HMEE303 - CONTROL ARCHITECTURES PROJECT REPORT

AUTHOR: EGE GURSOY

PROFESSOR: BENJAMIN NAVARRO

Objective

In this project we present a simulation of a production line which has a goal to assemble 3 different types of parts coming from a supply conveyor and to evacuate the unnecessary pieces. We have 3 main steps to achieve that goal. The first step is to create petri net models to confirm our approach, the second step is the implementation of the petri net models to the Robot Operating System (ROS) and the third step is the simulation of the system under V-Rep simulator.

Description of materials

The production line consists of a camera, a robotic arm, a supply conveyor, an evacuation conveyor and an assembly station. Each component contains a set of actuators and sensors to perform their role. We will look closely at each component in the following headlines.

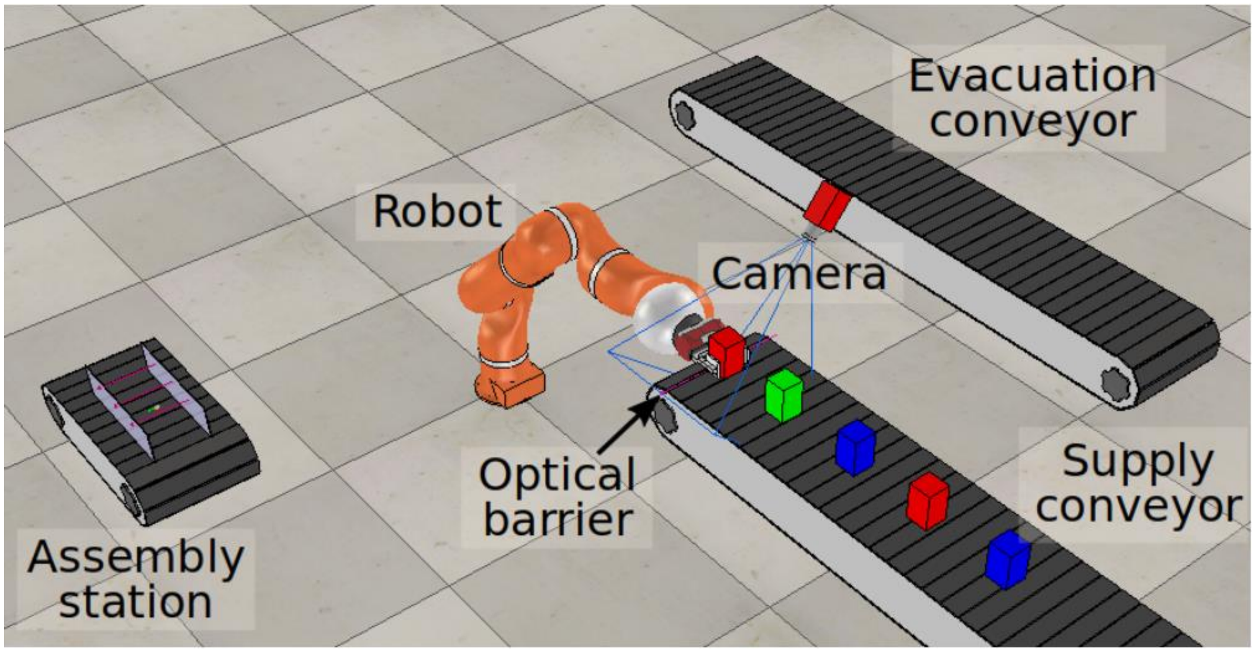


Figure : Scene description

Supply conveyor

The supply conveyor brings parts to be assembled in a random order. An optical barrier is set up at its end to detect parts to be processed.

Actuator : Run (SC_ON)

Sensor : Optical barrier (SC_OB)

Camera

The camera above the optical barrier is in the charge of detecting which part arrived.

Actuator : Image processing (CAM_PROCESSING)

Sensors : Recognition performed (CAM_DONE), Part 1 recognized (CAM_PART1), Part 2 recognized (CAM_PART2), Part 3 recognized (CAM_PART3)

Assembly station

The assembly station has three slots, one for each part that have to be filled correctly in order for the assembly to be considered valid.

Actuator : Check_assembly (AS_CHECK)

Sensors : Assembly valid (AS_VALID), Assembly evacuated (AS_EVACUATED)

Evacuation conveyor

The evacuation conveyor is here to dispose of unneeded parts.

Actuator : Run (EC_ON)

Sensor : Stopped (EC_STOPPED)

Robot

The robot is used to take parts at the supply conveyor and bring them to either the assembly station or the evacuation station if the part is already present in the current assembly.

Actuators : Move to the left (ROB_LEFT), Move to the right (ROB_RIGHT), Grasp a part (ROB_GRASP), Release a part (ROB_RELEASE), Assemble part 1 (ROB_ASS_PART1), Assemble part 2 (ROB_ASS_PART2), Assemble part 3 (ROB_ASS_PART3)

Sensors: At assembly station (ROB_AT_AS), At supply conveyor (ROB_AT_SC), At evacuation conveyor (ROB_AT_EC), Part grasped (ROB_GRASPED), Part released (ROB_RELEASED), Part 1 assembled (ROB_DONE_PART1), Part 2 assembled (ROB_DONE_PART2), Part 3 assembled (ROB_DONE_PART3)

Details of operations

This assembly of three parts Part1 (red), Part2 (green) and Part3 (blue) relies on three operations : ROB_ASS_PART1, ROB_ASS_PART2 and ROB_ASS_PART3 performed by the robot at the assembly station. These operations correspond to:

ROB_ASS_PART1 : Insert Part1 into the first slot
ROB_ASS_PART2 : Insert Part2 into the second slot
ROB_ASS_PART3 : Insert Part3 into the third slot and finish the assembly

This means that the ROB_GRASP and ROB_RELEASE actions are not involved for the assembly. As mentioned earlier, parts are coming in a random order from the supply conveyor. Once a part is detected by its optical barrier (SC_OB), the conveyor must be stopped and the image processing can be started (CAM_PROCESSING).

After the part has been processed by the camera (CAM_DONE) and the result given (CAM_PARTx), the robot must grasp (ROB_GRASP) the part before going either to the assembly station (ROB_LEFT), and realize one of the assembly operations (ROB_ASS_PARTx, ROB_DONE_PARTx), or to the evacuation conveyor (ROB_RIGHT) and release (ROB_RELEASE) the unwanted part. Once the robot has grasped the part (ROB_GRASPED), the supply conveyor can be restarted (SC_ON) to bring a new part.

The evacuation conveyor must be stopped before a part can be placed on it. Since this conveyor takes time to stop, the stop signal (EC_STOPPED) must be monitored to know when a new part can be placed on. Once the robot has released the part (ROB_RELEASED) on the conveyor, the conveyor has to be restarted (EC_ON) to evacuate the part.

Once the assembly is done, it has to be checked (AS_CHECK) for validity. If the assembly is found to be valid (AS_VALID), it is automatically evacuated (AS_EVACUATED) and the assembly station becomes free to perform a new assembly.

Initial state of the system should be as following: The robot is at the assembly station (empty) and the supply conveyor is free of any parts with both conveyors are on.

Petri models

Petri nets are a useful solution for modeling discrete systems. Since our system is based on discrete events, we will use petri nets as a validation tool for our model. For the sake of simplicity at the validation stage, we will create separate nets for each machine. Then we will merge all the nets to validate the general behavior of the system.

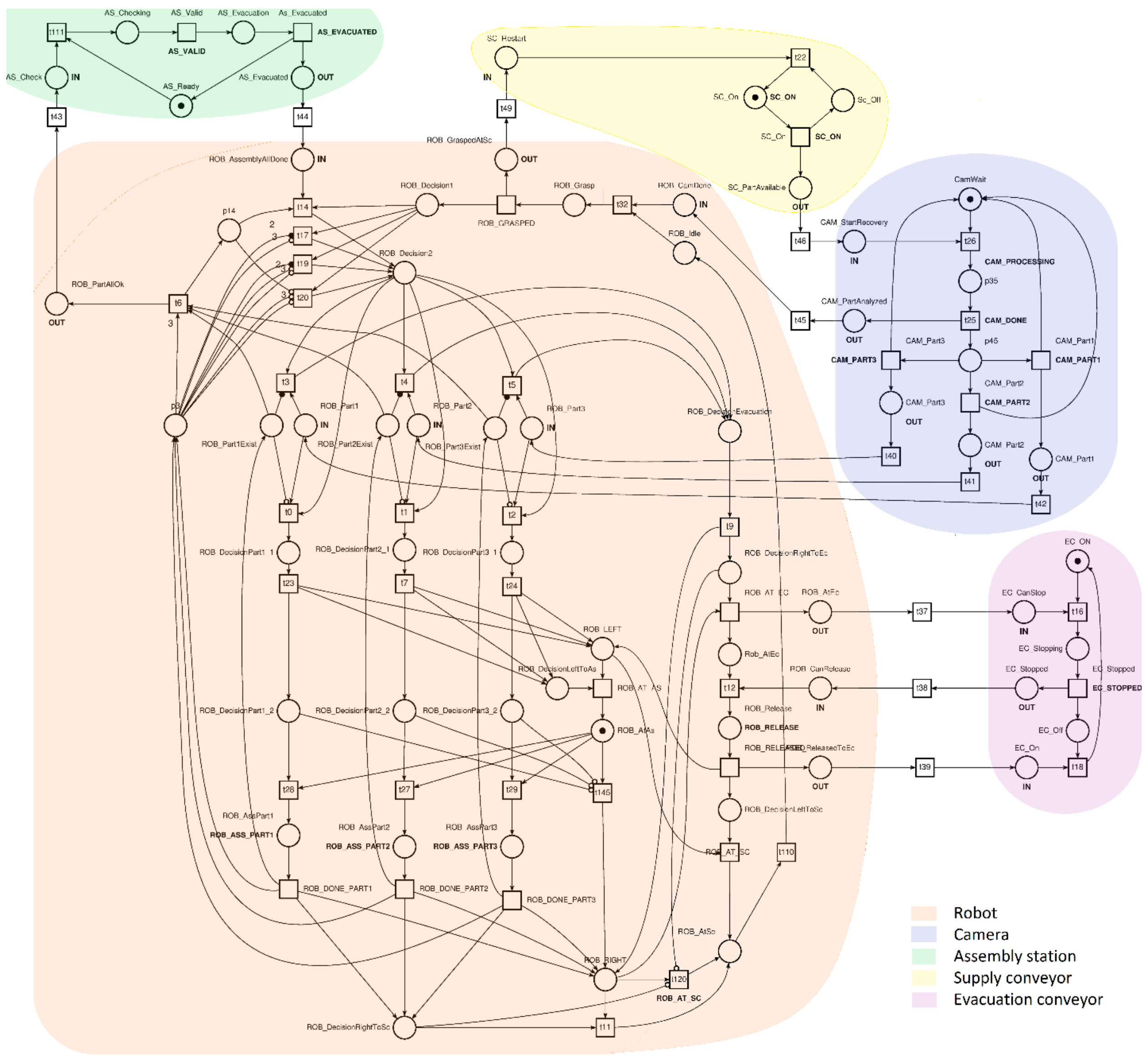


Figure : The petri nets modelization of the whole system

Supply conveyor

The supply conveyor consists of :

2 states : On, Off,

1 input : Restart,

1 output : Part available.

At the initial phase, we consider the supply conveyor as in the “On” state. Once the optical barrier detects a new part and sends the signal “SC_OB”, the supply conveyor switches to the “Off” state and outputs the “Part available” to announce a newly detected part. Then, if it receives the “Restart” input, it returns to the “On” state

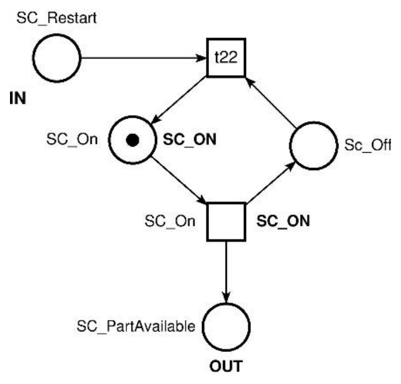


Figure : The supply conveyor petri net

Camera

The camera consists of :

6 states : Wait, Processing, Done, Part 1, Part 2, Part 3,

1 input : Start recovery,

4 outputs : Part analyzed, Part 1, Part 2, Part 3,

At the initial phase, we consider the camera as in the “Wait” state. Once it receives the “Start recovery” input, it changes it’s state to “Processing” and run the “CAM_PROCESSING” actuator.

When it receives the “CAM_DONE” signal from the sensor, it proceeds to the “Done” state and outputs the “Part analyzed” to remind the other machines. Then, it gives an output regarding to the type of the processed part determined by the sensors.

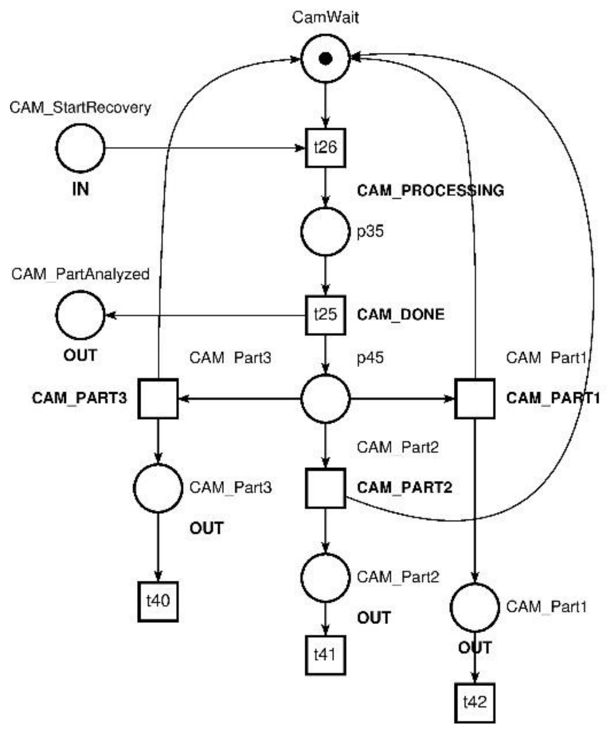


Figure : The camera petri net

Assembly station

The assembly station consists of :

3 states : Ready, Checking, Evacuation,

1 input : Check,

1 output : Evacuated

At the initial phase, we consider the assembly station as in the “Ready” state. If the “Check” input becomes active, the assembly station changes its state to the “Checking”. Once it receives the “AS_VALID” signal from the sensor, it passes towards the “Evacuation” state. Then, with the “AS_EVACUATED” signal coming from the sensor, the assembly station outputs the “Evacuated” and returns to the “Ready” state.

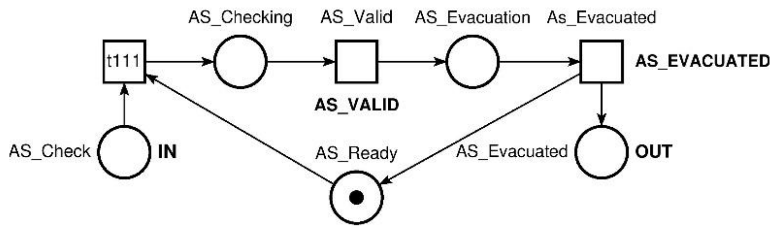


Figure : The assembly station petri net

Evacuation conveyor

The evacuation conveyor consists of :

3 states : On, Stopped, Off,

2 inputs : Can stop, On,

1 output : Stopped.

At the initial phase, we consider the assembly station as in the "On" state. Once it receives the "Can stop" input, the evacuation conveyor continues to the "Stopping" state. When it is fully stopped and received the "Stopped" signal from the sensor, the evacuation conveyor passes to the "Off" state while outputs the "Stopped". Then, with the "On" input, it returns to the "On" state.

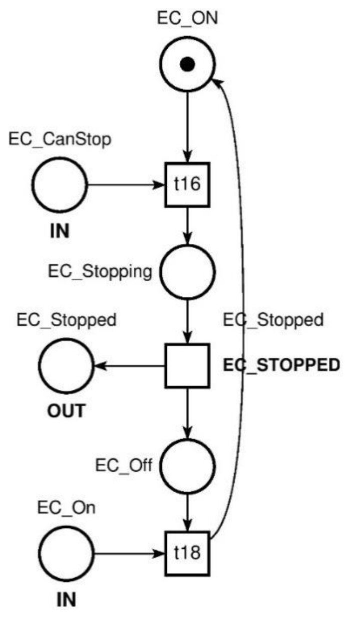


Figure : The evacuation conveyor petri net

Robot

The robot consists of :

29 states : Idle, Grasp, Release, Left, Right, At as, At sc, At ec, Decision 1, Decision 2, Decision evacuation, Decision right to sc, Decision right to ec, Decision left to sc, Decision left to as, 3 x (Part X exist), 3 x (Decision part X 1), 3 x (Decision part X 2), 3 x (Ass part X), Reset, Count,

6 inputs : Cam done, Part 1, Part 2, Part 3, Assembly all done, Can release,

4 outputs : Part all ok, Grasped at sc, At ec, Released to ec.

In this project, we choose to design a robot with the decisional capabilities. This prevents creating another machine for all the connections and the decisions. Using this method, the overall petri net looks much cleaner and easy to understand.

At the initial phase, we consider the robot as in the "At as" state. To grasp the incoming part, it should go towards right and stop at the supply conveyor. To do that, the robot first enter to the "Right" state then to the "At sc" state once it receives the "ROB_AT_SC" signal from the sensor. We set some inhibitor arcs to exclude other possibilities and force the robot to enter that sequence. Finally the robot arrives at the "Idle" state and wait the "Cam done" input to proceed.

When the "Cam done" input becomes active, robot must grab the part and decide between the assembly or evacuation procedure. To do that, the robot first enters the "Grasp" state and starts the "ROB_GRASP" actuator then to the "Decision 1" state if the "ROB_GRASPED" signal has received from the sensor. When passing between two states, the robot also sends the "Grasped at sc" output.

The "Decision 1" state is used to check if there are already three different parts sent to the assembly station during the current assembly cycle. In that case, the robot waits the "Assembly all done" input and "Reset" state to become active to continue. Otherwise, the robot continues to the "Decision 2" state.

The "Decision 2" state is used to make the decision between the assembly part X and evacuation. First, the robot waits an input signal coming from an incoming part, "Part X". Once it receives the signal, it compares with the "Part exist X" state which describes the existence of the part X at the assembly station. If it is empty then continues to the "Decision Part X 1" state. Otherwise, it continues towards "Decision evacuation" state.

If the robot follows the "Decision part X 1" route, it should go to the left until it reaches the assembly station, assemble the part X and go to the right until it reaches the supply conveyor. To do that, the robot continues to the "Left" state while activating the "Decision left to as" and

“Decision part X 2” states. Since the “Left” state used by multiple states and we can potentially stop at anywhere we want when going to the left, the “Decision left to as” used to force the robot to stop at the assembly station once it receives the “ROB_AT_AS” signal. After stopping at the assembly station, the robot can assemble the part 1, 2 or 3. We use “Decision part X 2” state to get back the robot to the right track and assemble the right part. The robot continues to the “Ass part X” state while working its “ROB_ASS_PARTX” actuator. Once it receives the “ROB_DONE_PART_X” signal from the sensor, the robot enters to the “Right” state while activating the “Decision right to sc” to force the robot to stop at the supply conveyor. In this transition, the robot also increases the value of the “Count” state by one. “Count” used to represent the number of the existing parts at the assembly station. At the end of this state, the robot returns to the “Idle“ state and waits for a new part.

On the other hand, if the robot choose the “Decision evacuation” path, it should go to the right until the evacuation conveyor, release the part and go to the left until the supply conveyor. To do that, the robot enters to the “Right” state while activates the “Decision Right to ec” state to prevent the robot from going anywhere else. Once it receives the “ROB_AT_EC” signal from sensors, the robot enters to the “At ec” state while outputs the “At ec” to remind the evacuation conveyor. Then, once the “Can release” input becomes active, robot switches to the “Release” state and runs the “ROB_RELEASE” actuator. When the sensor sends the “ROB_RELEASED” signal, the robot passes enters to the “Left” state while activating the “Decision left to sc” state and sending the “Released to ec” output. The “Decision left to sc” forces the robot to stop at the supply conveyor. The robot stops at the supply conveyor when it receives the “ROB_AT_SC” signal and enters to the “At sc” state and returns to the “Idle” state where it waits for a new part.

Finally, if the “Count” state has 3 attributes, the “Reset” state becomes active while the “Count” and all the “Part exist” states become empty. Also, the robot outputs the “Part all ok” to announce that there are three parts at the assembly station.

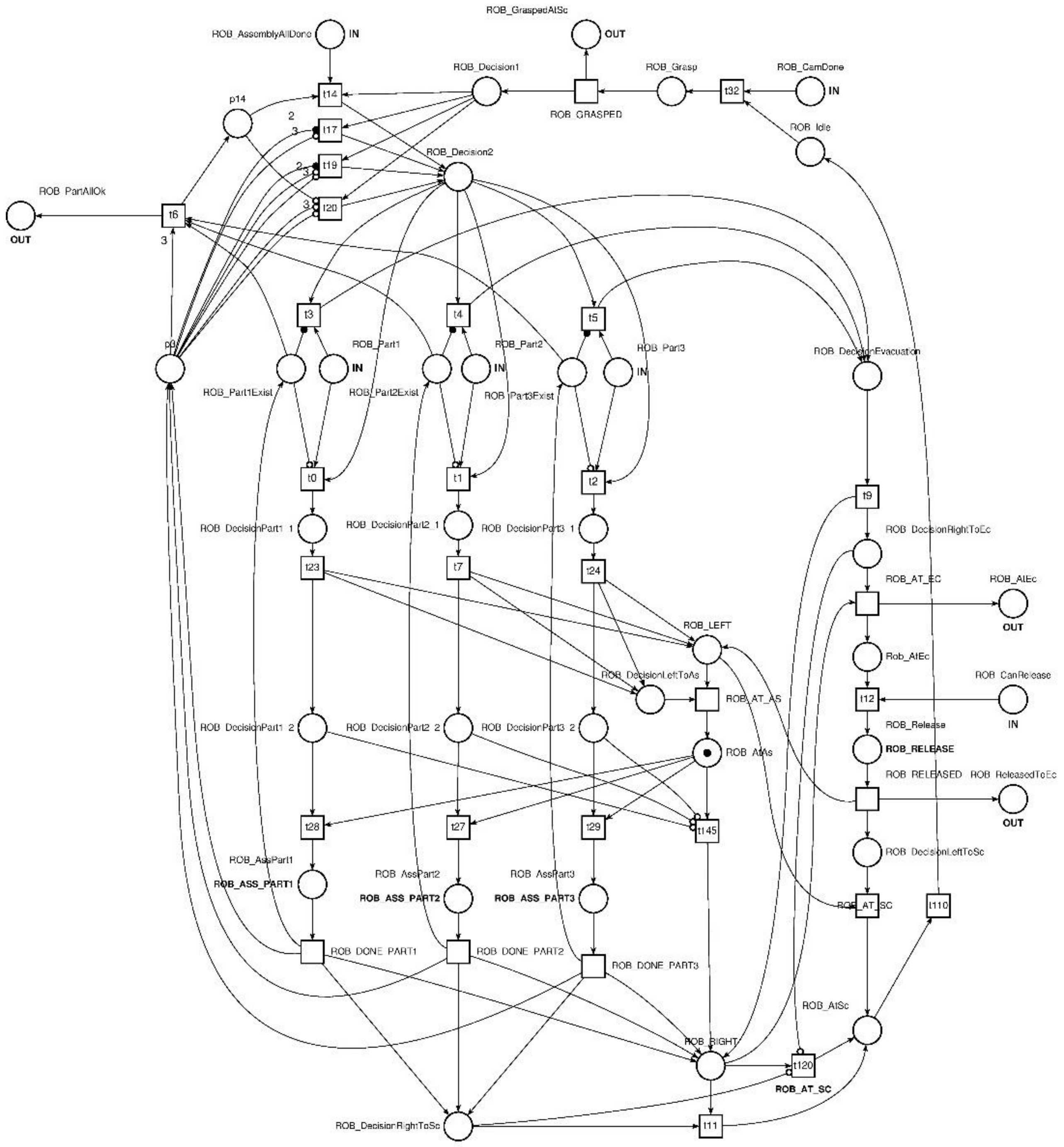


Figure : Scene description

ROS implementation

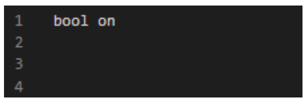
In this step we will implement our model to the ROS environment. First, we will define ROS messages to communicate between machines. Then we will create ROS nodes for each machine with an additional node to link the inputs and outputs. We will use C++ to create our nodes. Finally, we will test our model on the V-rep simulator.

ROS messages

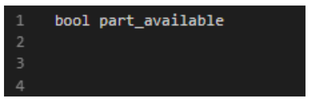
Firstly, we create four types of messages for each machine which are "command", "state", "input", "output".

Command and state types are used by the V-rep simulator and already available for each machine. The command message describe the commands to send to the related machine and the state message describe the essential states of the same machine.

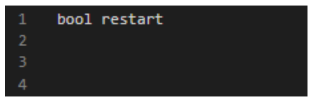
Since these two types are already available to us, we create the other two types which are the "input" and the "output" according to the petri models. Each input and output has two states, active or inactive. We can use a boolean to represent these two states. For each input, we declare a boolean in the input message and same thing for the outputs.



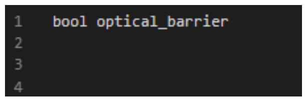
```
bool on
```



```
bool part_available
```



```
bool restart
```

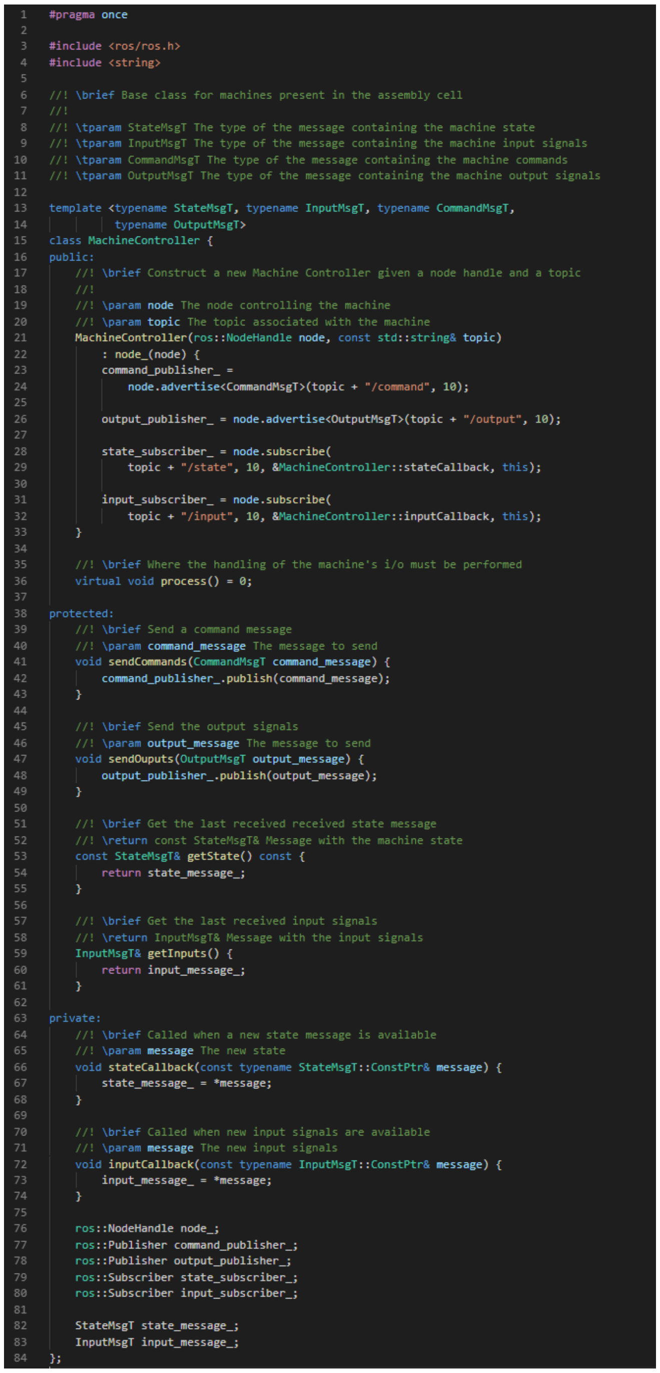


```
bool optical_barrier
```

Figure : Messages of the supply conveyor. From the left to the right "command", "output", "input", "state"

Machine controller class

A machine controller class is at our disposal in the project which can be used as a template to create a node. It automatically reads the associated message types, creates function callbacks for each message type, generates publisher and subscriber associated to node.

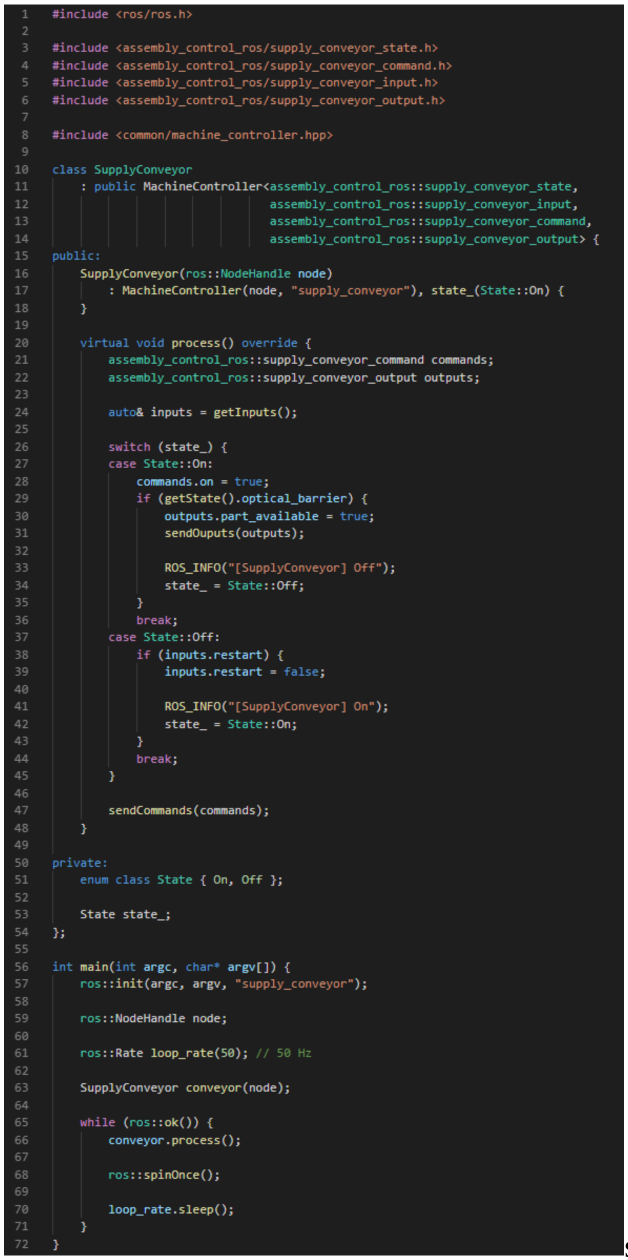


```cpp
#pragma once

#include <ros/ros.h>
#include <string>

//! \brief Base class for machines present in the assembly cell
//!
//! \tparam StateMsgT The type of the message containing the machine state
//! \tparam InputMsgT The type of the message containing the machine input signals
//! \tparam CommandMsgT The type of the message containing the machine commands
//! \tparam OutputMsgT The type of the message containing the machine output signals

template <typename StateMsgT, typename InputMsgT, typename CommandMsgT,
          typename OutputMsgT>
class MachineController {
public:
    //! \brief Construct a new Machine Controller given a node handle and a topic
    //!
    //! \param node The node controlling the machine
    //! \param topic The topic associated with the machine
    MachineController(ros::NodeHandle node, const std::string& topic)
        : node_(node) {
        command_publisher_ =
            node.advertise<CommandMsgT>(topic + "/command", 10);

        output_publisher_ = node.advertise<OutputMsgT>(topic + "/output", 10);

        state_subscriber_ = node.subscribe(
            topic + "/state", 10, &MachineController::stateCallback, this);

        input_subscriber_ = node.subscribe(
            topic + "/input", 10, &MachineController::inputCallback, this);
    }

    //! \brief Where the handling of the machine's i/o must be performed
    virtual void process() = 0;

protected:
    //! \brief Send a command message
    //! \param command_message The message to send
    void sendCommands(CommandMsgT command_message) {
        command_publisher_.publish(command_message);
    }

    //! \brief Send the output signals
    //! \param output_message The message to send
    void sendOuputs(OutputMsgT output_message) {
        output_publisher_.publish(output_message);
    }

    //! \brief Get the last received received state message
    //! \return const StateMsgT& Message with the machine state
    const StateMsgT& getState() const {
        return state_message_;
    }

    //! \brief Get the last received input signals
    //! \return InputMsgT& Message with the input signals
    InputMsgT& getInputs() {
        return input_message_;
    }

private:
    //! \brief Called when a new state message is available
    //! \param message The new state
    void stateCallback(const typename StateMsgT::ConstPtr& message) {
        state_message_ = *message;
    }

    //! \brief Called when new input signals are available
    //! \param message The new input signals
    void inputCallback(const typename InputMsgT::ConstPtr& message) {
        input_message_ = *message;
    }

    ros::NodeHandle node_;
    ros::Publisher command_publisher_;
    ros::Publisher output_publisher_;
    ros::Subscriber state_subscriber_;
    ros::Subscriber input_subscriber_;

    StateMsgT state_message_;
    InputMsgT input_message_;
};
```

Figure : Machine controller

ROS Nodes

From the beginning of the project, our approach was based on creating separate models for each machine. We carry this approach to the ROS and create a node for each machine to describe their behavior. The machines subscribe to their inputs and states, publish on their commands and outputs.

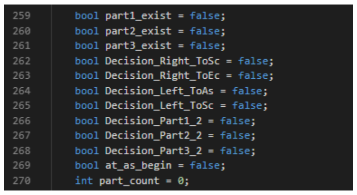


```cpp
#include <ros/ros.h>

#include <assembly_control_ros/supply_conveyor_state.h>
#include <assembly_control_ros/supply_conveyor_command.h>
#include <assembly_control_ros/supply_conveyor_input.h>
#include <assembly_control_ros/supply_conveyor_output.h>

#include <common/machine_controller.hpp>

class SupplyConveyor
    : public MachineController<assembly_control_ros::supply_conveyor_state,
                               assembly_control_ros::supply_conveyor_input,
                               assembly_control_ros::supply_conveyor_command,
                               assembly_control_ros::supply_conveyor_output> {
public:
    SupplyConveyor(ros::NodeHandle node)
        : MachineController(node, "supply_conveyor"), state_(State::On) {
    }

    virtual void process() override {
        assembly_control_ros::supply_conveyor_command commands;
        assembly_control_ros::supply_conveyor_output outputs;

        auto& inputs = getInputs();

        switch (state_) {
        case State::On:
            commands.on = true;
            if (getState().optical_barrier) {
                outputs.part_available = true;
                sendOuputs(outputs);

                ROS_INFO("[SupplyConveyor] Off");
                state_ = State::Off;
            }
            break;
        case State::Off:
            if (inputs.restart) {
                inputs.restart = false;

                ROS_INFO("[SupplyConveyor] On");
                state_ = State::On;
            }
            break;
        }

        sendCommands(commands);
    }

private:
    enum class State { On, Off };

    State state_;
};

int main(int argc, char* argv[]) {
    ros::init(argc, argv, "supply_conveyor");

    ros::NodeHandle node;

    ros::Rate loop_rate(50); // 50 Hz

    SupplyConveyor conveyor(node);

    while (ros::ok()) {
        conveyor.process();

        ros::spinOnce();

        loop_rate.sleep();
    }
}
```

s

Figure : Supply conveyor node

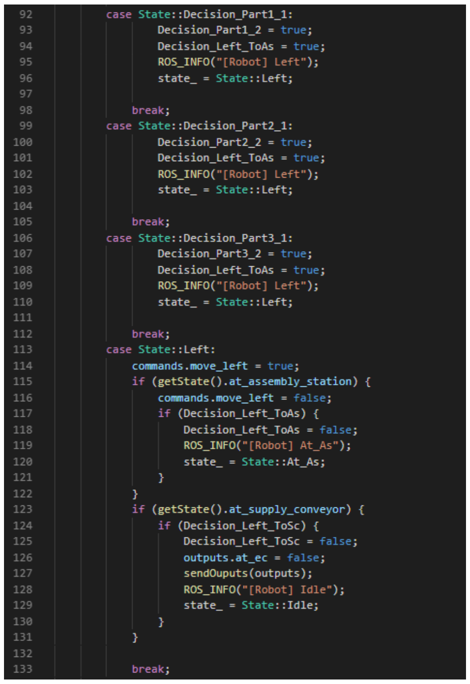
The figure above is an example for a machine node. We create a switch statement to describe the states (places of the petri net) according to our model. Since the machine is subscribed to its inputs, we can check the state of an input and continue to a new state when an input becomes active. Same thing applies also for the state messages. The machine can also publish on its outputs according to the petri model. Same thing applies for the command messages which are necessary for the V-rep simulation.



```
bool part1_exist = false;
bool part2_exist = false;
bool part3_exist = false;
bool Decision_Right_ToSc = false;
bool Decision_Right_ToEc = false;
bool Decision_Left_ToAs = false;
bool Decision_Left_ToSc = false;
bool Decision_Part1_2 = false;
bool Decision_Part2_2 = false;
bool Decision_Part3_2 = false;
bool at_as_begin = false;
int part_count = 0;
```

Figure : Variables instead of cases

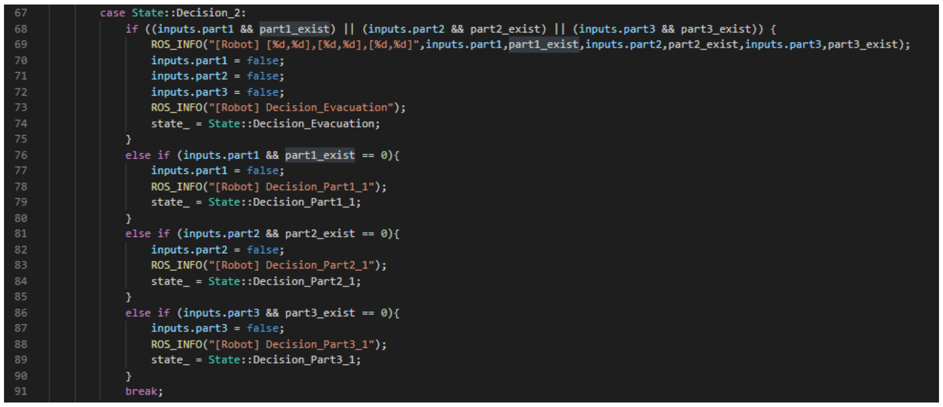
Sometimes, we use a state of the petri net model only to force the machine to go in the desired direction, or to prevent having multiple choices. For these situations, we use variables instead of inserting another case to the switch statement. This helps keep the code much cleaner.



```
case State::Decision_Part1_1:
        Decision_Part1_2 = true;
        Decision_Left_ToAs = true;
        ROS_INFO("[Robot] Left");
        state_ = State::Left;

    break;
case State::Decision_Part2_1:
        Decision_Part2_2 = true;
        Decision_Left_ToAs = true;
        ROS_INFO("[Robot] Left");
        state_ = State::Left;

    break;
case State::Decision_Part3_1:
        Decision_Part3_2 = true;
        Decision_Left_ToAs = true;
        ROS_INFO("[Robot] Left");
        state_ = State::Left;

    break;
case State::Left:
    commands.move_left = true;
    if (getState().at_assembly_station) {
        commands.move_left = false;
        if (Decision_Left_ToAs) {
            Decision_Left_ToAs = false;
            ROS_INFO("[Robot] At_As");
            state_ = State::At_As;
        }
    }
    if (getState().at_supply_conveyor) {
        if (Decision_Left_ToSc) {
            Decision_Left_ToSc = false;
            outputs.at_ec = false;
            sendOuputs(outputs);
            ROS_INFO("[Robot] Idle");
            state_ = State::Idle;
        }
    }

    break;
```

Figure : Example 1 of using variables in the Robot node

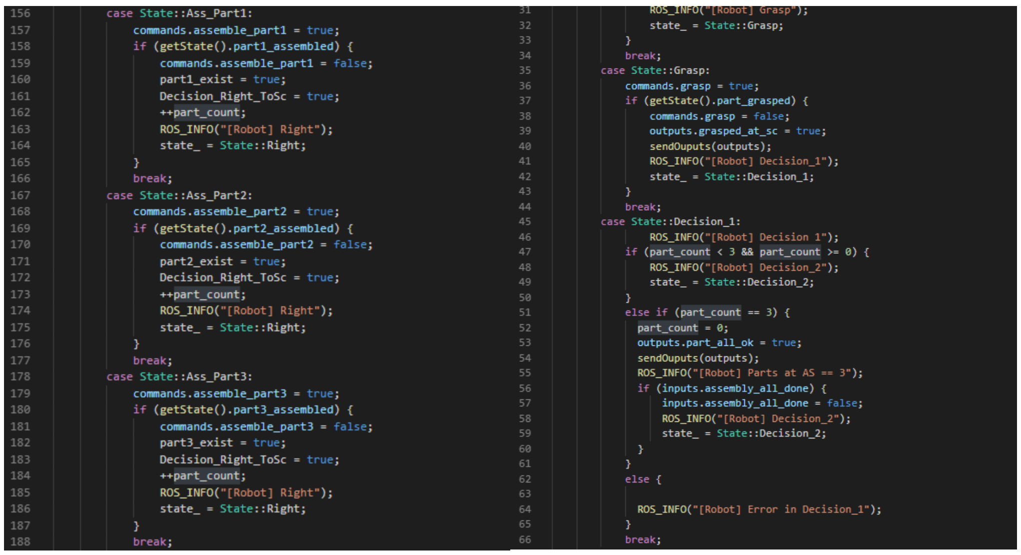
In the example above, the boolean variables that starts with “Decision_PartX_2” are used to stop the robot at the right station



```cpp
case State::Decision_2:
    if ((inputs.part1 && part1_exist) || (inputs.part2 && part2_exist) || (inputs.part3 && part3_exist)) {
        ROS_INFO("[Robot] [%d,%d],[%d,%d],[%d,%d]",inputs.part1,part1_exist,inputs.part2,part2_exist,inputs.part3,part3_exist);
        inputs.part1 = false;
        inputs.part2 = false;
        inputs.part3 = false;
        ROS_INFO("[Robot] Decision_Evacuation");
        state_ = State::Decision_Evacuation;
    }
    else if (inputs.part1 && part1_exist == 0){
        inputs.part1 = false;
        ROS_INFO("[Robot] Decision_Part1_1");
        state_ = State::Decision_Part1_1;
    }
    else if (inputs.part2 && part2_exist == 0){
        inputs.part2 = false;
        ROS_INFO("[Robot] Decision_Part2_1");
        state_ = State::Decision_Part2_1;
    }
    else if (inputs.part3 && part3_exist == 0){
        inputs.part3 = false;
        ROS_INFO("[Robot] Decision_Part3_1");
        state_ = State::Decision_Part3_1;
    }
    break;
```

Figure : Example 2 of using variables in the Robot node

In the second example, the boolean variables that starts with “PartX_exist” are used as a memory cell to represent an existing part at the assembly station. We use these variables to decide whether or not to evacuate the part. We reset these variables at the beginning of each assembly cycle.



```cpp
case State::Ass_Part1:
    commands.assemble_part1 = true;
    if (getState().part1_assembled) {
        commands.assemble_part1 = false;
        part1_exist = true;
        Decision_Right_ToSc = true;
        ++part_count;
        ROS_INFO("[Robot] Right");
        state_ = State::Right;
    }
    break;
case State::Ass_Part2:
    commands.assemble_part2 = true;
    if (getState().part2_assembled) {
        commands.assemble_part2 = false;
        part2_exist = true;
        Decision_Right_ToSc = true;
        ++part_count;
        ROS_INFO("[Robot] Right");
        state_ = State::Right;
    }
    break;
case State::Ass_Part3:
    commands.assemble_part3 = true;
    if (getState().part3_assembled) {
        commands.assemble_part3 = false;
        part3_exist = true;
        Decision_Right_ToSc = true;
        ++part_count;
        ROS_INFO("[Robot] Right");
        state_ = State::Right;
    }
    break;
```

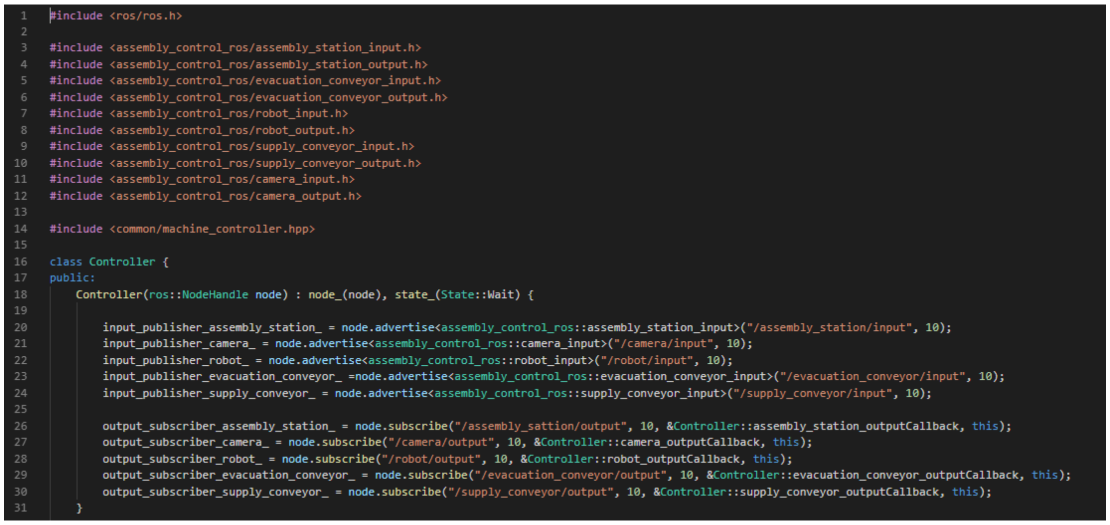
```cpp
            ROS_INFO("[Robot] Grasp");
            state_ = State::Grasp;
        }
        break;
    case State::Grasp:
        commands.grasp = true;
        if (getState().part_grasped) {
            commands.grasp = false;
            outputs.grasped_at_sc = true;
            sendOuputs(outputs);
            ROS_INFO("[Robot] Decision_1");
            state_ = State::Decision_1;
        }
        break;
    case State::Decision_1:
            ROS_INFO("[Robot] Decision 1");
        if (part_count < 3 && part_count >= 0) {
            ROS_INFO("[Robot] Decision_2");
            state_ = State::Decision_2;
        }
        else if (part_count == 3) {
          part_count = 0;
          outputs.part_all_ok = true;
          sendOuputs(outputs);
          ROS_INFO("[Robot] Parts at AS == 3");
          if (inputs.assembly_all_done) {
              inputs.assembly_all_done = false;
              ROS_INFO("[Robot] Decision_2");
              state_ = State::Decision_2;
          }
        }
        else {

          ROS_INFO("[Robot] Error in Decision_1");
        }
        break;
```

Figure : Example 3 of using variables in the Robot node

In the third example, the integer variable “part_count” is used to store the number of parts at the assembly station. We use that variable to decide whether or not to continue to the next assembly cycle. Once there are 3 parts at the assembly station, we publish an output and reset the “part_count”.

Controller node

The only role of the controller node is to connect the right outputs with the right inputs.

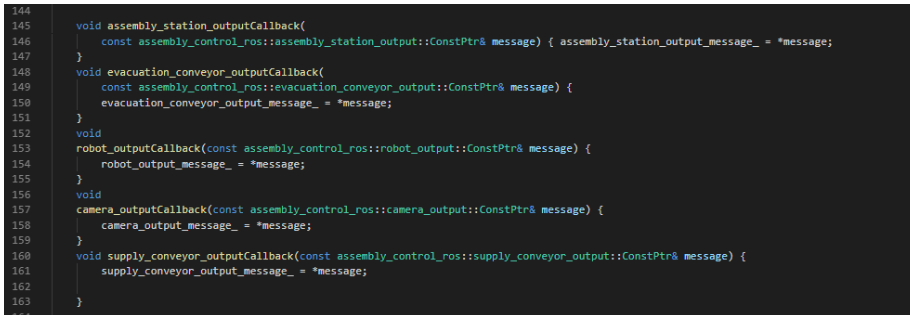


```cpp
#include <ros/ros.h>

#include <assembly_control_ros/assembly_station_input.h>
#include <assembly_control_ros/assembly_station_output.h>
#include <assembly_control_ros/evacuation_conveyor_input.h>
#include <assembly_control_ros/evacuation_conveyor_output.h>
#include <assembly_control_ros/robot_input.h>
#include <assembly_control_ros/robot_output.h>
#include <assembly_control_ros/supply_conveyor_input.h>
#include <assembly_control_ros/supply_conveyor_output.h>
#include <assembly_control_ros/camera_input.h>
#include <assembly_control_ros/camera_output.h>

#include <common/machine_controller.hpp>

class Controller {
public:
    Controller(ros::NodeHandle node) : node_(node), state_(State::Wait) {

        input_publisher_assembly_station_ = node.advertise<assembly_control_ros::assembly_station_input>("/assembly_station/input", 10);
        input_publisher_camera_ = node.advertise<assembly_control_ros::camera_input>("/camera/input", 10);
        input_publisher_robot_ = node.advertise<assembly_control_ros::robot_input>("/robot/input", 10);
        input_publisher_evacuation_conveyor_ =node.advertise<assembly_control_ros::evacuation_conveyor_input>("/evacuation_conveyor/input", 10);
        input_publisher_supply_conveyor_ = node.advertise<assembly_control_ros::supply_conveyor_input>("/supply_conveyor/input", 10);

        output_subscriber_assembly_station_ = node.subscribe("/assembly_sattion/output", 10, &Controller::assembly_station_outputCallback, this);
        output_subscriber_camera_ = node.subscribe("/camera/output", 10, &Controller::camera_outputCallback, this);
        output_subscriber_robot_ = node.subscribe("/robot/output", 10, &Controller::robot_outputCallback, this);
        output_subscriber_evacuation_conveyor_ = node.subscribe("/evacuation_conveyor/output", 10, &Controller::evacuation_conveyor_outputCallback, this);
        output_subscriber_supply_conveyor_ = node.subscribe("/supply_conveyor/output", 10, &Controller::supply_conveyor_outputCallback, this);
    }
```

Figure : Added inputs, outputs, subscribers and publishers

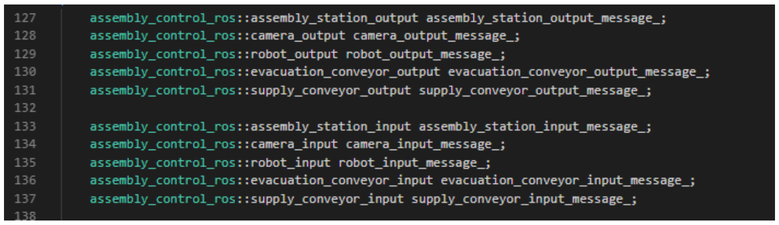
To achieve that, the controller must have access to all the inputs and outputs. Then, since will bring an output and pass to an input, it should subscribe to all the outputs and publish on all the inputs.

To implement those ideas, we add all the input and output messages to the controller node and declare a variable for each message type. Then we write callback functions for each output and create manually a subscriber and publisher for each node.



```cpp

    void assembly_station_outputCallback(
        const assembly_control_ros::assembly_station_output::ConstPtr& message) { assembly_station_output_message_ = *message;
    }
    void evacuation_conveyor_outputCallback(
        const assembly_control_ros::evacuation_conveyor_output::ConstPtr& message) {
        evacuation_conveyor_output_message_ = *message;
    }
    void
    robot_outputCallback(const assembly_control_ros::robot_output::ConstPtr& message) {
        robot_output_message_ = *message;
    }
    void
    camera_outputCallback(const assembly_control_ros::camera_output::ConstPtr& message) {
        camera_output_message_ = *message;
    }
    void supply_conveyor_outputCallback(const assembly_control_ros::supply_conveyor_output::ConstPtr& message) {
        supply_conveyor_output_message_ = *message;

    }
```

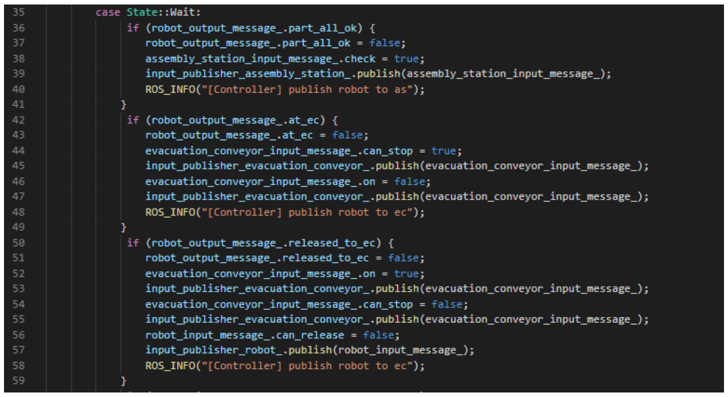
Figure : Callback functions



```
assembly_control_ros::assembly_station_output assembly_station_output_message_;
assembly_control_ros::camera_output camera_output_message_;
assembly_control_ros::robot_output robot_output_message_;
assembly_control_ros::evacuation_conveyor_output evacuation_conveyor_output_message_;
assembly_control_ros::supply_conveyor_output supply_conveyor_output_message_;

assembly_control_ros::assembly_station_input assembly_station_input_message_;
assembly_control_ros::camera_input camera_input_message_;
assembly_control_ros::robot_input robot_input_message_;
assembly_control_ros::evacuation_conveyor_input evacuation_conveyor_input_message_;
assembly_control_ros::supply_conveyor_input supply_conveyor_input_message_;
```

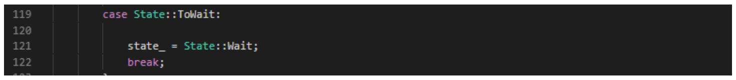
Figure : Declaration of all the message types

The controller has 2 states in its switch statement. The first state called "Wait" and it connects all the outputs with the correct input according to our model. When it receives an output, it publishes on the corresponding input. The only job of the second state "To wait" is to resend the controller to the "Wait" state.



```
        case State::Wait:
            if (robot_output_message_.part_all_ok) {
                robot_output_message_.part_all_ok = false;
                assembly_station_input_message_.check = true;
                input_publisher_assembly_station_.publish(assembly_station_input_message_);
                ROS_INFO("[Controller] publish robot to as");
            }
            if (robot_output_message_.at_ec) {
                robot_output_message_.at_ec = false;
                evacuation_conveyor_input_message_.can_stop = true;
                input_publisher_evacuation_conveyor_.publish(evacuation_conveyor_input_message_);
                evacuation_conveyor_input_message_.on = false;
                input_publisher_evacuation_conveyor_.publish(evacuation_conveyor_input_message_);
                ROS_INFO("[Controller] publish robot to ec");
            }
            if (robot_output_message_.released_to_ec) {
                robot_output_message_.released_to_ec = false;
                evacuation_conveyor_input_message_.on = true;
                input_publisher_evacuation_conveyor_.publish(evacuation_conveyor_input_message_);
                evacuation_conveyor_input_message_.can_stop = false;
                input_publisher_evacuation_conveyor_.publish(evacuation_conveyor_input_message_);
                robot_input_message_.can_release = false;
                input_publisher_robot_.publish(robot_input_message_);
                ROS_INFO("[Controller] publish robot to ec");
            }
```

Figure : Wait state

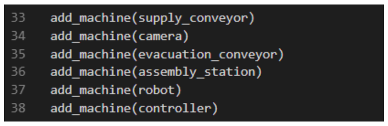


```
        case State::ToWait:

            state_ = State::Wait;
            break;
```

Figure : To wait state

Compiling

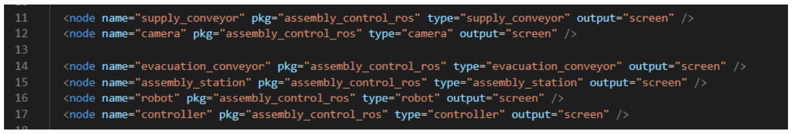
The project compiles using catkin. We add all the nodes to the CmakeList for the catkin to work properly.



```
add_machine(supply_conveyor)
add_machine(camera)
add_machine(evacuation_conveyor)
add_machine(assembly_station)
add_machine(robot)
add_machine(controller)
```

Figure : CmakeList

To avoid typing all the nodes to launch the entire program, we add all the nodes to the launchfile.



```
<node name="supply_conveyor" pkg="assembly_control_ros" type="supply_conveyor" output="screen" />
<node name="camera" pkg="assembly_control_ros" type="camera" output="screen" />

<node name="evacuation_conveyor" pkg="assembly_control_ros" type="evacuation_conveyor" output="screen" />
<node name="assembly_station" pkg="assembly_control_ros" type="assembly_station" output="screen" />
<node name="robot" pkg="assembly_control_ros" type="robot" output="screen" />
<node name="controller" pkg="assembly_control_ros" type="controller" output="screen" />
```

Figure : Launch file

Simulation

The connection between the ROS and the V-rep simulator has been already done at the beginning of the project. All we have to do is to launch V-rep then execute the launchfile of the project.

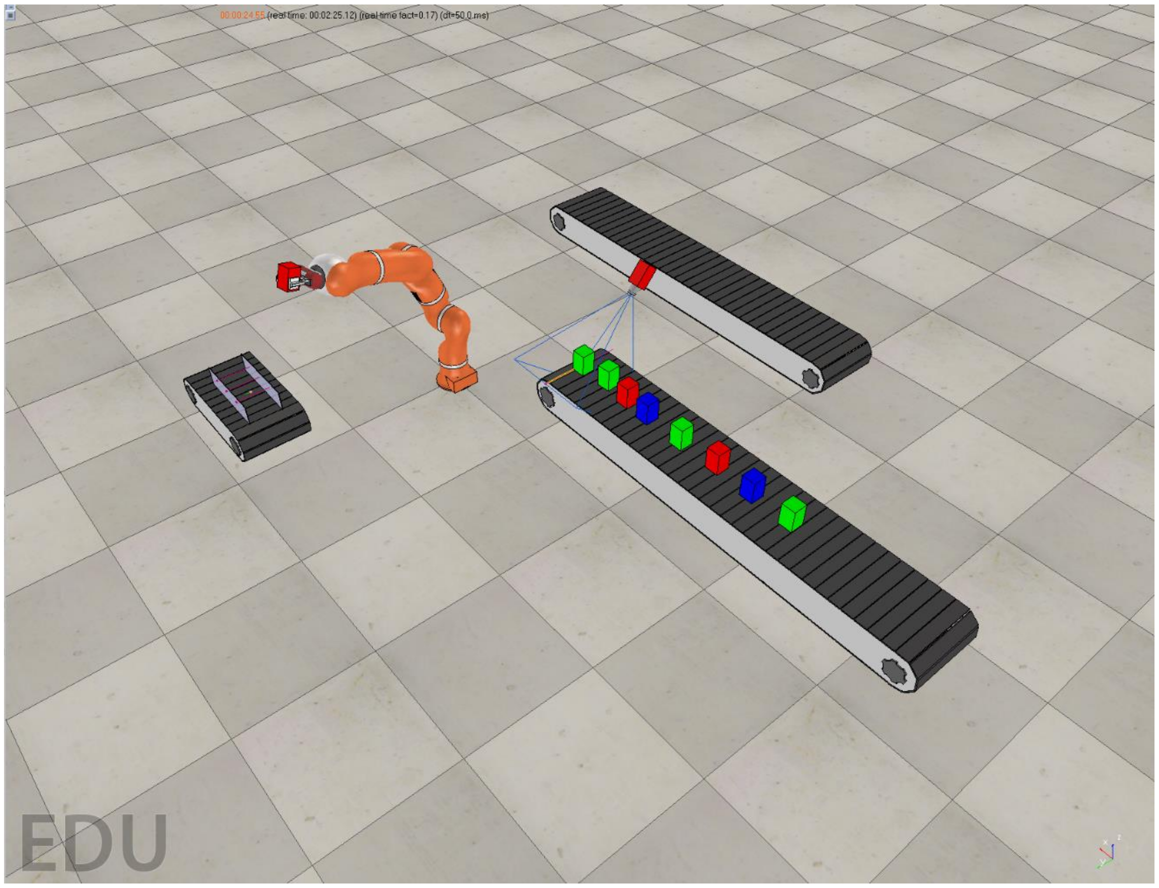

00:00:24.55 (real time: 00:02:25.12) (real time fact=0.17) (dt=50.0 ms)
EDU

Figure : V-Rep scene

Conclusion

During this project, we have learned about the modelling process of a given system. We have developed skills in the Petri nets and ROS. Also, we have seen the practical applications of programming with C++ and catkin.

These tools are very popular at the moment by the robotics companies around the world. For that reason I think that this project was in fact very useful for our future.