



AARHUS
UNIVERSITY

EIC

Embedded Real-Time systems

Assignment 3 - Modeling and Implementation of a system behavior using UML patterns.

Name	Initials	student number
Jakob Bjerregaard Olsen	JBO	202109022
Alexander Nygaard Thomsen	AT	201911329
Wouter Oudijk	WO	202109059

Lecturer: Jalil Boudjadar, Kim Bjerge

Due date: 01/12/2025

1 Introduction

The main goal of this assignment is to implement a simulated version of an embedded system using state patterns, more specifically a concurrent state machine, where the handling of events are processed, by using *GOF State pattern*, *Singleton pattern* and the *Active object pattern*.

In this assignment, the application has been implemented by adhering to these requirements, and following the provided state diagram given with the assignment. Some liberty has been taken in terms of implementation, regarding how to interact with the system, but also how the states and classes have been used and implemented. The goal of this report is to explain the major outlines of the decisions made along, with visual descriptions and depictions. This includes large code snippets and UML diagrams which will serve as context.

Included with this report is the full code implementation, an executable file, all UML diagrams and code to recreate/recompile the diagrams.

1.1 tools used

To complete this assignment some tools have been used.

Tool	Description
VS Code	IDE, for programming the assignment
C++	the programming language required to implement the assignment and a compiler for C++14++ is required.
LaTeX	using MIKTEX
plantUML	UML tool of choice, converting it to a programming language, requires Java JDK
Git	for version control, using GITHUB to control the repo

Table 1: Tools used to complete the assignment

1.2 Goals

- **GOF Singleton pattern** To ensure the correct behavior of the statemachine, the Singleton pattern must be used, it ensures that only one instance of every object class can exist at any given time.
- **GOF state pattern** This pattern ensures that an object changes behaviour when it changes state, but to the outside world it looks like a new object. Instead of enumeration, the state and its behaviour is encapsulated in multiple classes, where our object can delegate to the current state class.
- **Active Object pattern** The main purpose of the active object pattern, is to separate method invocation from method execution, by decoupling. This lets us asynchronously invoke methods while not tying up valuable time to execute those methods, by having separate threads do both scheduling and execution.
 - Active object pattern must implement the provided interface
 - not be blocking
- **Follow the provided state pattern** With the assignment, the *EmbeddedSystemX* class was provided, which outlined the overall requirement for the application.

2 Solution

2.1 Overall architecture

- **EmbeddedSystemX** Is the overall class which implements all functionality of the application. Any user interaction is done through a simple text based user interface written in the main function.

To avoid unnecessary C++ semantics, the entire composition of EmbeddedSystemX and all its subcomponents is written in one large *cpp* file, this has mainly been done to reduce compiler complexity and practical implementation difficulties, which is bound to arise, due to things such as *circular-referencing*.

2.1.1 State Diagram

The overall structure follows the provided model of the system, seen on figure 1.

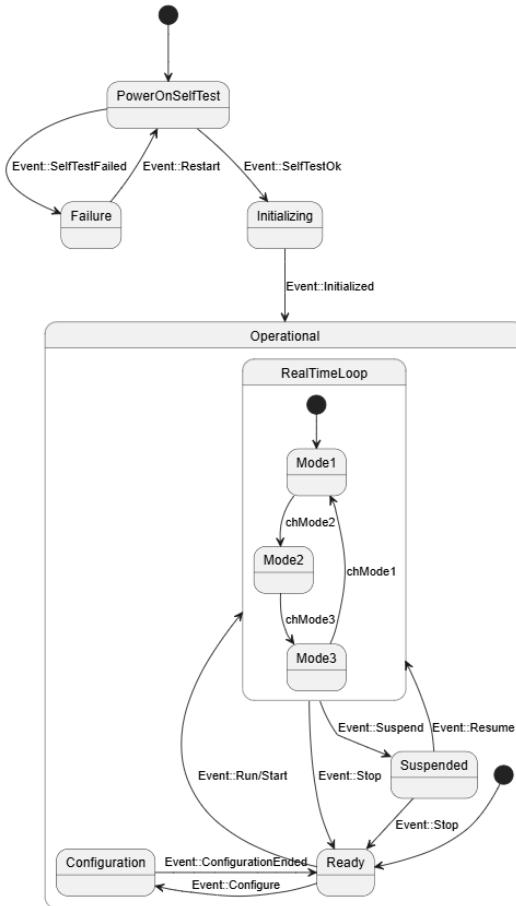


Figure 1: Overall statemachine of EmbeddedSystemX

There is a focus on the statemachine inside of the *Operational* state. This statemachine is referred to as *opFsm - operational state machine*.

2.1.2 Class Diagram

To conform with the requirement of the GOF state machine, all of the states are implemented as classes, which are inheriting from the *State*-class, see figure 2.

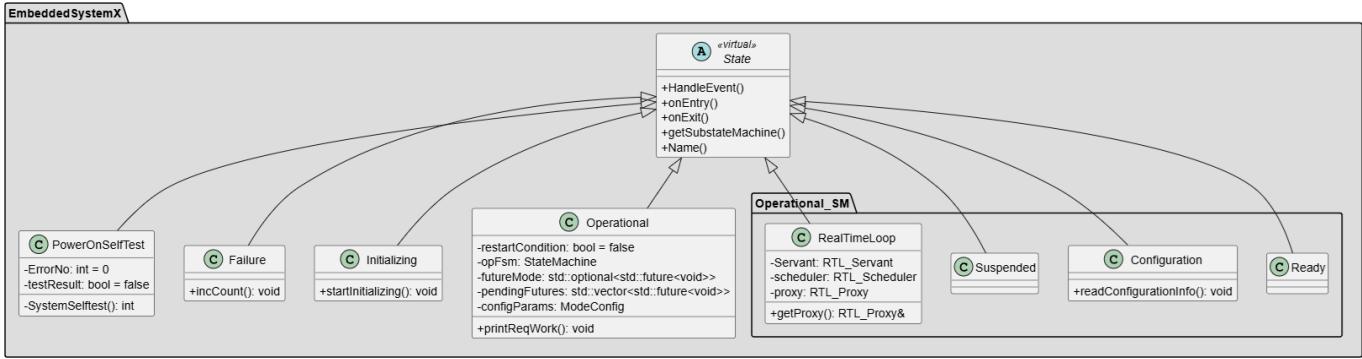


Figure 2: The State class, is a pure virtual class, which will be explained in a later section

The states are implemented this way, so that the statemachine trusts the state class with all the behavior of the implemented classes, this means that they all have the some core functions: `eventHandler`, `onEntry`, `onExit` and so on. It only fits together with the singleton pattern, which ensures that there will only ever be one of each state class and therefore stays consistent with previous interactions with that state class.

2.1.3 Sequence Diagram

The sequence of execution in the overall state machine is seen on figure 3.

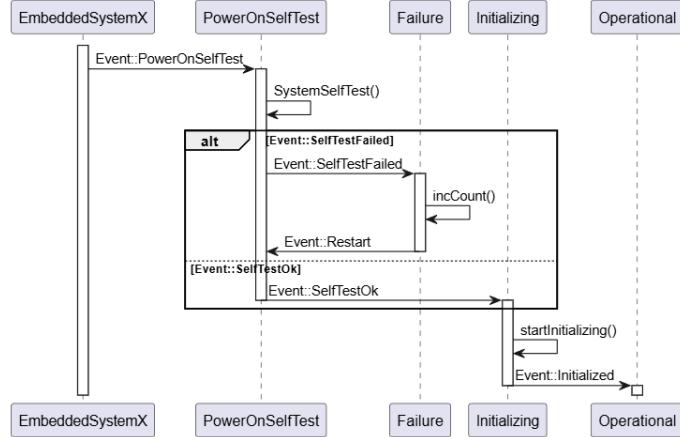


Figure 3: The initial phase of the EmbeddedSystemX's statemachine, here it automatically executes through the initial states. A small chance has been added so that the SelfTest might fail. Otherwise it justs continues to the ready state. It also shows the lifetime of the states as objects.

The operational class is where the internal statemachine gets activated by the *onEntry* function inside the operational class. That signifies the start of the operational sequence diagram, seen on figure 4

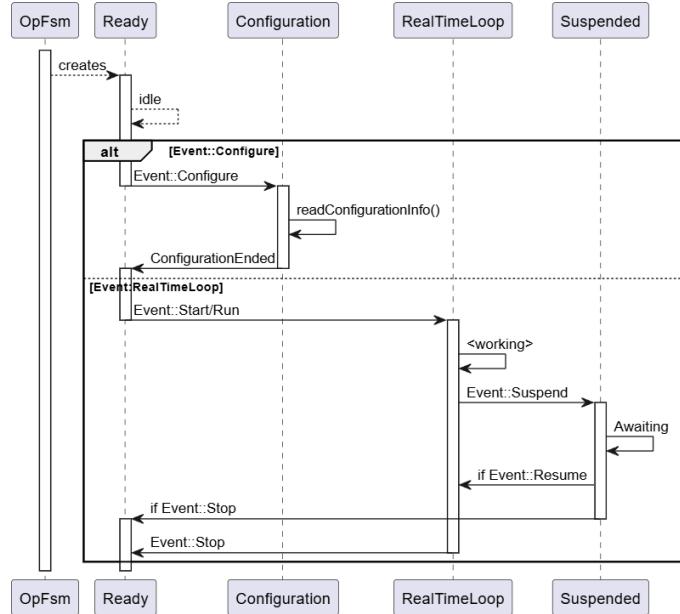


Figure 4: The internal statemachine *opFsm* inside operational, this is a concurrent statemachine to the one outside, however the two don't have any interplay. As with the outside statemachine, this also have classes as states, implemented by the state/eventHandler mechanism, all are singletons aswell. All data passed through opFsm, are referenced to the outside machine.

The important thing in this sequence diagram on figure 4 is, that the *RealTimeLoop* is an active object, meaning that its method invocation has been decoupled from its exection, this is also shown in figure 1. Where to overall mechanism follow the class diagram seen on figure 5.

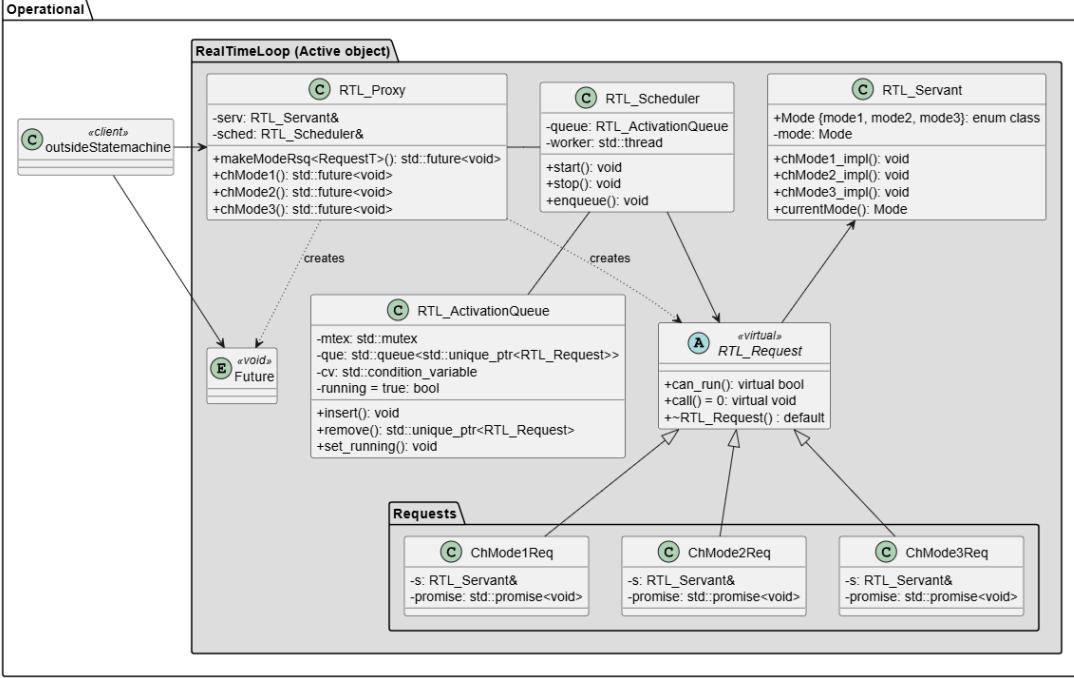


Figure 5: The internal class diagram of the active object, which resides inside the RealTimeLoop class

As already mentioned, the active object decouples the execution from the invocation, meaning that the caller or client is free to do whatever it wants in the meantime, so the call is non-blocking. This is done through the `std::future`, which is a pointer to a place where the work will, at some point, be completed. This is part of the future/promise functionality in C++. The overall idea is that it is up to the *Servant* to *fulfill the promise* of the future.

The sequence diagram for the internal statemachine *opFsm* can be found below

The essence of diagram shown in figure 6 is to show that there are two statemachines working at the same time, independently of each other, however, there are not conflicting states between them. The heavy lifting is done by the *handleEvent* function in operational, that passes any event that does not belong to the outer statemachine, to the inner statemachine.

The case in figure 6 is where the user wants to order work, ie. in this project, this is done by pseudo-work, were a enum state inside *RTLservant* is changed, according to the allowed states.

2.1.4 Active Object

The internal sequence diagram for RealTimeLoop is shown in figure 7, which is what happens when the diagram in figure 6 is executed. Thus the scheduler is "primed" in when entering the RealTimeLoop, however it does not do anything before the operational *handleEvent* is called to request the actual work.

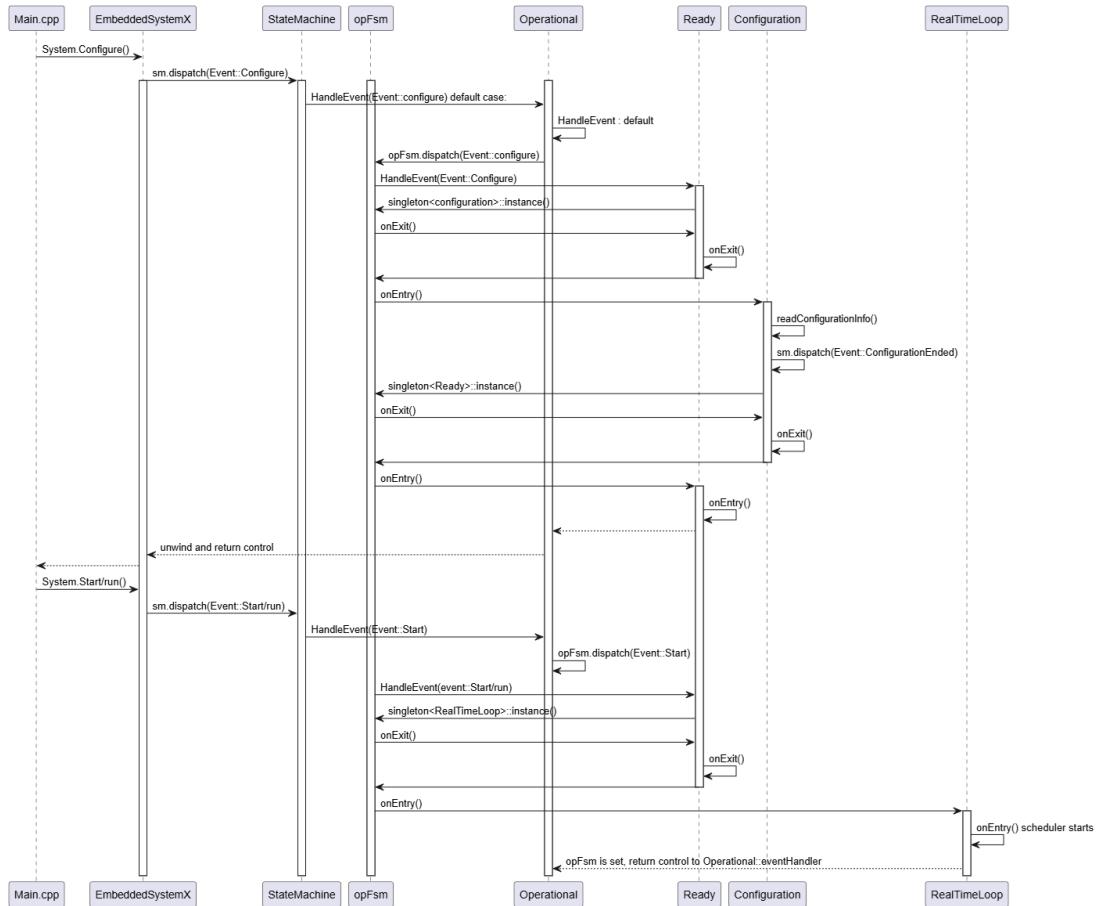


Figure 6: opFsm, statemachine sequence diagram.

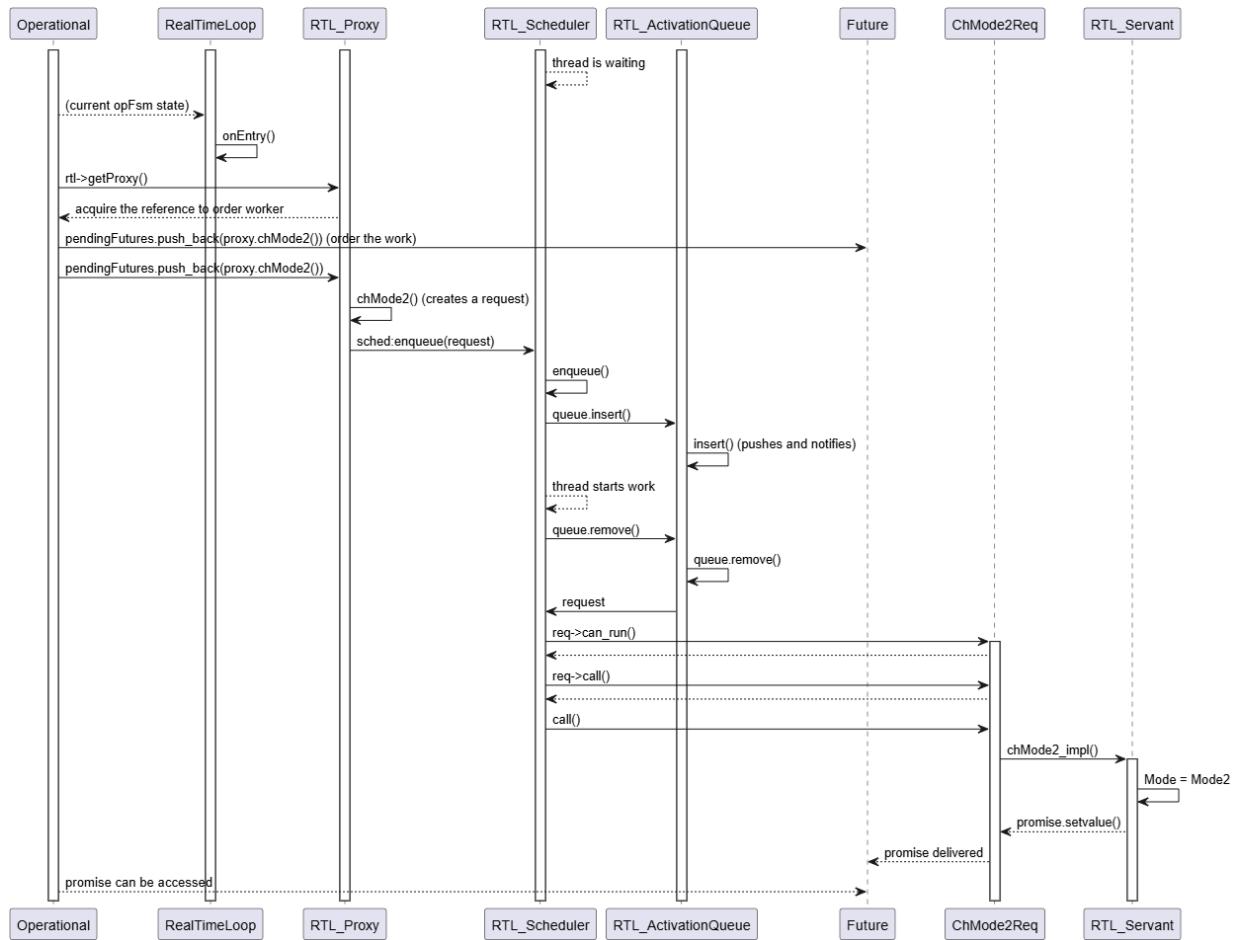


Figure 7: internal sequence diagram for RealTimeLoop

3 Implementation

As previously mentioned, to avoid C++ referencing issues, the entire implementation is made in one large file. In this section, some of the main takeaways from the implementation is highlighted.

3.1 Headers and Defines

Listing 1: include headers and custom defines

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <random>
4 #include <ctime>
5 #include <thread>
6 #include <chrono>
7 #include <future>
8 #include <condition_variable>
9 #include <mutex>
10 #include <queue>
11 #include <optional>
12 #include <vector>
13
14
15 #define TEST_MODE //comment out to disable test mode msg. Remember to build...
16 #ifdef TEST_MODE
17     #define TEST_PRINT(msg) \
18         do { std::cout << "[TEST]:" << msg << std::endl; }while(0)
19 #else
20     #define TEST_PRINT(msg) \
21         do {} while(0)
22 #endif
```

3.2 Singleton pattern

A main feature of the implementation was a singleton pattern, this means that the only one instance of every type that invokes this pattern can be created.

Listing 2: Singleton class

```
24 template<typename T>
25 class Singleton
26 {
27 public:
28     static T* Instance()
29     {
30         if(_instance == nullptr){
31             _instance = new T;
32         }
33         return _instance;
34     }
35 protected:
36     Singleton() = default;
37     ~Singleton() = default;
38
39     Singleton(const Singleton&) = delete;
40     Singleton& operator=(const Singleton&) = delete;
41
42 private:
```

```

43     static T* _instance;
44 };
45
46
47 template <typename T>
48 T* Singleton<T>::_instance = nullptr;

```

The templated type ensures that different classes can invoke the singleton pattern, because the destructor is protected, means that nothing can accidentally destroy the object. The idea is that the first time a state is required, it is constructed, and because the type is templated and made static, it is essentially remembered for the life of the program. Thus whenever a templated type is invoked, which already exists, the pointer to the existing instance is returned instead.

3.3 Event enum class

Listing 3: Event enum class

```

53 enum class Event
54 {
55     Initial,
56     SelfTestOk,
57     SelfTestFailed,
58     Error,
59     Initialized,
60     Restart,
61     Exit,
62     ready,
63     Configure,
64     ConfigurationEnded,
65     Stop,
66     Resume,
67     Start,
68     Run,
69     Suspend,
70     ChMode1,
71     ChMode2,
72     ChMode3,
73     WaitModeSwitch,
74     PrintReqQueue
75
76 };

```

The event enum class, acts a way to indicate what events are being passed around. This is combined with the handleevent function of every state, to control the statemachine.

3.4 State class

Listing 4: state class

```

92 class State
93 {
94 public:
95     virtual State* handleEvent(StateMachine& sm, Event e) = 0;
96     /*
97     after creation of object, state machine calls onEntry as overriden by children*/
98     virtual void onEntry(StateMachine& sm) {};
99
100    virtual void onExit(StateMachine& sm) {};

```

```

101
102     virtual StateMachine* getSubstateMachine() {return nullptr;}
103     virtual std::string name() const = 0;
104     virtual ~State() = default;
105
106 };

```

Pure virtual class State. Noteably the state class has a handleEvent function, whihc is simply = 0. This means that it must be overriden by any class that inherits State. The handleEvent and the two other functions, onEntry and onExit, make up the three functions that the statemachine uses to transfer control. It is also what enacts the GOF state pattern, because it creates a new class with defined functions when the states are changed.

3.5 Statemachine class

Listing 5: statemachine class

```

138 class StateMachine
139 {
140
141 private:
142     State* current;
143
144 public:
145     StateMachine() : current(nullptr) {}
146     //avoid implicit conversions. compiler would make State*->Statemachine
147     explicit StateMachine(State* initialState) : current(initialState)
148     {
149         if(current)
150             current->onEntry(*this); // the initial state at creation
151     }
152
153     void dispatch(Event e)
154     {
155         /*sequence explained in State.h*/
156         // std::cout << "[DISPATCH]:[" << e << "] " << std::endl;
157         TEST_PRINT("[DISPATCH]:[" << e << "]");
158         State* next = current->handleEvent(*this, e);
159         if (next != current)
160         {
161
162             //use of this
163             //this -> pointer to the object itself
164             ///*this -> reference to the statemachine
165             current->onExit(*this);
166             // delete current; //this might be a fix to the segmentation
167             // fault
168             current = next;
169             if (current)
170             {
171                 current->onEntry(*this);
172             }
173         } //a condition to check if next and current is the same can be set here.
174         // else the SM will set the end of execution here.
175         //execution delay here
176         // std::this_thread::sleep_for(std::chrono::milliseconds(1000));
177     }
178
179     State* getCurrentState() const{return current;}
180     std::string getStateName() const {return current ? current->name() : "<NULL>";}

```

```

180     std::string getFullStateName() const
181     {
182         std::string res = current ? current->name() : "<NULL>" ;
183         if(auto* sub = current->getSubstateMachine())
184         {
185             res += " -> " + sub->getFullStateName();
186         }
187         return res;
188     }
189 };

```

The statemachine class, is the main engine that drives the state changes. When instantiated, it sets the initialState and then drives the changes for everyclass. Due to the way the classes, or in this case, states are implemented, all states comply with the interface. All state classes makes use of dynamic binding, due to them having to override the functions to comply with the pure virtual class. Meaning that all states acts on the handleevent function etc. This is key to enacting the GOF state pattern.

Since all classes implement the same overall structure, only one will be shown here in the following section.

3.6 State example: Operational

Listing 6: Operational state class declaration

```

580 class Operational : public State
581 {
582     friend class Singleton<Operational>;
583 public:
584     void onEntry(StateMachine& sm) override;
585     void onExit(StateMachine& sm) override;
586     State* handleEvent(StateMachine& sm, Event e) override;
587     std::string name() const override {return "Operational";}
588
589     struct ModeConfig{
590         std::vector<RTL_Servant::Mode> req_work;
591     };
592     ModeConfig& getConfigParams() {return configParams;}
593     StateMachine* getSubstateMachine() override {return &opFsm;}
594     void printReqWork();
595 private:
596     Operational() = default;
597     ~Operational() override = default;
598
599     Operational(const Operational&) = delete;
600     Operational& operator=(const Operational&) = delete;
601
602     bool restartCondition = false;
603     StateMachine opFsm; //internal FSM, is invoked when initialized event is called
604     std::optional<std::future<void>> futureMode;
605     std::vector<std::future<void>> pendingFutures;
606     ModeConfig configParams;
607 };

```

Listing 7: Operational state class definition, partly

```

780 //OPERATIONAL-----
781 void Operational::onEntry(StateMachine& sm)
782 {
783     // std::cout << "[FUNC]:[onENTRY]:Operational" << std::endl;
784     TEST_PRINT("[FUNC]:[onENTRY]:Operational");
785     if(restartCondition)
786     {
787         sm.dispatch(Event::Restart);
788     }
789     /*SPIN UP INTERNAL FSM*/
790     //BE WARNED, FSM TRANSITION IS HAPPENING AUTOMATICALLY.
791     opFsm = StateMachine(Singleton<Ready>::Instance());
792 }
793
794 void Operational::onExit(StateMachine& sm)
795 {
796     // std::cout << "[FUNC]:[onEXIT]:Operational" << std::endl;
797     TEST_PRINT("[FUNC]:[onEXIT]:Operational");
798 }
799 State* Operational::handleEvent(StateMachine& sm, Event e)
800 {
801     switch (e)
802     {
803
804     case Event::Restart:
805         return Singleton<PowerOnSelfTest>::Instance();
806         break;
807
808     //these requests might not even work. - should probably be deleted.
809     case Event::ChMode1:
810     case Event::ChMode2:
811     case Event::ChMode3:
812     {
813         State* curr = opFsm.getCurrentState();
814         //dynamic_cast<> runtime type check
815         //is the actual object point to by curr of type RealTimeLoop or a derivative
816         //thereof
817         //returns a RealTimeLoop* if true else nullptr
818         if(auto* rtl = dynamic_cast<RealTimeLoop*>(curr))
819         {
820             RTL_Proxy& proxy = rtl->getProxy();
821
822             if(e == Event::ChMode1)
823                 futureMode = proxy.chMode1();
824             if(e == Event::ChMode2)
825                 futureMode = proxy.chMode2();
826             if(e == Event::ChMode3)
827                 futureMode = proxy.chMode3();
828         }else{
829             std::cout << "[OPERATIONAL]:EVENT_MODE_IGNORED" << std::endl;
830         }
831         return this;
832     }
833     case Event::WaitModeSwitch:
834     {
835         if(futureMode.has_value())
836         {
837             std::cout << "[OPERATIONAL]:reading_future:" << std::endl;

```

```

837         futureMode->get();
838         std::cout << "[OPERATIONAL]: mode_change_complete" << std::endl;
839         futureMode.reset();
840     }else{
841         std::cout << "[OPERATIONAL]: NO_FUTURE_TO_BEHOLD!" << std::endl;
842     }
843     return this; //were not changing state, only reading the future, like a freaking
844     seer
845 }
846 case Event::Start:
847 {
848     //transition of inner statemachine to RTL, created in READY eventhandler
849     opFsm.dispatch(Event::Run);
850     //ensure we are in RTL
851     State* curr = opFsm.getCurrentState();
852     if(auto* rtl = dynamic_cast<RealTimeLoop*>(curr))
853     {
854         RTL_Proxy& proxy = rtl->getProxy();
855         for(auto mode : configParams.req_work)
856         {
857             switch (mode)
858             {
859                 case RTL_Servant::Mode::Mode1:
860                     pendingFutures.push_back(proxy.chMode1());
861                     break;
862                 case RTL_Servant::Mode::Mode2:
863                     pendingFutures.push_back(proxy.chMode2());
864                     break;
865                 case RTL_Servant::Mode::Mode3:
866                     pendingFutures.push_back(proxy.chMode3());
867                     break;
868             }
869         }
870         configParams.req_work.clear(); //empties the vector
871     }
872     return this;
873 }
874
875 case Event::PrintReqQueue:
876 {
877     printReqWork();
878     return this;
879 }
880 //this is doing the heavy lifting
881 default:
882     opFsm.dispatch(e); //any changes to inner statemachine is done through this
883     dispatch!
884     return this;
885 }
886 }
```

The Operational class, which acts as the main driving state for the outer statemachine, aswell as the "owner" of the internal statemachine, refered to as "opFsm". The handleEvent function does most of the heavy lifting, in terms of functionality. Specifically the case "Event::start", which firstly dispatches the event run, to the internal statemachine, setting the stage for the scheduler to run. Afterwards its ensures that the state of the internal machine is RealTimeLoop, which means it can then use the proxy which is owned by that state. From here the pendingFutures, which has been made using the configuration state, can be requested using a templated requesting function.

3.7 EmbeddedSystemX implementation

Listing 8: EmbeddedSystemX

```

1128 class EmbeddedSystemX
1129 {
1130     public:
1131         EmbeddedSystemX() : sm(Singleton<PowerOnSelfTest>::Instance()){}
1132
1133
1134     //public interfaces
1135
1136
1137     void SelfTestOk()                                {sm.dispatch(Event::SelfTestOk);}
1138     void SelftestFailed()                            {sm.dispatch(Event::SelfTestFailed);}
1139     void Error()                                    {sm.dispatch(Event::Error);}
1140     void Initialized()                             {sm.dispatch(Event::Initialized);}
1141     void Restart()                                 {sm.dispatch(Event::Restart);}
1142     void Exit()                                    {sm.dispatch(Event::Exit);}
1143     void ready()                                   {sm.dispatch(Event::ready);}
1144     void Configure()                               {sm.dispatch(Event::Configure);}
1145     void ConfigurationEnded()                     {sm.dispatch(Event::ConfigurationEnded);}
1146     void Stop()                                    {sm.dispatch(Event::Stop);}
1147     void Resume()                                 {sm.dispatch(Event::Resume);}
1148     void Start()                                   {sm.dispatch(Event::Start);}
1149     void Run()                                     {sm.dispatch(Event::Run);}
1150     void Suspend()                                {sm.dispatch(Event::Suspend);}
1151     void printReqQueue()                           {sm.dispatch(Event::PrintReqQueue);}
1152
1153     void printCurrentState()                      {std::cout << "current state:" << sm << std::endl;}
1154
1155 private:
1156     StateMachine sm;
1157     // int VersionNO = 43
1158     // std::string = "Ver1.1A"
1159 };

```

The EmbeddedSystemX class is the one that holds the entire implementation. All the possible events are defined here and the statemachine is initialized by creating the first state, which in this case is the PowerOnSelfTest state. It is also through this class, that the user can interact with the statemachine.

3.8 Active object

A big focus point of this assignment, is to implement the RealTimeLoop state as an active object. What this means, is that the internal states of the class, has to be requested and cannot be guaranteed to be executed immidiately, also the main feature of the active object, is to separate method invocation from execution.

Listing 9: Active Object

```

243 class RTL_Servant {
244 public:
245     enum class Mode {Mode1, Mode2, Mode3};
246     RTL_Servant() : mode(Mode::Mode1) /*sets initial mode to mode 1*/
247
248     //---> this is where ill do the actual work, unwind and return to where this function
249     //      is called from
250     void chMode1_impl() {
251         mode = Mode::Mode1;
252         std::cout << "[RTL_servant]:switched to mode 1" << std::endl;

```

```

252     }
253     void chMode2_impl() {
254         mode = Mode::Mode2;
255         std::cout << "[RTL_servant]:switched_to_mode_2" << std::endl;
256     }
257     void chMode3_impl() {
258         mode = Mode::Mode3;
259         std::cout << "[RTL_servant]:switched_to_mode_3" << std::endl;
260     }
261
262     Mode currentMode() const {return mode;}
263
264 private:
265     Mode mode;
266 };
267
268 class RTL_Request
269 {
270 public:
271     /*
272     there is no dependencies, no preconditions, nothing is holding it back
273     thus execution be done immidiately, FIFO style.
274     */
275     virtual bool can_run() const {return true;}
276     virtual void call() = 0; //pure virual, must be overridden
277     virtual ~RTL_Request() = default;
278 };
279
280 class ChMode1Req : public RTL_Request{
281 public:
282     ChMode1Req(RTL_Servant& s, std::promise<void> p)
283         : s(s), promise(std::move(p))
284     {}
285
286     void call() override {    //--> this is where i call the servant to do the work -->(
287         goto RTL_servant)
288         std::this_thread::sleep_for(std::chrono::seconds(2+(std::rand() % 9)));
289         s.chMode1_impl();
290         //though the promise is <void> it still sets the shared "thread" state to "ready"
291         //--> i've returned to say i've fullfilled the promise
292         promise.set_value();
293     }
294     bool can_run() const override {
295         //check the servant to see if the current mode is Mode3, if true then this event
296         //is allowed to run
297         return s.currentMode() == RTL_Servant::Mode::Mode3;
298     }
299
300 private:
301     RTL_Servant& s;
302     std::promise<void> promise;
303 };
304
305 class ChMode2Req : public RTL_Request{
306 public:
307     ChMode2Req(RTL_Servant& s, std::promise<void> p)
308         : s(s), promise(std::move(p))
309     {}

```

```

309     void call() override {
310         std::this_thread::sleep_for(std::chrono::seconds(2+(std::rand() % 9)));
311         s.chMode2_impl();
312         promise.set_value();
313     }
314     bool can_run() const override {
315         //same logic, but for new mode
316         return s.currentMode() == RTL_Servant::Mode::Mode1;
317     }
318
319 private:
320     RTL_Servant& s;
321     std::promise<void> promise;
322 };
323
324 class ChMode3Req : public RTL_Request{
325 public:
326     ChMode3Req(RTL_Servant& s, std::promise<void> p)
327         : s(s), promise(std::move(p)) /*use of move semantics to transfer pointer*/
328     {}
329
330     void call() override {
331         std::this_thread::sleep_for(std::chrono::seconds(2+(std::rand() % 9)));
332         s.chMode3_impl();
333         promise.set_value();
334     }
335     bool can_run() const override {
336         return s.currentMode() == RTL_Servant::Mode::Mode2;
337     }
338
339 private:
340     RTL_Servant& s;
341     std::promise<void> promise;
342 };
343
344 class RTL_ActivationQueue
345 {
346 public:
347     void insert(std::unique_ptr<RTL_Request> req)
348     {
349         std::lock_guard<std::mutex> lock(mtex);
350         que.push(std::move(req));
351         cv.notify_one();
352     }
353     //this might be overkill
354     std::unique_ptr<RTL_Request> remove()
355     {
356         //create unique pointer of RTL_requests
357         std::unique_ptr<RTL_Request> req;
358
359         //this might be super overkill
360         //acquire the lock so the que can be used safely, no one else can push/pop
361         std::unique_lock<std::mutex> lock(mtex);
362
363         //make the thread sleep until either request is available or system is stopping
364         //capture everything by reference, q and running
365         cv.wait(lock, [&]{return !que.empty() || !running; });
366         //if queue is not empty. undefined behaviour if empty queue is pop'd
367         if(!que.empty())

```

```

368     {
369         //take ownership of the first object in the queue
370         req = std::move(que.front());
371         //and pop it, remove the first element;
372         que.pop();
373     }
374     return req;
375 }
376
377 void set_running(bool status)
378 {
379     //take the lock and lock it now, then automatically unlock it when the scope ends
380     /*
381     we create a temporary object named lock, that takes types of std::mutex
382     the invoked constructor locks the mutex
383     when the lock goes out of scope (it leaves this function body) the destructor
384     is automatically invoked and the resources is released. */
385     std::lock_guard<std::mutex> lock(mtex);
386     //indicate we are working
387     running = status;
388     cv.notify_all();
389 }
390
391 private:
392     std::mutex mtex;
393     std::queue<std::unique_ptr<RTL_Request>> que; //queue of type unique pointers of type
394     RTL requests
395     std::condition_variable cv; //avoid needless polling, uses notify one on the mutex to
396     signal when ready
397     bool running = true;
398 };
399
400 class RTL_Scheduler{
401 public:
402     RTL_Scheduler() = default;
403
404     void start()
405     {
406         /*
407         start the queue
408         set_running contains a lock that acquires the mutex
409         "i own the mutex now, no other thread may pass"
410         */
411         queue.set_running(true);
412         //-->spin up the threads with work from the queue
413         /*
414         lambda invocation
415         [this] gives the lambda access to fields/functions from RTL_ActivationQueue
416         the while, makes the thread execute as long as the object is alive. */
417         worker = std::thread([this](){
418             while(true)
419             {
420                 //-->take the request from the queue
421                 auto req = queue.remove();
422                 if(!req) //if the queue is empty
423                 {
424                     break;
425                 }
426                 if(req->can_run()) //-->if the request can be ran, ie. work can be done

```

```

425
426
427
428
429
430
431
432
433 void stop()
434 {
435     //set the running flag to false
436     queue.set_running(false);
437     // join all threads if they are done working
438     if(worker.joinable())
439     {
440         //joins the thread, terminates the processing.
441         worker.join();
442     }
443 }
444 void enqueue(std::unique_ptr<RTL_Request> req)
445 {
446     //add/move work to the queue using insert.
447     queue.insert(std::move(req));
448 }
449
450 private:
451     RTL_ActivationQueue queue;
452     std::thread worker;
453 };
454
455
456 class RTL_Proxy{
457 public:
458     // RTL_Proxy() = default;
459     RTL_Proxy(
460         RTL_Servant& servant,
461         RTL_Scheduler& scheduler)
462         : serv(servant), sched(scheduler) {}
463
464     //templated type
465     template<typename RequestT>
466     std::future<void> makeModeRsq() { //function returns a std::future pointer
467         std::promise<void> prom; //declare a std::promise
468         auto fut = prom.get_future(); //bridge the future to the promise
469         /* -->
470          in one foul swoop, allocate memory for RequestT type, with given arguments
471          and wraps the object in a std::unique_ptr, without the use of new.
472          In other words:
473          A promise is created
474          A future is extracted
475          A request object is created --> this object determines
476          - which servant to call (serv)
477          - how to call it (through the future)
478          - the promise of the work
479
480          this is then sent to the queue by scheduler.
481         */
482         auto req = std::make_unique<RequestT>(serv, std::move(prom));
483         //queue the request through the scheduler.

```

```

484     sched.enqueue(std::move(req)); //--> scheduler thread is the run from RealTimeLoop
        ::onEntry --> (goto)
485
486     return fut;
487 }
488
489 //use template to create functions
490 std::future<void> chMode1() { return makeModeRsq<ChMode1Req>(); } // --> invokes the
    makeModeRsq function and specifies what request -->(goto)
491 std::future<void> chMode2() { return makeModeRsq<ChMode2Req>(); }
492 std::future<void> chMode3() { return makeModeRsq<ChMode3Req>(); }
493
494
495
496 private:
497     RTL_Servant& serv;
498     RTL_Scheduler& sched;
499 };

```

The listing above is the entire code for using the active object implementation. This implementation is an attempt at following the active object structure with the commands pattern. The state RealTimeLoop owns the servant, scheduler and the proxy. The scheduler itself owns the ActivationQueue and all the requests are a inherited request type. Following the sequence diagram on figure 7. A precondition to this is that some work has been requested through the configuration state, which is then stored in the configPrams memberfield of the operational class. Issuing the event "start" from main, starts the scheduler, which will then wait until it gets a signal from the ActivationQueue. After this, the handleevent will invoke the proxy to request the work from the list. Doing this makes use of the .enqueue() function, will then inserts them into the ActivationQueue. This actions notifies the waiting thread, that there is work to be done. This newly spawned thread will then take the work from the queue using .remove() and it will spawn a request. Using polymorphism, it will then check if this work can be ran at all, and afterwards invoke the .call() function. This invokes the function from the references servant and the mode is changed. This also fulfills the promise and the proxy is delivered the future. Although it is a void type future, it still signals that the promise is delivered.

4 Discussion

Looking at the requirements for the overall system, the final implementaion is a close attempt, however minor functionality are still missing, specifically the suspend mechanism, which should halt the thread execution.

Looking at the Active object implementation, because the system is dealing with almost instantenously executed code, there is no real consequence of execution time, which is why an artificial thread sleep is introduced. This is also why the all the futures used in the implementation are void. They are merely used to indicate that some work has been done, however they don't return anything. And all state processing is being held in the Servant.

As to whether the desired implementation lives up to the provided diagram is up for discussion, some of the descriptors, specifically "configX", "eventX/responseM n eventX" seemed a little vague, so some artistic liberty was necessary to create a satisfactory solution.

5 Conclusion

It can then be concluded that the group has made a valid implementation of the requested system. All critical functionality is preset, however some minor functions are still missing, such as the Suspend state in the Operational statemachine. As mentioned some artistic liberty has been taken due to vague assignment description, which in turn, has led to decisions like using void futures, as the provided state diagram was up to interpretation.