Table of Contents

Introduction

The framework is formed by a number of C++ classes that implement key features of the RoboChart language. This manual describes how the provided C++ classes are used to build a simulation of a RoboChart model. Currently, the framework supports a limited subset of the RoboChart notation characterised as follows:

- There is a single robotic platform;
- There is a single controller;
- There is a single state machine;
- Entry, exit and transition actions terminate and fit within a single step of the simulation

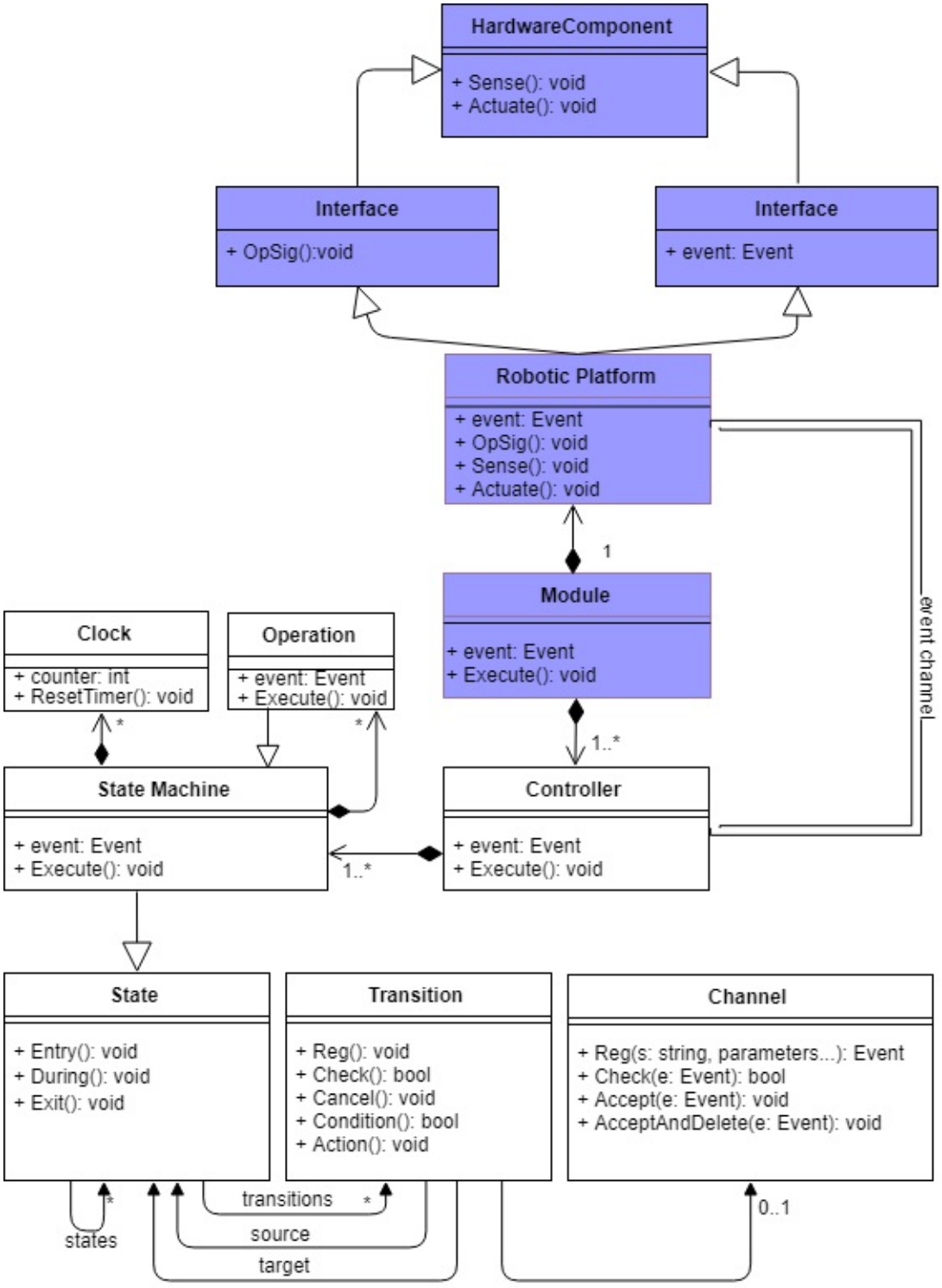
Despite the constrains of the current simulation framework, it can still be used for modelling a wide variety of robotic controllers. In the next chapter, we demonstrate how these classes are used to construct a simulation of a simple RoboChart model. The final chapter provides more details about the framework classes.

One special aspect that is not enforced by the framework, but must be considered when implementing simulations is the usage of constants, given types (primitive types) and simulation specific parameters. Global constants of a model and simulation parameters such as `tstep` (time step) can be implemented as class variables and initialised from an external source to avoid hard-coding and the necessity of recompilation when changes are necessary.

In our simulation targeting the ARGoS platform, these constants and parameters are implemented as variables of the controller class initialised by the module class based on values read from the XML configuration file. This approach relies heavily on ARGoS specific features, and might require some changes in different platforms, but the implementation of constants and parameters as variables in the appropriate classes should be feasible in different platforms.

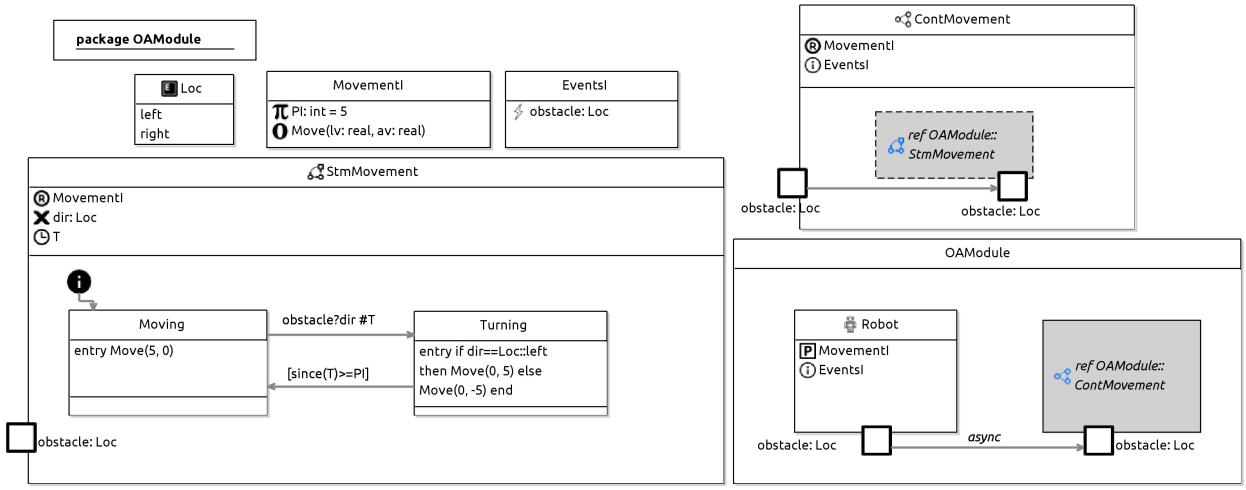
The treatment of given types (primitive types) is slightly less clear because ARGoS have limited configuration mechanisms to define (reinterpret) types. The types should probably be hard-coded using C++ type aliasing mechanisms such as `typedef` . For example, a type `ID` could be declared in the simulation code as `typedef int ID` .

The architecture of a simulation is as shown in the picture below, where we consider an example described in the next section as well the general classes of the framework.



The entry point of the simulation is the execute method of the `Module` class. The blue boxes correspond to framework classes that must be extended to create a simulation. The methods shown in the classes can be implemented in the subclass to provide specific behaviours such as the entry action of a state.

Framework Usage



A package called OAModule is created. This package defines a module `OAModule,` and the module includes a robotic platform `Robot` and a controller `ContMovement` . The robot provides one interface `MovementI` and uses one interface `EventsI.` The first interface groups the operation `Move` and a const variable `PI` , and the second interface groups event `obstacle` . The robotic platform is composed of a controller `ContMovement` that contains the state machine `StmMovement` . The state machine declares a variable `dir` and the clock `T` . It consists of two states (`Moving` and `Turning`), and the first uses the operation to move the robot forward. If an obstacle is found, the clock is reset and the state `Turning` is entered. In the `Turning` state, the robot starts turning based on the location of the obstacle, and if the time passed a certain threshold PI, the robot enters the `Moving` state again.

Module

The Module class instantiates the robot, the controller and all the channels used in the model.

```
//OAModule.h
class OAModule {
public:
    OAModule() :
            OAModule_Robot(nullptr),
            OAModule_ContMovement(nullptr),
            obstacle(std::make_shared<robochart::obstacle_channe
l>("obstacle")) {}
    virtual ~OAModule() {};

    void Init();
    void Execute();

private:
    std::shared_ptr<robochart::obstacle_channel> obstacle;
    std::shared_ptr<Robot> OAModule_Robot;
    std::shared_ptr<ContMovement> OAModule_ContMovement;
};
```

It also implements the functions `Init` and `Execute`. Init function instantiates the `OAModule_Robot` and `OAModule_ContMovement` class. This function can also be extended to pass initial parameters to the robot and controller. In ARGoS, this is done via reading a XML file using the Init function. The function `Execute` executes the sensors of the robot, the behaviour of the controller and then executes the actuators of the robot. This function can be executed in the step function of a particular simulator. In ARGoS, the step function is called `ControlStep()`.

RobotHardware

The `HardwareComponent` class is a abstract class that include two virtual functions `Sense()` and `Actuate()`. These two functions need to be extended.

```
//HardwareComponent.h
class HardwareComponent {

public:
    HardwareComponent() {}
    virtual ~HardwareComponent() {}
    virtual void Sense() = 0;
    virtual void Actuate() = 0;
};
```

Interface

Each of the interfaces is declared as a class that implements `HardwareComponent`.

```
//EventsI.h
class EventsI : public HardwareComponent {
public:
    EventsI(std::shared_ptr<robochart::obstacle_channel> obstacl
e);
    virtual ~EventsI();

    virtual void Sense();
    virtual void Actuate();


private:
    std::shared_ptr<robochart::obstacle_channel> obstacle;

};
```

```
//MovementI.h
class MovementI : public HardwareComponent {
public:
    MovementI();
    virtual ~MovementI();
    void Move(double lv, double av);

    virtual void Sense();
    virtual void Actuate();

public:
    int PI;


};
```

The `Move()` function provided by the interface sets the desired linear and angular speed of the robot. In the `Actuate()` function, the desired linear and angular speed is used to calculate the required velocity of the left and right wheels to activate the motor. The class `MovementI` or `EventsI` requires the implementation of the functions `Sense()` and `Actuate()`. This is the wrapper that maps the notions of RoboChart to the native simulation.

Robot

The robot inherits two interfaces: `MovementI` and `EventsI` that provide the operation `Move` and event in the channel `obstacle`, respectively.

```
//Robot.h
class Robot: public MovementI, public EventsI {
public:
    Robot(
        std::shared_ptr<robochart::obstacle_channel> obstacle);
    virtual ~Robot();

    void Sense();
    void Actuate();

private:
    std::shared_ptr<robochart::obstacle_channel> obstacle;

};
```

The constructor instantiates each of the interfaces, and the functions `Sense()` and `Actuate()` execute the respective functions of each of the interfaces.

Controller

A controller class `ContMovement` implements a generic Controller class that contains a reference to a state machine.

```
//ContMovement.h
class ContMovement: public robochart::Controller {
public:
    ContMovement(
            std::shared_ptr<Robot> R_Robot,
            std::shared_ptr<robochart::obstacle_channel> obstacl
e);
    virtual ~ContMovement();
    virtual void Execute();

private:
    std::shared_ptr<robochart::obstacle_channel> obstacle;
private:
    std::shared_ptr<Robot> R_Robot;

};
```

The `ContMovement` controller receives pointers to the robot, and one channel `obstacle`. The `stm` field needs to be set by the module in the `Init()` function of the module class to avoid issues with circular dependencies.

State Machine

A state machine is a class that inherits from an abstract class `StateMachine`, instantiates its substates and transitions and links them together. The classes that implement the states and transitions of the machine are presented in the next section.

```
//StmMovement.h
#define SM_DEBUG

class StmMovement: public robochart::StateMachine
{
public:
    std::shared_ptr<robochart::obstacle_channel> obstacle;
public:
    std::shared_ptr<robochart::Timer> T;
```

```
    std::shared_ptr<Robot> R_Robot;
    std::shared_ptr<ContMovement> C_ContMovement;
public:
    Loc dir;
public:
    StmMovement(
            std::shared_ptr<Robot> R_Robot,
            std::shared_ptr<ContMovement> C_ContMovement,
            std::shared_ptr<robochart::obstacle_channel> obstacl
e);
    ~StmMovement();
    int Initial();
    virtual void Execute();

public:
    class Moving : public robochart::State
    {
        public:
            Moving(std::shared_ptr<Robot> R_Robot, std::shared_p
tr<ContMovement> C_ContMovement, std::shared_ptr<StmMovement> S_
StmMovement) : State("Moving"), R_Robot(R_Robot), C_ContMovement
(C_ContMovement), S_StmMovement(S_StmMovement)
            {
            }
            void Entry()
            {
                R_Robot->Move(5, 0);
            }
        private:
            std::shared_ptr<Robot> R_Robot;
            std::shared_ptr<ContMovement> C_ContMovement;
            std::shared_ptr<StmMovement> S_StmMovement;
    };
    class Turning : public robochart::State
    {
        public:
            Turning(std::shared_ptr<Robot> R_Robot, std::shared_
ptr<ContMovement> C_ContMovement, std::shared_ptr<StmMovement> S
_StmMovement) : State("Turning"), R_Robot(R_Robot), C_ContMoveme
nt(C_ContMovement), S_StmMovement(S_StmMovement)
```

```
            {
            }
            void Entry()
            {
                if (S_StmMovement->dir == Loc::left)
                {
                    R_Robot->Move(0, 5);
                }
                else
                {
                    R_Robot->Move(0, -5);
                }
            }
        private:
            std::shared_ptr<Robot> R_Robot;
            std::shared_ptr<ContMovement> C_ContMovement;
            std::shared_ptr<StmMovement> S_StmMovement;
    };
    class i0 : public robochart::State
    {
        public:
            i0(std::shared_ptr<Robot> R_Robot, std::shared_ptr<C
ontMovement> C_ContMovement, std::shared_ptr<StmMovement> S_StmM
ovement) : State("i0"), R_Robot(R_Robot), C_ContMovement(C_ContM
ovement), S_StmMovement(S_StmMovement)
            {
            }
        private:
            std::shared_ptr<Robot> R_Robot;
            std::shared_ptr<ContMovement> C_ContMovement;
            std::shared_ptr<StmMovement> S_StmMovement;
    };

public:
    class t1 : public robochart::Transition {
        private:
            std::shared_ptr<Robot> R_Robot;
            std::shared_ptr<ContMovement> C_ContMovement;
            std::shared_ptr<StmMovement> S_StmMovement;
            std::shared_ptr<robochart::obstacle_event> reg_obsta
```

```
cle_event;
        public:
            t1(std::shared_ptr<Robot> R_Robot, std::shared_ptr<C
ontMovement> C_ContMovement, std::shared_ptr<StmMovement> S_StmM
ovement, std::weak_ptr<robochart::State> src, std::weak_ptr<robo
chart::State> tgt):
                robochart::Transition("S_StmMovement_t1", src, tg
t), R_Robot(R_Robot), C_ContMovement(C_ContMovement), S_StmMovem
ent(S_StmMovement), reg_obstacle_event(nullptr)
            {}
            void Reg() {
                if (reg_obstacle_event == nullptr) {
                    reg_obstacle_event = S_StmMovement->obstacle
->Reg("StmMovement", robochart::Optional<Loc>());
                }
            }
            bool Check() {
                Reg();
                if (S_StmMovement->obstacle->Check(reg_obstacle_
event) == true) {
                    #ifdef SM_DEBUG
                        printf("TREATING EVENT obstacle\n");
                    #endif
                    S_StmMovement->dir = std::get<0>(*reg_obstac
le_event->GetOther().GetValue().lock()->GetParameters()).GetValu
e();
                    ClearEvent();
                    return true;
                }
                else {
                    Cancel();
                    return false;
                }
            }
            void Cancel() {
                if (reg_obstacle_event != nullptr) {
                    S_StmMovement->obstacle->Cancel(reg_obstacle
_event);
                    reg_obstacle_event = nullptr;
                }
```

```
            }
            void ClearEvent() {
                S_StmMovement->obstacle->AcceptAndDelete(reg_obs
tacle_event);
                reg_obstacle_event = nullptr;
            }
            void Action() {
                S_StmMovement->T->SetCounter(0);
                #ifdef SM_DEBUG
                    printf("Resetting Clock T\n");
                #endif
            }
    };
public:
class t2 : public robochart::Transition {
    private:
        std::shared_ptr<Robot> R_Robot;
        std::shared_ptr<ContMovement> C_ContMovement;
        std::shared_ptr<StmMovement> S_StmMovement;
    public:
        t2(std::shared_ptr<Robot> R_Robot, std::shared_ptr<ContM
ovement> C_ContMovement, std::shared_ptr<StmMovement> S_StmMovem
ent, std::weak_ptr<robochart::State> src, std::weak_ptr<robochar
t::State> tgt):
            robochart::Transition("S_StmMovement_t2", src, tgt),
R_Robot(R_Robot), C_ContMovement(C_ContMovement), S_StmMovement(
S_StmMovement)
        {}
        bool Condition() {
            if (S_StmMovement->T->GetCounter() >= R_Robot->PI) {
                #ifdef SM_DEBUG
                    printf("Condition of transition S_StmMovemen
t_t2 is true\n");
                #endif
                return true;
            }
            else {
                #ifdef SM_DEBUG
                    printf("Condition of transition S_StmMovemen
t_t2 is false\n");
```

```
                #endif
                return false;
            }
        }
};
public:
class t0 : public robochart::Transition {
    private:
        std::shared_ptr<Robot> R_Robot;
        std::shared_ptr<ContMovement> C_ContMovement;
        std::shared_ptr<StmMovement> S_StmMovement;
    public:
        t0(std::shared_ptr<Robot> R_Robot, std::shared_ptr<ContM
ovement> C_ContMovement, std::shared_ptr<StmMovement> S_StmMovem
ent, std::weak_ptr<robochart::State> src, std::weak_ptr<robochar
t::State> tgt):
           robochart::Transition("S_StmMovement_t0", src, tgt),
R_Robot(R_Robot), C_ContMovement(C_ContMovement), S_StmMovement(
S_StmMovement)
        {}
};

};
```

A state machine is essentially treated as a composite state with a vector of substates. Notice that the instantiation of state machine is incomplete. It is declared without its substates and transitions in order to simplify the constructor and also due to a circular dependency between states and transitions. Therefore, in the constructor of the state machine, its substates should be instantiated.

States should be instantiated bottom up, and transitions should be instantiated after the source and target states are instantiated, and added to their respective source states.

Classes

The classes provided in the framework include some aspects of the semantics of the corresponding RoboChart components. They cover the following components:

- Module
- Robotic Platform
- Controller
- StateMachine
- State
- Transition
- Channel
- Event

Module

The module class is the root of the simulation. When used in the ARGoS simulator it implements a `CCI_Controller` class, which provides a `ControlStep()` function that does not take any parameters and describes a single step of execution. Any subclass of `Module` must implement the methods `Init()` and `ControlStep()` .

In ARGoS, `Init()` has the following signature:

```
virtual void Init(argos::TConfigurationNode& t_node);
```

It takes a `TConfigurationNode` that can be used to read parameters from the XML experiment file. The module class overwrites the `ControlStep()` function of the `CCI_Controller` class and calls the `Execute()` function. For the function `Execute()` , in the current version of the framework, it first calls the `Sense()` method of the robot, then the `Execute()` method of the controller and finally the `Actuate()` method of the robot. The implementation of the module class for the obstacle avoidance example is as follows:

```
//OAModule.h
#include "Robot.h"
#include "ContMovement.h"
#include "ChannelTypes.h"

#include <argos3/core/control_interface/ci_controller.h>

class OAModule : public argos::CCI_Controller {
public:
    OAModule() :
            OAModule_Robot(nullptr),
            OAModule_ContMovement(nullptr),
            obstacle(std::make_shared<robochart::obstacle_channe
l>("obstacle")) {}
    virtual ~OAModule() {};

    virtual void Init(argos::TConfigurationNode& t_node);
    virtual void ControlStep();
    void Execute();

private:
    std::shared_ptr<robochart::obstacle_channel> obstacle;
    std::shared_ptr<Robot> OAModule_Robot;
    std::shared_ptr<ContMovement> OAModule_ContMovement;
};
```

```
//OAModule.cpp
#include "OAModule.h"
#include "StmMovement.h"

void OAModule::Init(argos::TConfigurationNode& t_node) {
    OAModule_Robot = std::make_shared<Robot> (obstacle);
    OAModule_ContMovement = std::make_shared<ContMovement> (OAMo
dule_Robot, obstacle);
    std::shared_ptr<StmMovement> ContMovement_StmMovement = std:
:make_shared<StmMovement>(OAModule_Robot, OAModule_ContMovement,
 obstacle);
    OAModule_ContMovement->stm = ContMovement_StmMovement;

    // Epuck Sensors
    OAModule_Robot->EventsI::light_sensor_epuck = GetSensor<argo
s::CCI_EPuckLightSensor>("epuck_light");
    OAModule_Robot->EventsI::proximity_sensor_epuck = GetSensor<
argos::CCI_EPuckProximitySensor>("epuck_proximity");
    // Epuck Actuators
    OAModule_Robot->MovementI::wheels_actuator = GetActuator<arg
os::CCI_EPuckWheelsActuator>("epuck_wheels");
    OAModule_Robot->MovementI::base_leds_actuator = GetActuator<
argos::CCI_EPuckBaseLEDsActuator>("epuck_base_leds");
}

void OAModule::Execute() {
    OAModule_Robot->Sense();
    OAModule_ContMovement->Execute();
    OAModule_Robot->Actuate();
}

void OAModule::ControlStep() {
    Execute();
    printf("\n");
}
```

Hardware Component

A subclass of `HardwareComponent` should implement two methods `Sense()` and `Actuate()` . These classes provide a connection between the simulator (ARGoS in our case) and the RoboChart model. An extension of `HardwareComponent` function is as follows:

```
//HardwareComponent.h
/***********epuck*************/
#include <argos3/plugins/robots/e-puck/control_interface/ci_epuc
k_wheels_actuator.h>
#include <argos3/plugins/robots/e-puck/control_interface/ci_epuc
k_base_leds_actuator.h>
#include <argos3/plugins/robots/e-puck/control_interface/ci_epuc
k_rgb_leds_actuator.h>
#include <argos3/plugins/robots/e-puck/control_interface/ci_epuc
k_proximity_sensor.h>
#include <argos3/plugins/robots/e-puck/control_interface/ci_epuc
k_light_sensor.h>

using namespace argos;

class HardwareComponent {

public:
    HardwareComponent() {}
    virtual ~HardwareComponent() {}
    virtual void Sense() = 0;
    virtual void Actuate() = 0;

    // Sensors
    CCI_EPuckProximitySensor* proximity_sensor_epuck;
    CCI_EPuckLightSensor* light_sensor_epuck;

    // Actuators
    CCI_EPuckWheelsActuator* wheels_actuator;
    CCI_EPuckBaseLEDsActuator* base_leds_actuator;
};
```

In this function, the model (sensors and actuators) of the robot (e-puck in our case) is defined.

Interface

The interface class implements the `HardwareComponent` class. For a particular implementation, the `Sense()` function needs to be expanded to check whether the events happen or not. In our example, it is `obstacle`. The `Actuate()` function needs to be expanded to set the motor of the robot. For example, for differential wheeled robot, the `Actuate()` function will set the left and right speed of the robot.

Robotic Platform

The hardware specific operations, variables and events of the robotic platform can be either grouped in interfaces or defined inside the platform. If the operations and/or variables are grouped in interfaces, a robotic platform should *provides (represented by a symbol **P** in the RoboChart diagram*) these interfaces*. These operations grouped in the interfaces can then be used later in the controller or state machine. In order to use the operations, the controller/state machine need to require (represented by a symbol **R** in the RoboChart diagram) the interfaces provided by a robotic platform. Therefore, in the simulation, the controller/state machine class needs to have a reference of the robotic platform class.*

*A robotic platform also defines (represented by a symbol **i*** in the diagram*) interfaces that group events. Note that the operations and events can not be grouped in the same interface. The events are connected between different components using directional arrow to exchange message.* The implementation of a robotic platform must extend Interface subclasses.

Controller

A controller contains a single reference to a state machine that must be instantiated in the variable `stm` by the functionl `Init()` in the Module class. This is the only method that must be defined by the controller class. It is used to instantiate states and transitions, build the state machine and initialise the `stm` variable.

```
#ifndef ROBOCALC_CONTROLLER_H_
#define ROBOCALC_CONTROLLER_H_

#include "State.h"

namespace robochart {

class Controller {
public:
    std::shared_ptr<StateMachine> stm;
    Controller() {}
    virtual ~Controller() {}
    virtual void Execute() {
        if (stm != nullptr) stm->Execute();
    }
    virtual void Initialise() {}
};

}

#endif
```

State

The state machine components are defined in the `State.h` header file. It declares three classes: State, State Machine and Transition. A subclass of `State` can implement the following methods:

- `Entry()` : implement the entry action of the state;
- `During()` : implement the during action of the state;
- `Exit()` : implement the exit action of the state;
- `Initial()` : return the index of the initial substate.

The transitions of the state are stored in the variable `transitions` . The code for `State` and `Transition` class is as follows:

```
//State.h

#ifndef ROBOCALC_STATE_H_
#define ROBOCALC_STATE_H_

#include <vector>
#include <memory>

#define STATE_DEBUG

namespace robochart {
class Transition;
class State {

public:
    std::string name;
    bool mark;
    State(std::string n) : name(n), stage(s_Inactive), mark(false
) {}
    virtual ~State() { printf("Deleting state %s\n", name.c_str(
)); }
    virtual void Entry() {}
    virtual void During() {}
    virtual void Exit() {}
```

```
    virtual int Initial() { return -1; }

    enum Stages {
        s_Enter, s_Execute, s_Exit, s_Inactive
    };
    Stages stage;
    std::vector<std::shared_ptr<State>> states;
    std::vector<std::shared_ptr<Transition>> transitions;
    virtual void Execute();

    bool TryTransitions();

    bool TryExecuteSubstates(std::vector<std::shared_ptr<State>>
 s);

    void CancelTransitions(int i);
};

class StateMachine: public State {
public:
    StateMachine(std::string n): State(n) {}
    virtual ~StateMachine() {}
};

class Transition {

private:
    std::weak_ptr<State> source, target;
public:
    std::string name;
public:
    Transition(std::string n, std::weak_ptr<State> src, std::wea
k_ptr<State> tgt) :
            name(n), source(src), target(tgt) {
    }
    virtual void Reg() {}
    virtual bool Check() { return true; }
    virtual void Cancel() {}
    virtual bool Condition() { return true; }
    virtual void Action() {}
```

```
    virtual void ClearEvent() {};
    virtual ~Transition() { source.reset(); target.reset(); prin
tf("Deleting transition\n");}
    bool Execute();
};

}

#endif
```

If the state is composite, its substates are stored in the variable `states`. In this case, the function `Initial()` must be implemented and return the index of the initial state. Any state (e.g. a state machine) must extend the class State and can provide entry, during and exit actions.

Notice that it is not possible to define the transitions of the state directly in the class because of a circular dependency between states and transitions. For this reason, transitions must be instantiated and added to the variable `transitions` of the source state.

For example, in our obstacle avoidance example, the `Turning` state has reference to the robot (to access the `Move` operation), and the state machine (to access variable `dir` and clock `T`); it provides the entry action that sets the angular speed of the robot.

State Machine

A state machine is just a state. This class exists only to make the notion of a state machine explicit. To execute a state machine, the `execute` function is called. Inside this function, the functions try_*execute_substates, try*_transitions, cancel_transitions are called. A state has three stages: `s_Enter`, `s_Execute` and `s_Exit`.

```
//State.cpp
#include "State.h"

namespace robochart {
```

```
bool Transition::Execute() {
    if (Condition() & Check()) {  //check condition() first if i
t is false no need to perform check(); condition() && check()
        auto src = source.lock();
        src->stage = State::s_Exit;
        src->Execute();
        Action();
        auto tgt = target.lock();
        tgt->stage = State::s_Enter;
        tgt->Execute();
        return true;
    }
    return false;
}

void State::Execute() {
    switch (stage) {
    case s_Enter:
#ifdef STATE_DEBUG
        printf("Entering State %s\n", this->name.c_str());
#endif
        Entry();
        if (Initial() >= 0) {
            states[Initial()]->stage = s_Enter;  //this has alre
ady makes sure that every time the state machine is entered, it
starts executing from initial state?
            states[Initial()]->Execute();
        }
        stage = s_Execute;
        break;
    case s_Execute:
#ifdef STATE_DEBUG
        printf("Executing a state %s\n", this->name.c_str());
#endif
        while(TryExecuteSubstates(states));      //this makes su
re more than one transition can happen at one cycle; execute the
 state from bottom to up
        if (TryTransitions() == false) {
#ifdef STATE_DEBUG
```

```
            printf("Executing during action of %s!\n", this->nam
e.c_str());
#endif
            During();                                   //if no trans
ition is enabled, execute during action in every time step
        }
        else {
#ifdef STATE_DEBUG
            printf("Not Executing during action of %s!\n", this-
>name.c_str());
#endif
        }
        break;
    case s_Exit:
        Exit();
        stage = s_Inactive;
        break;
    }
}

bool State::TryTransitions() {
#ifdef STATE_DEBUG
    printf("trying %ld transitions\n", transitions.size());
#endif
    for (int i = 0; i < transitions.size(); i++) {
#ifdef STATE_DEBUG
        printf("trying transition: %s\n", transitions[i]->name.c
_str());
#endif
        bool b = transitions[i]->Execute();
        if (b) {
            this->mark = true;
            CancelTransitions(i);  //erase OTHER events (in the
channel) already registered by the transitions of this state, as
 the state tried its every possible transitions
#ifdef STATE_DEBUG
            printf("transition %s true\n", transitions[i]->name.
c_str());
#endif
            return true;
```

```
        }
        else {
#ifdef STATE_DEBUG
            printf("transition %s false\n", transitions[i]->name
.c_str());
#endif
        }
    }
    this->mark = false;
    return false;
}

void State::CancelTransitions(int i) {
    for (int j = 0; j < transitions.size(); j++) {
        if (j != i) {
#ifdef STATE_DEBUG
            printf("CANCEL transition: %s\n",transitions[j]->nam
e.c_str());
#endif
            transitions[j]->Cancel();
        }
    }
}

//return false either no sub states or no transitions are enable
d in the sub states
bool State::TryExecuteSubstates(std::vector<std::shared_ptr<Stat
e>> s) {
    for (int i = 0; i < s.size(); i++) {
        // printf("state index : %d; stage: %d\n", i, states[i]-
>stage);
        // there should be only one active state in a single sta
te machine
        if (s[i]->stage == s_Inactive) continue;
        else {
            s[i]->Execute();
            return s[i]->mark;      //keep trying the transition
s at the same level if there is transition from one state to ano
ther
        }
```

```
        }
        return false;
    }

    }
```

Transition

Similarly to states, transitions must extend the class Transition. They can provide a number of functions: `Reg` , `Check` , `Cancel` , `Condition` and `Action` . The first three are necessary if the transition has an event as a trigger, the fourth if the transition has a condition and the final if the transition has an action. A subclass of `Transition` may implement five optional functions:

- `void Reg()` : this function is only implemented if the transition has a trigger with an event. In this case, the function `Reg` of a channel must be called (with appropriate parameters) and the event returned by the call must be stored in a variable such as `event` ;
- `bool Check()` : this function is implemented to check the occurrence of the registered event. It calls the `Check` function of the channel on the event produced by `Reg` ;
- `void Cancel()` : this function calls the method `Cancel` of the channel with the event produced by `Reg` as a parameter;
- `bool Condition()` : this function implements the condition of the transition;
- `void Action()` : this function implements the action of the transition. If the transition's trigger contains an event, this function must call the function `Accept` or `AcceptAndDelete` of the channel (on the event obtained from `Reg`). The function `Accept` should be called if the channel is synchronous, and `AcceptAndDelete` should be called if the channel is asynchronous.

In the obstacle avoidance example, the transition `t1` has a trigger of the form `obstacle?dir` and also reset the clock `T` . This transition is implemented as the classt `1` , where `Reg` is implemented as a function that calls the function `Reg` of the channel `obstacle` of the state machine with source name

`StmMovement` (identifying the state machine StmMovement) and undefined parameter `Optional<Loc>()`. The result of the `Reg` function of the channel is a pointer to an event that is recorded in the variable `event`.

The `Check` function simply returns the result of calling the `Check` function of the channel `obstacle`. If the `Check` function return true, the first parameter of the event to the state machine variable `dir` using the function `get<0>` applied to the event. The event must be accepted to mark it as treated and set to the `nullptr`. In the case of the transition `t1`, the action finishes with a call to `AcceptAndDelete` because the channel `obstacle` is asynchronous. If it were synchronous, the method `Accept` should be called, which will only delete the event if both sides of the communication have accepted it. If the check function return false, the function `Cancel` calls the function `Cancel` of the same channel. All these calls are guarded by a check on the variable `event` that guarantees it is not null.

Transition `t2` illustrates the implementation of a transition with a condition.

The condition calculates the time steps elapsed recorded in `T`. The time step is increased in every cycle of the simulation followed by the execution of the state machine. If the time passes certain threshold, transition `t2` is trigger.

Channel

A channel is instantiated using the class `Channel` instantiated with the types of the parameters. To simplify the usage of channels and events, we suggest declaring auxiliary types. For example, the obstacle channel and events are declared as, respectively, a Channel and an Event with an optional `string` parameters. The optional class allows the construction of expressions of the form `c?x` , where `x` is an input parameter. The channel is a buffer. It has a name and also a vector of reference of the events. The size of the buffer should be always kept to 2. This means the communication is always one-to-one.

```
//Channel.h

#ifndef ROBOCALC_CHANNEL_H_
#define ROBOCALC_CHANNEL_H_

#include <stdlib.h>
#include <memory>
#include <mutex>
#include <set>
#include <iostream>
#include <algorithm>
#include "Event.h"

namespace robochart {

template<typename ...Args>
class Channel {
private:
    std::string name;
//    std::mutex m;
    std::vector<std::shared_ptr<Event<Args...>>>events;  //store
 the number of shared event pointer in the channel
public:
    Channel(std::string n):
    name(n) {
```

```
    }
    virtual ~Channel() {
    }
    /* return the size of the channel
     *
     */
    uint Size() {
        return events.size();
    }
    /* clear the channel
     *
     */
    void Clear() {
        events.clear();
    }

    /* Args ... args: the number of arbitrary type of arguments
of the event;
     * In principle, there can be multiple numbers, but here we
use one, which is Optional.
     * The Optional class includes the value of the argument
     */
    std::shared_ptr<Event<Args...>> Reg(std::string source, Args
 ... args) {
        //std::lock_guard<std::mutex> lck(m);
        std::shared_ptr<Event<Args...>> e = std::make_shared<Eve
nt<Args...>>(name,
                source, args...);    //the event in the channel
is instantiated here: call Event constructor
        events.push_back(e);         //push the shared event poi
nter into the channel
#ifdef EVENT_DEBUG
        printf("channel %s size (registered): %ld\n", GetName().
c_str(), events.size());
#endif
        return e;
    }

    //This is the overloaded function
    std::shared_ptr<Event<Args...>> Reg(std::shared_ptr<Event<Ar
```

```
gs...>> ci) {
        //std::lock_guard<std::mutex> lck(m);
        events.push_back(ci);
        return ci;
    }

    bool Check(std::shared_ptr<Event<Args...>> e) {
        //std::lock_guard<std::mutex> lck(m);
        for (typename std::vector<std::shared_ptr<Event<Args...>
>>::iterator it = events.begin();
                it != events.end(); ++it) {
            #ifdef EVENT_DEBUG
                printf("channel %s size: %ld\n", GetName().c_str
(), events.size());
            #endif
            if (e->Compatible(*it)) {
                e->Match(*it);
                (*it)->SetOther(e);   //e->match(*it) and (*it)-
>setOther(e) will make sure the matched events will have a refer
ence to each other
#ifdef EVENT_DEBUG
                printf("checking of channel %s is true\n", GetNa
me().c_str());
#endif
                return true;
            }
        }
#ifdef EVENT_DEBUG
        printf("checking of channel %s is false\n", GetName().c_
str());
#endif
        return false;
    }
    void Cancel(std::shared_ptr<Event<Args...>> e) {
        //std::lock_guard<std::mutex> lck(m);
        if (!e->GetOther().Exists()) {
            typename std::vector<std::shared_ptr<Event<Args...>>
>::iterator position = std::find(events.begin(), events.end(), e
);
            if (position != events.end())
```

```
                events.erase(position);
#ifdef EVENT_DEBUG
            printf("event %s (%d) removed from channel\n", GetNa
me().c_str(), e != nullptr);
#endif
        } else {
#ifdef EVENT_DEBUG
            printf("error removing event from channel %s\n", Get
Name().c_str());
#endif
        }
    }

    void Accept(std::shared_ptr<Event<Args...>> e) {    //This w
ill create a temp new shared pointer which will be out of scope
when the function terminates
        //std::lock_guard<std::mutex> lck(m);
        if (e->getOther().exists()) {
            e->accept();            //The first component will o
nly accept the event (but not delete the event in the channel),
because e->getOther().value().lock()->isAccepted() is false;
                                    //the second component will
accept the event as well; but it will also delete both events in
 the channel, because e->getOther().value().lock()->isAccepted()
 becomes true.
            if (e->getOther().value().lock()->isAccepted()) {

                // The other has already been accepted so I can
remove and reset both
                std::weak_ptr<Event<Args...>> other = e->getOthe
r().value();
                typename std::vector<std::shared_ptr<Event<Args.
..>>>::iterator p1 = std::find(events.begin(), events.end(), e);
                if (p1 != events.end())
                    events.erase(p1);
                typename std::vector<std::shared_ptr<Event<Args.
..>>>::iterator p2 = std::find(events.begin(), events.end(), oth
er.lock());
                if (p2 != events.end())
                    events.erase(p2);  //erase the share_ptr usi
```

```
ng weak_ptr.lock; the weak_ptr will expire when it is out of the
 if scope; in this case, both the shared_point are deleted in th
e channel
#ifdef EVENT_DEBUG
                printf("done accepting and reseting event\n");
#endif
            }
        }
    }

    void AcceptAndDelete(std::shared_ptr<Event<Args...>> e) {
        //std::lock_guard<std::mutex> lck(m);
        if (e->GetOther().Exists()) {      //delete both shared_
ptr in the channel; if check() returns true; e->getOther().exist
s() will return true
            std::weak_ptr<Event<Args...>> other = e->GetOther().
GetValue();
            typename std::vector<std::shared_ptr<Event<Args...>>
>::iterator p1 = std::find(events.begin(), events.end(), e);
            if (p1 != events.end())
                events.erase(p1);
            typename std::vector<std::shared_ptr<Event<Args...>>
>::iterator p2 = std::find(events.begin(), events.end(), other.l
ock());
            if (p2 != events.end())
                events.erase(p2);
        }
#ifdef EVENT_DEBUG
        printf("channel %s size after acceptance: %ld\n", GetNam
e().c_str(), events.size());
#endif
    }

    std::string GetName() {
        return name;
    }
};

}
```

```
#endif
```

- `Reg` This function registers an event in the channel. That is, an event pointer is pushed into the channel buffer. Every time a transition is tried, the `Reg` function will be called.
- `Check` This function checks whether two events in the channel are compatible or not. It returns true if they are compatible; false vice versa. When the two events comes from two different sources and the type of the event is the same, these two events are considered as compatible. If `Check` is successful, each event would have a reference to each other via the functions `Match` and `SetOther` .
- `Cancel` This function remove and destroy the event from the channel buffer. For example, if an event is registered into the channel but `Check` is false, then this event need to be removed from the channel.
- `Accept` and `AcceptAndDelete` These two functions remove the events from the channel if they are treated (after `check` return true). `Accept` is used in the synchronous communication, and `AcceptAndDelete` is used in the asynchronous communication.
- `Clear` This will clear the channel and all the shared pointers in the channel will be destroyed. When you clear the vector, the shared pointers it contains are destroyed, and this action automatically de-allocates any encapsulated objects with no more shared pointers referring to them.

The entire purpose of smart pointers is that they manage memory for you, and the entire purpose of shared pointers is that the thing they point to is automatically freed when no more shared pointers point to it.

Event

The `Event` class is used only as parameters to the channel function and to obtain the values associated with the event. The `Event` class has the attributes of a channel name, source name (stating where it comes from). A event can also have parameters in order to pass information (via `send` e!x), the value would be stored in the attribute of parameters. It is a tuple which means it can store multiple value rather than a single value. The value can be empty. In this case, when an event is registered, only the source name would be passed. The `Event` class also has an attribute of reference of the other paired event inside the channel. This value is only set when the events in the channel are matched. The `Event` class also has a attribute accepted, marking whether an event is accepted/treated by the other component.

```
//Event.h

#ifndef ROBOCALC_CHANNEL_H_
#define ROBOCALC_CHANNEL_H_

#include <stdlib.h>
#include <memory>
#include <mutex>
#include <set>
#include <iostream>
#include <algorithm>
#include "Event.h"

namespace robochart {

template<typename ...Args>
class Channel {
private:
    std::string name;
//    std::mutex m;
    std::vector<std::shared_ptr<Event<Args...>>>events;  //store
 the number of shared event pointer in the channel
public:
```

```
    Channel(std::string n):
    name(n) {
    }
    virtual ~Channel() {
    }
    /* return the size of the channel
     *
     */
    uint Size() {
        return events.size();
    }
    /* clear the channel
     *
     */
    void Clear() {
        events.clear();
    }

    /* Args ... args: the number of arbitrary type of arguments
of the event;
     * In principle, there can be multiple numbers, but here we
use one, which is Optional.
     * The Optional class includes the value of the argument
     */
    std::shared_ptr<Event<Args...>> Reg(std::string source, Args
 ... args) {
        //std::lock_guard<std::mutex> lck(m);
        std::shared_ptr<Event<Args...>> e = std::make_shared<Eve
nt<Args...>>(name,
                source, args...);    //the event in the channel
is instantiated here: call Event constructor
        events.push_back(e);         //push the shared event poi
nter into the channel
#ifdef EVENT_DEBUG
        printf("channel %s size (registered): %ld\n", GetName().
c_str(), events.size());
#endif
        return e;
    }
```

```
    //This is the overloaded function
    std::shared_ptr<Event<Args...>> Reg(std::shared_ptr<Event<Ar
gs...>> ci) {
        //std::lock_guard<std::mutex> lck(m);
        events.push_back(ci);
        return ci;
    }

    bool Check(std::shared_ptr<Event<Args...>> e) {
        //std::lock_guard<std::mutex> lck(m);
        for (typename std::vector<std::shared_ptr<Event<Args...>
>>::iterator it = events.begin();
                it != events.end(); ++it) {
            #ifdef EVENT_DEBUG
                printf("channel %s size: %ld\n", GetName().c_str
(), events.size());
            #endif
            if (e->Compatible(*it)) {
                e->Match(*it);
                (*it)->SetOther(e);   //e->match(*it) and (*it)-
>setOther(e) will make sure the matched events will have a refer
ence to each other
#ifdef EVENT_DEBUG
                printf("checking of channel %s is true\n", GetNa
me().c_str());
#endif
                return true;
            }
        }
#ifdef EVENT_DEBUG
        printf("checking of channel %s is false\n", GetName().c_
str());
#endif
        return false;
    }
    void Cancel(std::shared_ptr<Event<Args...>> e) {
        //std::lock_guard<std::mutex> lck(m);
        if (!e->GetOther().Exists()) {
            typename std::vector<std::shared_ptr<Event<Args...>>
>::iterator position = std::find(events.begin(), events.end(), e
```

```
);
            if (position != events.end())
               events.erase(position);
#ifdef EVENT_DEBUG
            printf("event %s (%d) removed from channel\n", GetNa
me().c_str(), e != nullptr);
#endif
        } else {
#ifdef EVENT_DEBUG
            printf("error removing event from channel %s\n", Get
Name().c_str());
#endif
        }
    }

    void Accept(std::shared_ptr<Event<Args...>> e) {    //This w
ill create a temp new shared pointer which will be out of scope
when the function terminates
        //std::lock_guard<std::mutex> lck(m);
        if (e->getOther().exists()) {
            e->accept();            //The first component will o
nly accept the event (but not delete the event in the channel),
because e->getOther().value().lock()->isAccepted() is false;
                                    //the second component will
accept the event as well; but it will also delete both events in
 the channel, because e->getOther().value().lock()->isAccepted()
 becomes true.
            if (e->getOther().value().lock()->isAccepted()) {

                // The other has already been accepted so I can
remove and reset both
                std::weak_ptr<Event<Args...>> other = e->getOthe
r().value();
                typename std::vector<std::shared_ptr<Event<Args.
..>>>::iterator p1 = std::find(events.begin(), events.end(), e);
                if (p1 != events.end())
                    events.erase(p1);
                typename std::vector<std::shared_ptr<Event<Args.
..>>>::iterator p2 = std::find(events.begin(), events.end(), oth
er.lock());
```

```
                if (p2 != events.end())
                    events.erase(p2);  //erase the share_ptr usi
ng weak_ptr.lock; the weak_ptr will expire when it is out of the
 if scope; in this case, both the shared_point are deleted in th
e channel
#ifdef EVENT_DEBUG
                printf("done accepting and reseting event\n");
#endif
            }
        }
    }

    void AcceptAndDelete(std::shared_ptr<Event<Args...>> e) {
        //std::lock_guard<std::mutex> lck(m);
        if (e->GetOther().Exists()) {      //delete both shared_
ptr in the channel; if check() returns true; e->getOther().exist
s() will return true
            std::weak_ptr<Event<Args...>> other = e->GetOther().
GetValue();
            typename std::vector<std::shared_ptr<Event<Args...>>
>::iterator p1 = std::find(events.begin(), events.end(), e);
            if (p1 != events.end())
                events.erase(p1);
            typename std::vector<std::shared_ptr<Event<Args...>>
>::iterator p2 = std::find(events.begin(), events.end(), other.l
ock());
            if (p2 != events.end())
                events.erase(p2);
        }
#ifdef EVENT_DEBUG
        printf("channel %s size after acceptance: %ld\n", GetNam
e().c_str(), events.size());
#endif
    }

    std::string GetName() {
        return name;
    }
};
```

```
}

#endif
```

- `Compatible` Check whether two events in the channel come from different sources by comparing two strings. This means the communication is only valid between two different components. The state machine can not `send` an event that triggers a transition of its own.
- `Match` Set the reference of each event. This is used for deleting the event from the channel after they are treated. If it is asynchronous communication, after the events are treated (e.g. a transition is enabled), both registered will be deleted from the channel by one side of the communication. In the obstacle avoidance example, the state machine will be responsible to clear the paired events from the channel.

Optional

The `Optional` class is a template class. It can be used to define the associated value of an event. The event type can be any primitive or used-defined types. It is also used for define the other event as the paired event needs to have the reference to each other. Usage examples:

```
Optional<double> doubleType
```

```
Optional<std::weak_ptr<Event<Args...>>> otherEvent
```

```
//Optional.h

#ifndef ROBOCALC_OPTIONAL_H_
#define ROBOCALC_OPTIONAL_H_

namespace robochart {

template <typename T>
class Optional {
public:
    Optional(): set(false) {}
    Optional(T t): set(true),v(t) {}

    bool Exists() {
        return set;
    }
    T GetValue() {
        return v;
    }
    void SetValue(T t) {
        set = true;
        v = t;
        printf("### CHANGING VALUE OF OPTIONAL\n");
    }
    ~Optional() {}
private:
    T v;
    bool set;
};

}

#endif
```

We use the template class, as the type of the associated value the an event is unknown. Just like we can create **function templates**, we can also create **class templates**, allowing classes to have members that use template parameters as types.

Timer

In simulation, the smallest update unit for a timer is the control step. The simulation is executed in a cyclic way. All the timers are updated in the same phase, which means the clock is global.

```
//Timer.h

#ifndef ROBOCALC_TIMER_H_
#define ROBOCALC_TIMER_H_

#include <string>

#define TIMER_DEBUG

namespace robochart {

class Timer {
public:

    Timer(std::string name) : name(name), counter(0), waitFlag(f
alse), waitPeoriod(0), startingCounter(65535)
    {}

    int GetCounter() const {
        return counter;
    }

    void SetCounter(int i) {counter = i;}

    void IncCounter() {
        counter++;
        if (counter - startingCounter >= waitPeoriod) {
            waitFlag = false;
            startingCounter = 65535;
        }
        #ifdef TIMER_DEBUG
                printf("%s counter: %d\n", name.c_str(), counter
```

```
);
        #endif
    }

    void Wait(int i) {
        waitFlag = true;
        waitPeoriod = i;
        startingCounter = counter;
    }

    bool CheckWaitStatus() {
        return waitFlag;
    }

    std::string GetName() {
        return name;
    }

private:
    int counter, waitPeoriod, startingCounter;
    bool waitFlag;
    std::string name;
};

}

#endif
```

The following are current constraints of usage of RoboChart for constructing a simulation. Some of them are the constraints of RoboChart language, while the others are for the user to construct a useful simulation.

- The name and type of the event used for connecting communication between different components should be the same.
- The generated code is not targeted for some specific simulator. The user needs to extend the framework to fit different needs.
- The clock should only be defined inside the state machine.
- The simulation framework can execute multiple transitions at one control cycle, but it may cause the 'infinite loop' issue. This is known as deadlock.
- Defined interfaces only contain events.
- No event trigger is allowed for the outgoing transition of a junction or probabilistic junction.
- A module should be self-contained. The operations used by state machine should be provided by the robotic platform or defined inside the controller.
- There should be be only one entry/exit action in each state, but the entry/exit action can include a sequence of statements.
- The name of variables and operations should be unique.
- Only support since for the clock related condition.

A video showing how to build a basic state machine controller can be found here:

https://youtu.be/I5L_PGstu40

The complete code for the obstacle avoidance example can be found here:

https://drive.google.com/file/d/1xLGi0AyrETHuEqb5r4m4SmGwW8ymMWeG/view?usp=sharing