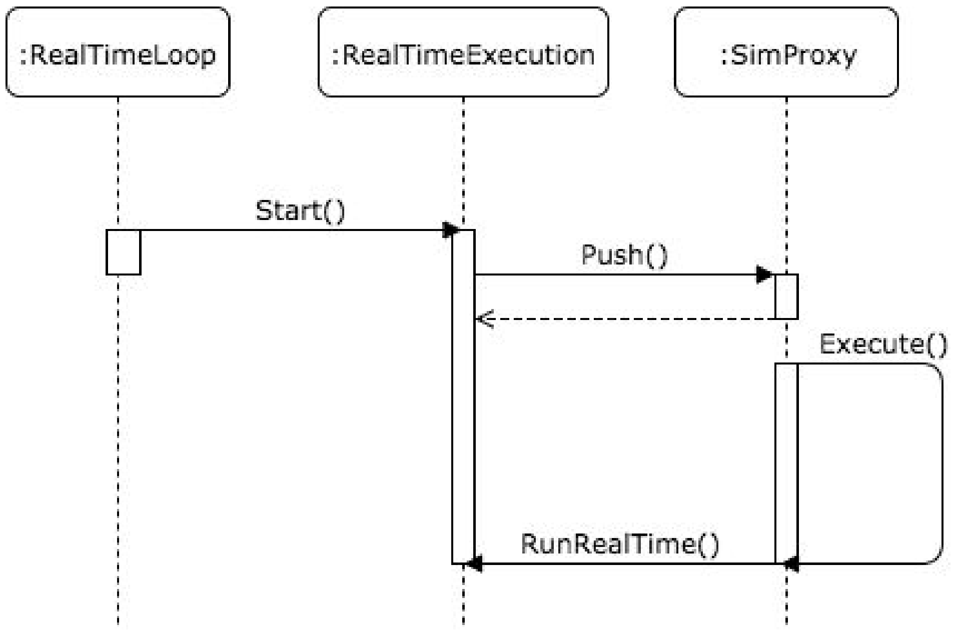

Figure 3. Sequence diagram for the interaction between RealTimeLoop, the execution and our proxy (the active object). RealTimeExecution creates new Simulation objects, pushes them on a queue and SimProxy executes them asynchronously. RunRealTime() method is called on RealTimeExecutio when the task has finished, like a future.

### 2.2.4 State Diagram(s)

The AND-states can be seen in the figure below, which is taken from the assignment specification. This is the only change in the states from assignment 3 and therefore the only thing shown.

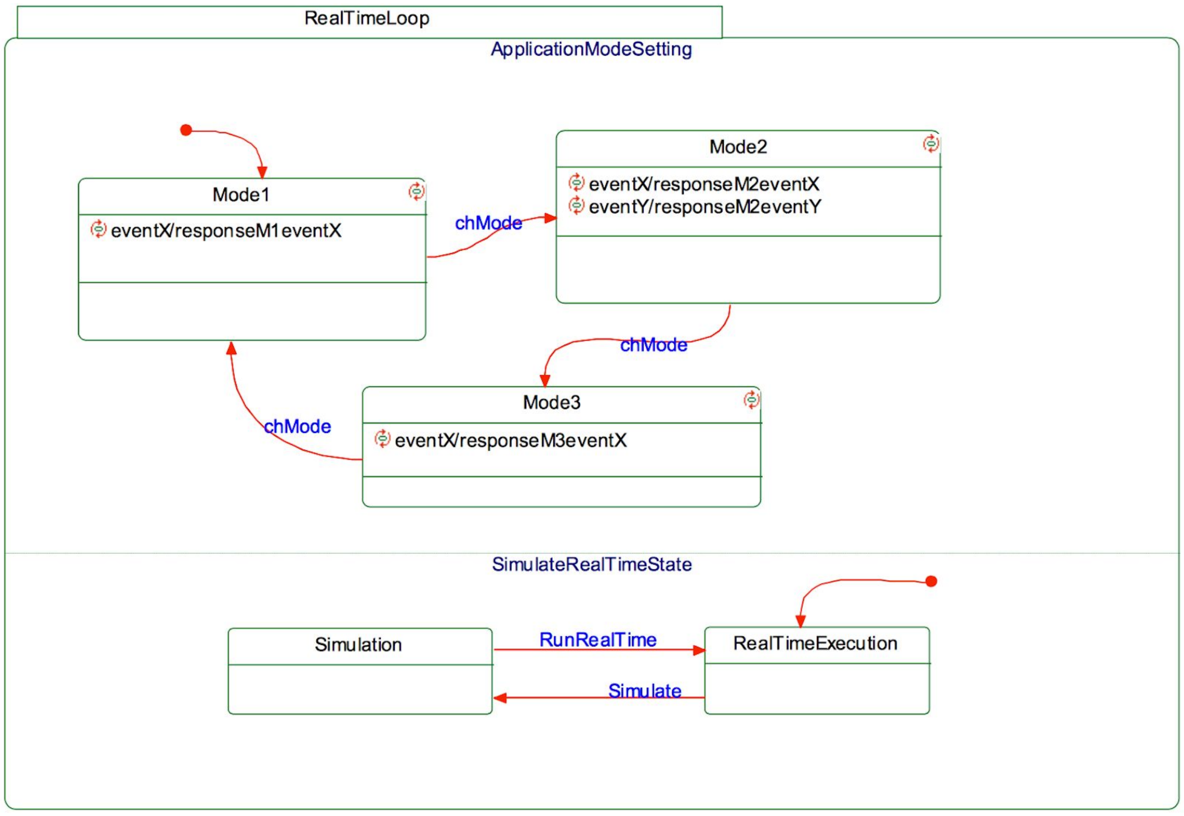

Figure 4. AND-State diagram [Assignment4.pdf]

## 2.3 Implementation View

### 2.3.1 Implementation details

The active object has been implemented such that there is a proxy that the client (RealTimeExecution) calls to push a simulation object to the queue. Then Simulation executes one simulation at a time. Interesting code snippets from our solution can be found below. The solution is implemented on the ZyBo board.

The client is left unchanged and does not need recompiling for assignment 4.

```
class EmbeddedSystemX
{
public:
    EmbeddedSystemX();
    ~EmbeddedSystemX();
    void SelftestOk(EmbeddedSystemX* context);
    void Initalized(EmbeddedSystemX* context);
    void Restart(EmbeddedSystemX* context);
    void Configure(EmbeddedSystemX* context);
    void ConfigurationEnded(EmbeddedSystemX* context);
    void Exit(EmbeddedSystemX* context);
    void Stop(EmbeddedSystemX* context);
    void Start(EmbeddedSystemX* context);
    void Suspend(EmbeddedSystemX* context);
    void SelfTestFailed(EmbeddedSystemX* context, int errorNo);
    void ConfigX(EmbeddedSystemX* context);
    void chMode(EmbeddedSystemX* context);
    void eventX(EmbeddedSystemX* context);
    void eventY(EmbeddedSystemX* context);
    void Resume(EmbeddedSystemX* context);
    void setCurrent(State *s);
private:
    int _versionNo;
    char* _name;
    // Pointer which holds the current state
    State* _currentState = PowerOnSelfTest::GetInstance();
};
```

Figure 5. Client EmbeddedSystemX.h

Our main.cpp defines two OSAPI threads with normal priority and starts the task scheduler on the ZyBo board. The MainThread runs the client in a thread and SimulationThread runs the active object (SimProxy). simCount is global variable we increment when a task has finished in the SimProxy.

Figure 6. Main.cpp

```
int simCount = 0;
SimProxy simProxy;

int main()
{
    // Threads
    MainThread mMainThread(Thread::PRIORITY_NORMAL, "MainThread");
    SimulationThread mSimThread(Thread::PRIORITY_NORMAL, "SimulationThread");

    /* Start FreeRTOS, the tasks running. */
    vTaskStartScheduler();

}

```

```
class MainThread : public Thread
{
public:

    MainThread(ThreadPriority pri, string name) :
         Thread(pri, name)
     {  }
    virtual void run()
    {
        printf("Booting ...\n");
        EmbeddedSystemX* context = new EmbeddedSystemX();
        context->SelfTestFailed(context, 99);
        context->Restart(context);
        context->SelftestOk(context);
        context->Initalized(context);
        context->Configure(context);
        context->ConfigurationEnded(context);
        context->Start(context);

        // Start simulation
        printf("Starting simulation from UserThread\n");
        context->Start(context);
        context->Start(context);
        context->Start(context);
        context->Start(context);
        context->Start(context);
        printf("Ending ...\n");
    }
};
```

Figure 7. UserThread.h

Figure 7 shows the UserThread class. This class is instantized and runs independently on the ZyBo board. We work our way through the various states until we reach the RealTimeLoop and we then create multiple tasks (Simulation) for the SimProxy to execute in a separate thread. Figure 8 shows the simulation thread that calls Execute() on the proxy. This method takes the front of the queue and executes that task.

Figure 8. SimulationThread.h

```
extern SimProxy simProxy;
class SimulationThread : public Thread {
public:

    SimulationThread(ThreadPriority pri, string name) :
        Thread(pri,name)
    {}

    virtual void run(){
        simProxy.Execute(this);}

};
```

Our new RealTimeLoop class now has derived from the superclass Operational as well as two new concurrent state instances (ApplicationModeSetting and RealTimeExecution) as shown on the state diagram.

```
extern int simCount;
class RealTimeLoop :
    public Operational
{
public:
    static RealTimeLoop* GetInstance();
    void Entry();
    void Exit(EmbeddedSystemX* context);
    void Restart(EmbeddedSystemX* context);
    void RunRealTime();
    void Simulate();
    void setCurrent(ApplicationModeSetting* newAppState);
    void SelftestOk(EmbeddedSystemX* context) override;
    void Initialized(EmbeddedSystemX* context) override;
    void Configure(EmbeddedSystemX* context) override;
    void ConfigurationEnded(EmbeddedSystemX* context) override;
    void Stop(EmbeddedSystemX* context) override;
    void Start(EmbeddedSystemX* context) override;
    void Suspend(EmbeddedSystemX* context) override;
    void Resume(EmbeddedSystemX* context) override;
    void SelfTestFailed(EmbeddedSystemX* context, int errorNo) override;
    void ConfigX(EmbeddedSystemX* context) override;
    void chMode(EmbeddedSystemX* context, int mode) override;
    void eventX(EmbeddedSystemX* context) override;
    void eventY(EmbeddedSystemX* context) override;
private:
    RealTimeLoop();
    ~RealTimeLoop();
    static ApplicationModeSetting* _appState;
    static RealTimeExecution* _simState;
    static RealTimeLoop* _instance;
```

Figure 9. RealTimeLoop.h

Interesting bits from the RealTimeLoop.cpp implementation is shown in figure 10. In Entry(), we set default instantiations of the two concurrent states. On Exit() we terminate the states. Start() orders the RealTimeExecution to create new Simulation objects and put them on the queue. Our chMode changes between modes on the ApplicationModeSetting based on argument.

```
ApplicationModeSetting* RealTimeLoop::_appState = 0;
RealTimeExecution* RealTimeLoop::_simState = 0;
RealTimeLoop* RealTimeLoop::_instance = 0;

void RealTimeLoop::Entry() {
    _appState = Mode1::GetInstance();
    _simState = new RealTimeExecution();
}

void RealTimeLoop::Exit(EmbeddedSystemX* context) {
    printf("Exit. Changing to Ready\n");
    _appState = NULL;
    _simState = NULL;
    Ready* state = Ready::GetInstance();
    context->setCurrent(state);
}

void RealTimeLoop::setCurrent(ApplicationModeSetting* newAppState) {
    _appState = newAppState;
}

void RealTimeLoop::Start(EmbeddedSystemX* context) {
    printf("Start called in RealTimeLoop");
    _simState->Start();
}

void RealTimeLoop::chMode(EmbeddedSystemX* context, int mode)  {
    // Change between the modes
    printf("In RealTimeLoop.chMode\n");
    if (mode = 1) _appState = Mode2::GetInstance();
    else if (mode = 2) _appState = Mode3::GetInstance();
    else if (mode = 3) _appState = Mode1::GetInstance();
    else _appState = Mode1::GetInstance();
}

void RealTimeLoop::eventX(EmbeddedSystemX* context) {
    _appState->responseEventX();
}

void RealTimeLoop::eventY(EmbeddedSystemX* context)
{
    _appState->responseEventY();
}
```

Figure 10. RealTimeLoop.cpp

Next is our SimProxy. Figure 11 shows the SimProxy.h. We define two queues, one for the commands and one for whoever client requested the command. The Push() method is called by RealTimeExecution with a Simulation object and a pointer to itself. Execute() runs a loop that takes commands from  the proxyQueue and executes them.

```
class SimProxy
{
public:
    SimProxy();
    ~SimProxy();
    void Push(Simulation* simObject, RealTimeExecution* rte);
    void Execute(AbstractOS::Thread* threadPtr);
private:
    std::queue<Simulation*> _proxyQueue;
    std::queue<RealTimeExecution*> _clientQueue;
    bool _executing = true;
    AbstractOS::Mutex simMutex;
};
```

Figure 11. SimProxy.h

Figure 12 shows the implementation of SimProxy. Execute() runs continuously and waits while the queue is empty. When new Simulation objects (commands) gets placed in the queue, we acquire the OSAPI mutex, take the front element of the queue as well as the front client, pop the elements and then calls RunSimulation() on the task. This logic is implemented in the Simulation class. The Push() method acquires the mutex in a similar way, pushes new commands onto the queue and releases the mutex afterwards.

Figure 12. SimProxy.cpp

```
SimProxy::SimProxy()
{
    simMutex = AbstractOS::Mutex("simMutex");
}

SimProxy::~SimProxy()
{
}

void SimProxy::Execute(AbstractOS::Thread* threadPtr)
{
    printf("SimProxy.Execute()\n");
    while(_executing)
    {
        while(_proxyQueue.empty()) {
            //threadPtr->yield();
        }
        simMutex.Acquire();
        printf("Now executing object\n");
        Simulation* task = _proxyQueue.front();
        RealTimeExecution* future = _clientQueue.front();
        _proxyQueue.pop();
        _clientQueue.pop();
        task->RunSimulation(future);
        delete task;
        delete future;
        simMutex.Release();
    }
}

void SimProxy::Push(Simulation* simObject, RealTimeExecution* rte)
{
    printf("In SimProxy. Now pushing objects\n");
    simMutex.Acquire();
    printf("Acquired mutex");
    _proxyQueue.push(simObject);
    _clientQueue.push(rte);
    simMutex.Release();
    printf("In SimProxy. Finished pushing objects\n");
}
```

Figure 13 is the implementation of the Simulation class. This class is created by RealTimeExecution, which we will see next. Simulation simply runs the simulation by incrementing the global variable simCount and invokes a callback onto the client, who requested it, when it is complemented.

Figure 13. Simulation.cpp
```cpp
Simulation::Simulation() { }

Simulation::~Simulation() { }

void Simulation::RunSimulation(RealTimeExecution* client)
{
    // Do simulation here
    // Return by calling future
    printf("In RunSimulation\n");
    simCount++;
    for (int i = 0; i < 100; i++) {};
        // sleep here
    client->RunRealTime();
}
```

Figure 14 shows the RealTimeExecution implementation, when the Simulation objects are created. One can view RealTimeExecution as the client to Simulation, because it receives the callback in RunRealTime().

Figure 14. RealTimeExecution.cpp
```cpp
void RealTimeExecution::RunRealTime()
{
    printf("Simulation finished, RealTimeExecution called");
}

RealTimeExecution::RealTimeExecution()
{
    printf("In ctor RealTimeExecution");
    // new proxy object
}

RealTimeExecution::~RealTimeExecution()
{
}

void RealTimeExecution::Start()
{
    printf("In RealTimeExecution.Start. Now pushing sim objects\n");
    simProxy.Push(new Simulation(), this);
}
```

## 3 Discussion of results

The concurrent part of this system gave some troubles when implementing on the ZYBO-board. This were due to the yielding of the tasks, but it were resolved by making freeRTOS handle it itself.

As seen below in figure 15 in the output from the ZYBO-board, the system works as expected. It behaves like in assignment3, going through the different states, but now it also shows, that RealTimeExecution pushes Simulation objects onto the SimProxy queue and then the SimProxy executes them individually and concurrently with the rest of the system.

Figure 15. Output from the ZyBo board, when we run our test with main.cpp.

```
Booting ...
SelfTestFailed. Changing to Failure
Restart. Changing to PowerOnSelfTest
SelfTestOk. Changing to Initializing
Initialized. Changing to Ready
Initialized. Changing to Ready
Configure. Changing to Configuration
Reading configuration...
ConfigurationEnded. Changing to Ready
Start. Changing to RealTimeLoop
In ctor RealTimeExecutionStarting simulation from UserThread
Start called in RealTimeLoopIn RealTimeExecution.Start. Now pushing sim objects
In SimProxy. Now pushing objects
Acquired mutexIn SimProxy. Finished pushing objects
Start called in RealTimeLoopIn RealTimeExecution.Start. Now pushing sim objects
In SimProxy. Now pushing objects
Acquired mutexIn SimProxy. Finished pushing objects
Start called in RealTimeLoopIn RealTimeExecution.Start. Now pushing sim objects
In SimProxy. Now pushing objects
Acquired mutexIn SimProxy. Finished pushing objects
Start called in RealTimeLoopIn RealTimeExecution.Start. Now pushing sim objects
In SimProxy. Now pushing objects
Acquired mutexIn SimProxy. Finished pushing objects
Start called in RealTimeLoopIn RealTimeExecution.Start. Now pushing sim objects
In SimProxy. Now pushing objects
Acquired mutexIn SimProxy. Finished pushing objects
Ending ...
SimProxy.Execute()
Now executing object
In RunSimulation
Simulation finished, RealTimeExecution called Now executing object
In RunSimulation
Simulation finished, RealTimeExecution called

```

## 4 Conclusion

In this journal we have shown the design and implementation of an AND-state machine using the active object pattern. Using this gives a concurrent execution of the client and the execution of the method.  Furthermore we have shown an implementation of this on the ZYBO-board.