How to use clfsm with ROS-2

Vladimir Estivill-Castro
MiPal

August 11, 2024

Abstract

This document gets you started on using `clfsm` with ROS-2 (a similar document exists for ROS-1). It can be used as a tutorial to gain an understanding of basic behaviours defined with logic-labelled finite-state machines (*llfsms*). More sophisticated examples, like machines and submachines that are suspended and restarted are possible, but this is a beginners guide.

Contents

1 Examples of logic-labelled finite-state machines using `clfsm`

To help you use `clfsm` with ROS-2 we have four examples. You can see the first two examples running with ROS-1 in the short video www.youtube.com/watch?v=AJYA2hB4i9U&feature=youtu.be.

1. The first *llfsm* is a simple machine named ROS2_PING_PONG.MACHINE that publishes ROS:`messages`. We deswcribe how to build it and compile it appears in Subsection 2.1. Thus, the machine RosTwoPingPongMachine is just a publisher. It loops between two states:

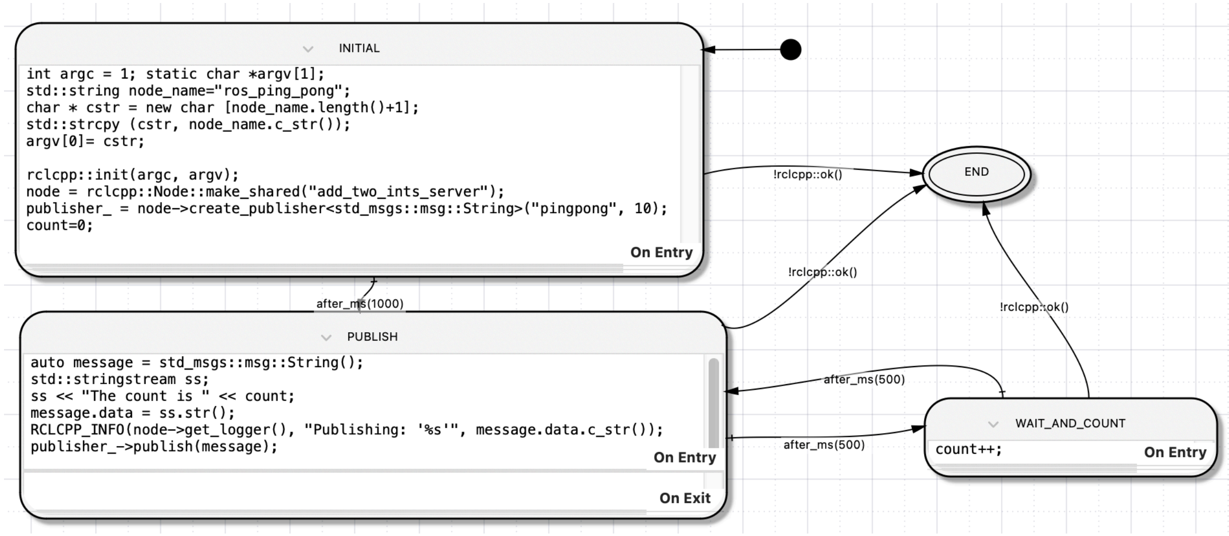

Figure 1: ROS2_PING_PONG.MACHINE is a simple *llfsm*. If you have completed the ROS tutorials and understand the semantics of *llfsms* you should be able to describe what it does before you actually run it.

  `WAIT_AND_COUNT` : where we see the variable `count` is incremented,

  `PUBLISH` that actually performs the publishing.

  Figure 1 shows a picture of this machine.

2. The second machine ROS2_BLIND_TURTLE_BOT.MACHINE is a machine that actually instructs ROS's `turtlesim` to walk around. This is a simple behaviour but has no use of sensors. The behaviour controls actuators but does not collect any information from the environment. It does so by publishing control messages to ROS's `turtlesim`. Subsection 2.4 describes building and running ROS2_BLIND_TURTLE_BOT.MACHINE.

3. The third example is named ROS2_WALL_TURTLE_BOT.MACHINE and it does control the ROS's `turtlesim` with a reactive behaviour. The idea is that the turtle walks straight until it is too close to the border of the simulation. When it gets too close, it drives back for a bit, and turns for a bit. After that, it returns to the moving-forward state. A short video illustrating this behaviour with ROS-1 is available at youtu.be/4txscEXN8lQ. Subsection 2.5 describes building and running ROS2_WALL_TURTLE_BOT.MACHINE.

4. In the fourth example we illustrate an arrangement of *llfsms* where two machines execute concurrently (in fact in a sequential schedule), and one suspends and resumes the other.

1.1 The setup

This section assumes you have read the first 6 sections of the "How to use" document for MiEditLLFSM". This document should provide you further understanding of the structure of *MiPal*'s *llfsms*(MiEditLLFSM is a tool to build *llfsms* such as the ones used here; it can be downloaded form mipal.net.au/downloads.php, and the "How to use" document for MiEditLLFSM"; from its Section 7 it describes *llfsms* for `Webots` or the Nao robot that are not needed now and instead of ROS they use the *MiPal* whiteboard infrastructure). MiEditLLFSM and the demonstration machines have been tested on many version of Ubuntu and ROS, as far back as Ubuntu 14.04-64 bits and ROS-Indigo. They were once tested on MacOS-Mavericks and ROS-hydro. Also, we assume you have successfully installed ROS-2 and completed the beginners ROS tutorials. We have successfully ran this examples with **Ubuntu 24.04 64-bit (jammy) and ROS-2 humble** (moreover, we have tested this in a native Intel machine, where the OS is identified as Linux-x86_64, and on a Mac with M1 and running UTM, in which case the OS is identified as Linux-aarch64. You can check your version of Ubuntu with the command

```
lib_release -a
```

Thus, you should have a ROS-2 `colcon` workspace and be able to build ROS modules like in the tutorials with

```
colcon build --packages-select <package_name>
```

Remember that a ROS-2 `colcon` *workspace* is nothing more than a directory with a sub-directory named `src` and all packages are placed inside `src`. However, the build is performed just inside the work package.

The *MiPal llfsms* are executed by a scheduler named `clfsm`. For ROS-2 we have produced a version that does not use the *MiPal* whiteboard infrastructure (named `gusimplewhiteboard`) and thus, does not depend on `libdispatch`.

The ROS-2 `colcon` workspace with sources of `clfsm`, and the library `libclfsm` are provided in the *MiPal* downloads as
`plain_clfsm_ws.tar.gz`.
For illustration, we assume you will work on a workspace such as `ros2_ws` under your home directory. Such a workspace is described as per the ROS-2 tutorials (Creating a workspace). However, you can use a new or different workspace. Place the code in an accessible directory; here we place in our `$HOME` directory, but you could leave it in your `Downloads` directory. We will be copying the packages out. But you can also use this workspace as a place to work:

```
mv $HOME/Downloads/plain_clfsm_ws.tar.gz $HOME
```

You should extract the files with

```
 gunzip plain_clfsm_ws.tar.gz
tar -xvf plain_clfsm_ws.tar
```

then move the packages from one workspace to the one used for work.

```
mv plain_clfsm_ws/src/libclfsm/ ros2_ws/src/
mv plain_clfsm_ws/src/clfsm/ ros2_ws/src/
ls ros2_ws/src/
```

should produce

```
clfsm
libclfsm
```

You are now in a position to build them (but as we mentioned you could have built them in the workspace they came, just place yourself at the root of the workspace). We recommend to do one package at a time, because despite the dependencies, `colcon` launches parallel building tasks that not always conform to the dependencies.

```
cd ros2_ws
colcon build --packages-select libclfsm
colcon build --packages-select clfsm
```

If all building went correctly (despite many warnings), you should be able to see the dynamic library

```
ls install/libclfsm/lib
```

should show `liblibclfsm.so` and

```
ls install/clfsm/lib/clfsm
```

should show the executable `clfsm`.

2 Machines with the ROS-2 turtlesim

2.1 Building the machine ROS2_PING_PONG.MACHINE using MIEDITLLFSM

Please attempt to construct ROS2_PING_PONG.MACHINE using MIEDITLLFSM although this machine is provided in the *MiPal* ROS-2 downlaods in the file `clfsmRos2DemoMachines.tar.gz`. For the ROS2_PING_PONG.MACHINE machine (Figure 1) you need the following includes, in the `include` section of the machine.

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
#include "CLMacros.h"
```

On the other hand, the `variables` section of this machine are as follows.

```
//
//ros2_ping_pong_Variables.h
//
//Automatically created through MiEditCLFSM -- do not change manually!
//
int      count;   ///<
rclcpp::Publisher<std_msgs::msg::String>::SharedPtr      publisher_;      ///<
std::shared_ptr<rclcpp::Node>     node;     ///<
```

The machine ROS2_PING_PONG.MACHINE was built with MIEDITLLFSM (so we insist that is good practice to use MIEDITLLFSM). The INITIAL state is rather simple, it is just a set up. There is just code for the **OnEntry** section of this state. There is a bit of work handling C-string versus C++ strings. This code is C++ compatible.

```
int argc = 1; static char *argv[1];
std::string node_name="ros_ping_pong";
char * cstr = new char [node_name.length()+1];
std::strcpy (cstr, node_name.c_str());
argv[0]= cstr;

rclcpp::init(argc, argv);
node = rclcpp::Node::make_shared(node_name);
publisher_ = node->create_publisher<std_msgs::msg::String>("pingpong", 10);
count=0;
```

From INITIAL, we go to PUBLISH after one second; thus, the transition is `after_ms(1000)`[1]. The state PUBLISH also has code only for the **OnEntry** section

```
auto message = std_msgs::msg::String();
std::stringstream ss;
ss << "The count is " << count;
message.data = ss.str();
RCLCPP_INFO(node->get_logger(), "Publishing: '%s'", message.data.c_str());
publisher_->publish(message);
```

The PUBLISH state alternates with the WAIT_AND_COUNT state with transitions of half a second (500ms); that is, respective transitions `after_ms(500)`. The state WAIT_AND_COUNT only has a simple **OnEntry** section to increment the counter and to the also go back to PUBLISH after half a second.

```
count++;
```

[1]This is why we need the include `CLMacros.h`

There is an accepting final state called END. The transitions to it are the test that ROS has finished `!rclcpp::ok()`. Note that transitions with a common source state are sorted. In this machine the transitions to END are before other transitions for all the states (INITIAL, WAIT_AND_COUNT and WAIT_AND_COUNT) since as soon as ROS is not running we want the machine to reach the terminal state.

2.2 How to compile `clfsm` machines with `colcon`

The demonstration logic-labelled finite-state machines of `clfsm` are a series of files, and thus, each exmaple comes in a directory `<machine_name>.machine`. That is, they have an extension `.machine`. The editor MiEditLLFSM manages machines by opening directories with this name format.

We now explain the assistance *MiPal* provides so you can set a machine as a `colcon` package and compile them with the command

```
colcon build --packages-select <package_name>
```

Building executables in this manner will be familiar to you from ROS tutorials or projects. The instructions are detailed but generic for any *llfsm*. We use the first machine ROS2_PING_PONG.MACHINE for illustration but the process will be the same for the other machines.

We place each machine as a package. So we recommend that you do so in your `colcon` workspace. For example, for the ROS2_PING_PONG.MACHINE *llfsm* we recommend the following. Move to your workspace

```
cd $HOME/ros2_ws/src
```

2.2.1 Creating the package of the *llfsm* (`package.xml`)

You create the package for the machine like any other package creation for ROS2.

```
ros2 pkg create --build-type ament_cmake --license Apache-2.0 ros2_ping_pong --dependencies rclcpp std_msgs clfsm libclfsm
```

We recommend to keep the name of the package the same as the name of the machine.

You can view the content of the `package.xml` file from the workspace directory as follows.

```
more ros2_ping_pong/package.xml
```

You notice that if you now go an edit the file `package.xml` as suggested in the ROS tutorials to add a name and contact, you already find there are lines that indicate the required packages.

```
<depend>rclcpp</depend>
<depend>std_msgs</depend>
<depend>clfsm</depend>
<depend>libclfsm</depend>
```

Thus, in general, for another machine with name *machineName*, you create a package for the *llfsm* as follows (recall that packages always go inside the `src` of the work space).

```
cd $HOME/ros2_ws/src
ros2 pkg create --build-type ament_cmake --license Apache-2.0 machineName --dependencies rclcpp clfsm libclfsm
```

Note that the command `ros2 pkg create` above is you must supply `machineName`. In more elaborate *llfsms* where other packages participate, even more dependencies are indicated. It is essential you specify the dependencies at the time of package creation, or latter editing the file `package.xml` and adding lines of the form

```
<depend>package name the llfsm depends on</depend>
```

Place only necessary dependencies for compialtion. If executables are built, they can run and be tested with ROS instrionspection even if theyr are designed to run with other nodes (we do not recommend palcing other runs that are meant to execute simultaneously in the dependencies).

However, we said to you that the *MiPal* implementation of *llfsms* uses a directory *machineName*.machine

for all the source files. This has to be managed with `colcon` packages that separate include files in a directory `include` and source files that are in a directory `src`. Thus, the package for a *llfsms* will have the name of the machine as the name of the package and the following structure. For this first *llfsm* the structure is as follows.

```
ros2_ws
└─ src
   └─ ros2_ping_pong
      ├─ package.xml
      ├─ CMakeLists.txt
      ├─ include
      ├─ src
      └─ machine
         └─ ros2_ping_pong.machine
```

In general we propose that the package name is the same as the machine, and will look as follows.

```
workspace name
└─ src
   └─ package name (same as machine name)
      ├─ package.xml
      ├─ CMakeLists.txt
      ├─ include
      ├─ src
      └─ machine
         └─ machine name.machine
```

Thus, once the package is created the next thing is to create a `machine` directory that is sibling to the `src` and `include` directories of your package. Place you directory *machineName*`.machine` in there

```
cd $HOME/ros2_ws/src
cd machineName
mkdir -p machine
mv path_to_machine/machineName.machine machine
```

2.2.2 The instructions for building/compiling (`CMakeLists.txt`)

To restructure the `CMakeLists.txt` file (this file is the main input to `colcon` to create a series of `cmake` building tasks) we provide and assistance script.

Copy the assisting script `machine_colcon_setup.sh` (you can download `machine_colcon_setup.sh` from the *MiPal* downloads mipal.net.au/downloads.php) into the `machine` directory as well. Depending of where you download `machine_colcon_setup.sh`, in what follows, the first command may be different, and also, you may need to change its permissions to make it an executable script.

```
cp $HOME/Download/machine_colcon_setup.sh machine
cd machine
chmod ugo+x machine_colcon_setup.sh
```

Thus, now things look as follows (we are using green to show the new file).

```
ros2_ws
└─ src
   └─ ros2_ping_pong
      ├─ package.xml
      ├─ CMakeLists.txt
      ├─ include
```

```
    ├── src
    └── machine
        ├── machine_colcon_setup.sh
        └── ros2_ping_pong.machine
```

Since you just created the package, if you are inside `machine` the following commands will show you almost empty directories.

```
../src
../include
```

The exception is the `include` directory that would show another subdirectory with the name of the package (and thus of the machine), but this one on itself will be also empty.

Now, if you run the script

```
./machine_colcon_setup.sh MachineName.machine
```

and re-inspect the directories `include` and `source` you will see many files.

```
ros2_ws
├── install
└── src
    └── ros2_ping_pong
        ├── package.xml
        ├── CMakeLists.txt
        ├── include
        │   └── many files, usually .h
        ├── src
        │   └── many files, usually .mm
        └── machine
            ├── machine_colcon_setup.sh
            ├── ros2_ping_pong.machine
            └── Amet_Suggested_CMakeLists_ros2_ping_pong.txt
```

As illustrated above, the script creates a file `Amet_Suggested_CMakeLists_`*`machineName`*`.txt` that provides you with the hints of how to edit the `CMakeLists.txt` of the package of the machine in order to complete configuring it for `colcon`. The script inspects whether you are running MacOS or Ubuntu, and it will give you the suggestions so the resulting `colcon` package can be used in both.

The script is such that most of the time you can overwrite the file `CMakeLists.txt` of the package by `Amet_Suggested_CMakeLists_`*`machineName`*`.txt`.

```
cp Amet_Suggested_CMakeLists_ros2_ping_pong.txt ../CMakeLists.txt
```

However, the script uses information about dependencies form the `package.xml` file; thus, that is why it is essential the `package.xml` file be up to date about dependencies[2]

2.2.3 Changes to an *llfsm*

Many times you develop a *llfsm* in stages or there is a compilation error that needs fixing.

If you ever edit your *llfsm* with MiEditLLFSM making structural changes (add/delete a state or transition, or add variables or include files), you need to run the script `machine_colcon_setup.sh` again. This will update `Amet_Suggested_CMakeLists_`*`machineName`*`.txt` so additional files are set

[2]. For our running example this is not an issue, but for other machines with incompletely specified dependencies, simply replacing `CMakeLists.txt` with `Amet_Suggested_CMakeLists_`*`machineName`*`.txt` will not work. If you created the package with the correct dependencies, these will show in the form

```
find_package(a-dependency REQUIRED)
```

in your `CMakeLists.txt` file. Double check the dependencies in `Amet_Suggested_CMakeLists_`*`machineName`*`.txt` and add missing ones before you overwrite `CMakeLists.txt`.

up for compilation and linking, but note that **you need to overwrite** `CMakeLists.txt` again to update this file. If the changes are as simple as a syntax error on the code inside a state, it may be easier to not use MiEditLLFSM and find the file in *machineName*`.machine` by the name of the state (and whether it is in a transition or in a section of the state). You do not need to overwrite `CMakeLists.txt` but you need to run the script `machine_colcon_setup.sh` again so the `src` directory receives a fresh copy for compilation with your updates.

You also notice that in the structure above we have indicated a new directory `install`. This directory is usually constructed during a successful building process. However, the script constructs it, if it does not exist because it will create the path

`install/`*`package name`*`/lib/`*`package name`*`machine/`*`OS type`*

You probably need some expertise in `colcon` and on `cmake` to understand everything that goes on in the `CMakeLists.txt` file. Suffice to say that during building, `colcon` will create a `cmake` environment in a sibling directory named `build`. Then, each state of the *llfsm* is compiled separately and then brought together into a dynamic library. The building by `colcon` usually leaves the results in the sibling directory `include`.

2.2.4 Building (compiling) an *llfsm*

So we now proceed to compile.

```
$HOME/ros2_ws/
colcon build --packages-select ros2_ping_pong
```

This will produce your machine in

`$HOME/ros2_ws/install/`*`machineName`*`/lib/lib`*`machineName.some-Extension`*

where the extension depends on the operating system. Recall the script `machine_colcon_setup.sh` should have also created a directory

`$HOME/ros2_ws/install/`*`machineName`*`/lib/`*`machineName`*`.machine/`*`some-OS-description`*

2.3 How to run `clfsm` machines once built with `colcon`

You must copy the compiled machine `lib`*machineName*`.so` with path

`$HOME/ros2_ws/install/`*`machineName`*`/lib/lib`*`machineName`*`.so`

into the directory

`$HOME/ros2_ws/install/`*`machineName`*`/lib/`*`machineName`*`.machine/`*`some-OS-description`*

The compilation should have called the appropriate linker to create a dynamic library with the correct extension, thus in Ubuntu, this usually has a `lib` before the *machineName* and the extension ṡo. So, you should remove the `lib` part. With our running example, in a native Intel Linux, you should have

`$HOME/ros2_ws/install/ros2_ping_pong/lib/ros2_ping_pong.machine/Linux-x86_64/ros2_ping_pong.so`

or

`$HOME/ros2_ws/install/ros2_ping_pong/lib/ros2_ping_pong.machine/Linux-aarch64/ros2_ping_pong.so`

The impact of compilation and copying the dynamic library (**in Ubunut**) should result in a structure as indicated in red below.

```
ros2_ws
├── install/ros2_ping_pong/lib/ros2_ping_pong.machine/OS_type/ros2_ping_pong.so
└── src
    └── ros2_ping_pong
        ├── package.xml
        └── CMakeLists.txt
```

```
 |
 |__include
 |__src
 |__machine
     |__machine_colcon_setup.sh
     |__ros2_ping_pong.machine
```

In another terminal, inside your package you can run the machine. First, go to the workspace.

```
cd  /ros2_ws
```

Install the package

```
source install/setup.bash
```

Then you can run the executable of `clfsm` which you should have in

```
$HOME/ros2_ws/lib/clfsm/clfsm
```

providing as argument the directory

```
ros2 run clfsm clfsm install/lib/machineName.machine
```

You can explore how the machine is posting by investigating what topics are there, and then eavesdropping into the topic.

```
ros2 topic list
ros2 topic info /pingpong
ros2 topic echo /pingpong
```

Use `ctl-C` to terminate `clfsm`. Alternatively, `clfsm` offers and option to trace in the terminal the states it is going trough.

```
ros2 run clfsm clfsm -v install/lib/machineName.machine
```

2.4 A machine that controls actuators: ROS2_BLIND_TURTLE_BOT.MACHINE

Now we demonstrate sending messages so that a robot does something; that is, a very simple machine that control the walking behaviour of ROS:`turtlesim`. This appears in the second part of the video www.youtube.com/watch?v=AJYA2hB4i9U&feature=youtu.be and corresponds to the machine ROS2_BLIND_TURTLE_BOT.MACHINE. Figure 2 shows the schematics of it.

The behaviour is simple, walk straight for a bit, then turn for a bit, and repeat these two states.

Please also attempt to build the machine from scratch using MiEditLLFSM. All states have only **OnEntry** sections, except the END state, which is actually empty. The INITIAL state just initializes the necessary ROS node similarly to our previous machine.

```
int argc = 1; static char *argv[1];
std::string node_name="ros2_blind_turtle_bot";
char * cstr = new char [node_name.length()+1];
std::strcpy (cstr, node_name.c_str());
argv[0]= cstr;

rclcpp::init(argc, argv);
node = rclcpp::Node::make_shared(node_name);
publisher_ = node->create_publisher<geometry_msgs::msg::Twist>("/turtle1/cmd_vel", 10);

msg = geometry_msgs::msg::Twist();

msg.linear.x = 0.0;
msg.linear.y = 0.0;
msg.linear.z = 0.0;
msg.angular.x = 0.0;
msg.angular.y = 0.0;
msg.angular.z = 0.0;
```

However, it already posts a `geometry_msgs` message to halt the `turtlesim`.

You may wish to explore the ROS documentation for the ROS:`turtlesim` to understand better some of this values, although they should be somewhat understandable from their names. It also makes more sense if we describe the variables global to all states.

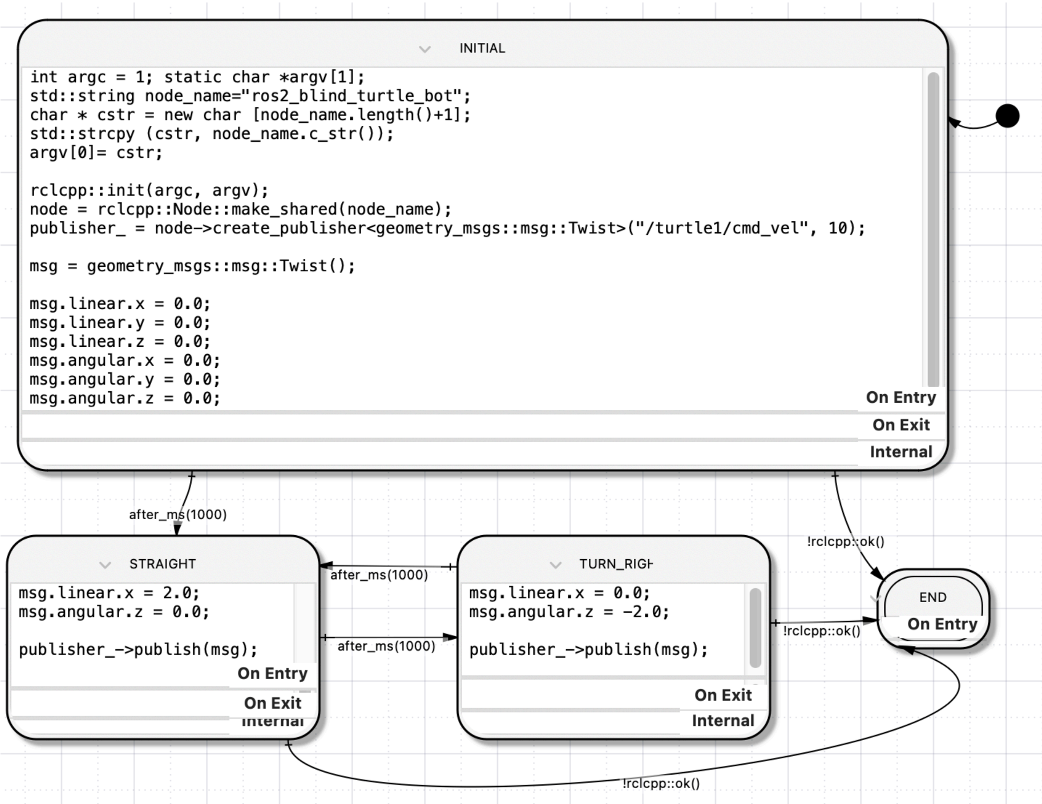

Figure 2: ROS2_BLIND_TURTLE_BOT.MACHINE is a simple *llfsm*. If you have completed the ROS tutorials and understand the semantics of *llfsms* you should also be able to see that this machine is a publisher, but now on the topic that sends control messages to the subscriber in ROS:turtlesim. to describe what it does before you actually run it.

```
//
//ros2_blind_turtle_bot_Variables.h
//
//Automatically created through MiEditCLFSM -- do not change manually!
//
rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr publisher_;     ///<
std::shared_ptr<rclcpp::Node>   node;    ///<
geometry_msgs::msg::Twist       msg;     ///<
```

And this data types from ROS require the corresponding `include` files.

```
#include "rclcpp/rclcpp.hpp"
#include "geometry_msgs/msg/twist.hpp"
#include "CLMacros.h"
```

The **OnEntry** part of state STRAIGHT sets up the linear speed of the X direction in the reference frame of the robot to a value greater than zero, making the turtle simulator walk the turtle straight. It publishes the corresponding message.

```
msg.linear.x = 2.0;
msg.angular.z = 0.0;

publisher_->publish(msg);
```

The **OnEntry** of the TURN_RIGHT state is very similar, but now is the angular seed that changes.

```
msg.linear.x = 0.0;
msg.angular.z = -2.0;

publisher_->publish(msg);
```

All transitions are of one second `after_ms(1000)` except the transitions to END which test if ROS is operational (`!ros::ok()`).

2.4.1 Summary of compiling and running ROS2_BLIND_TURTLE_BOT.MACHINE

You can compile and build the machine using the `colcon` package approach and the help of the script `machine_colcon_setup.sh` (see Section 2.2). Recall that the main steps are

1. Build a ROS-2 package indicating dependencies for `clfsm` and `libclfsm`

   ```
   ros2 pkg create --build-type ament_cmake --license Apache-2.0 ros2_blind_turtle_bot --dependencies rclcpp clfsm libclfsm geometry_msgs
   ```

   The create command here shows dependencies necessary for compilation, not for execution (that is why `turtlesim` is not shown. The program will run, an you can echo its output, even if `turtlesim` is not running.

2. Place the *llfsm* with extension `.machine` in a directory named `machine`

```
ros2_ws
└── src
    └── ros2_blind_turtle_bot
        ├── package.xml
        ├── CMakeLists.txt
        ├── include
        ├── src
        └── machine
```

```
├── machine_colcon_setup.sh
└── ros2_blind_turtle_bot.machine
```

that is sibling to the `src` and `include` of the package.

3. Use the script `machine_colcon_setup.sh` inside the directory `machine` to populate `include` and `src`

4. Generate and examine the suggested file

   `Amet_Suggested_CMakeLists_ros2_blind_turtle_bot.txt`

   which usually replaces `CMakeLists.txt` (unless there are mode dependencies)

   ```
   cd src/ros2_blind_turtle_bot/machine
   ./machine_colcon_setup.sh ros2_blind_turtle_bot.machine
   cp Amet_Suggested_CMakeLists_ros2_blind_turtle_bot.txt ../CMakeLists.txt
   ```

5. From the workspace root directory use `colcon` to build

   ```
   cd \$HOME/ros2_ws
   colcon build --packages-select ros2_blind_turtle_bot
   ```

6. Copy (or move) the resulting dynamic library further down in the `install` directory and into a path *MachineName.machine*/*OS-Descritpion*/*Machinename*.so. Here we use "\" to split the `bash` command on two lines.

   ```
   $ mv install/ros2_blind_turtle_bot/lib/libros2_blind_turtle_bot.so \
   > install/ros2_blind_turtle_bot/lib/ros2_blind_turtle_bot.machine/Linux-aarch64/ros2_blind_turtle_bot.so
   ```

7. Run the machine after installing the ROS-2 packages with `source instal/setup.bash` (and maybe other nodes like the `turtlesim`.

   ```
   ros2 run clfsm clfsm -v install/ros2_blind_turtle_bot/lib/ros2_blind_turtle_bot.machine
   ```

   You should observe the turtle making triangles as in the video www.youtube.com/watch?v=AJYA2hB4i9U&feature=youtu.be.

2.5 ROS2_WALL_TURTLE_BOT.MACHINE: A machine for reactive behaviour: sensors and actuators

ROS2_WALL_TURTLE_BOT.MACHINE is the third demonstration machine. It will require a bit more work, as with *llfsms*, we use an approach to messages that gives preference to the `get_Message` approach (a non-blocking polling approach), rather than a publisher-subscriber approach (where the subscriber is highly couple with the publisher, or messages will be lost). The publisher-subscriber also is demanding on resources because a call-back must be enacted for each messahed published. For some more discussion on this you can see the paper "High Performance Relaying of C++11 Objects Across Processes and Logic-Labeled Finite-State Machines" [3]. *llfsms* can be executed in time triggered fashion (even in microcontrolles [4]) and thus they are not event-driven, allowing them to be used for formal verification, even during run-time [2] and checking real-time properties [6]. Suffice to say we have two approaches to relay the messages from a sender to a receiver through some middleware (see Figure 3).

PUSH: (closer to event-driven) the receivers subscribe a call-back in the whiteboard. The posting of a message by the sender spans new threads in the receivers.

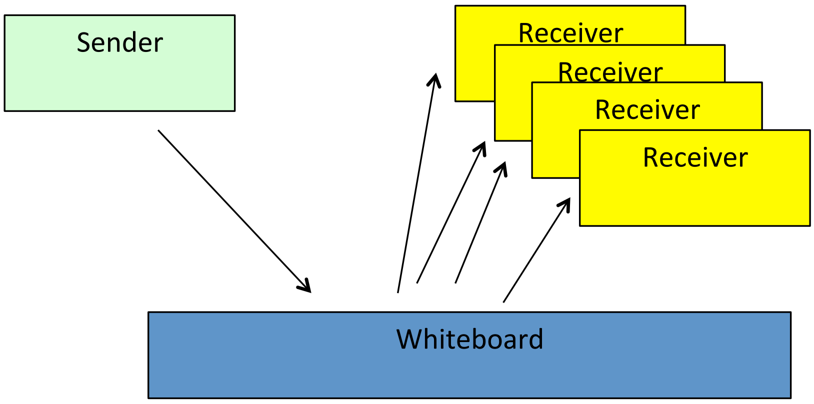

Figure 3: The role of some middleware (or whiteboard) is to simplify the APIs of communication between a sender of a message and the receivers.

PULL: (closer to time-triggered) receivers query the whiteboard for the latest from the sender. The receiver, in its own thread, retrieves the message. The sender, in its own thread, just adds messages.

From the perspective of software architectures, middleware provides the flexibility of a blackboard [5], which has also received names like *broker*. Thus, it is not surprising that this pattern has also emerged as the CORBA standard (of the Object Management Group, OMG) with the aim of facilitating communication on systems that are deployed on diverse platforms. In simple terms, these types of infrastructures enable a sender to issue what we will refer to as an `add_Message`($msg : T$) which is a non-blocking call. In a sense, posting msg to the middleware is simple. Such a posting may or may not include additional information, e.g. a sender signature, a timestamp, or an event counter that records the belief the sender has of the currency of the message. But when it comes to retrieving the message, there are essentially two modes.

`subscribe`(T, f)**:** In a preliminary step, the receiver subscribes to messages of a certain *type* T (of an implied *class*) and essentially goes to sleep. Subscription includes the name f of a function. The middleware will notify the receiver of the message msg every time someone posts for the given *class* T by invoking $f(msg)$ (usually queued in a *type* T specific thread). This is typically called PUSH technology.

`get_Message`(T)**:** The receiver issues a `get_Message` to the middleware that supplies the latest msg received so far for the *type* T. This is usually called PULL.

For example, ROS' PUSH technology names a communication channel, a `ROS::topic` (corresponding to what we call a named channel with a *type*). The modules posting or getting messages are called *nodes*. Posting a message in ROS is also called publishing. In fact, there is another mechanism for communication, called ROS-services, which is essentially a remote-procedure call, the requester/client invokes though the middleware a function and obtains a data structure as a response (or a failure signal) from a call-back in a server (we will construct a simple example in the next section) This server functionality is called a *service* in ROS. In ROS-1, the client would invoke the service with a blocking call. This leads to several issues, again, creating to much coupling between the client nad the server [1]. As of ROS-2, the it is recommended that the client invokes the service with what ROS-2 calls Synchronous vs. asynchronous service clients. The problem is that in *asynchronous services*, the client is not blocked but it is required to poll the readiness of a result. This seems to be still an active evolving aspect of ROS2 version, and it is unclear what the client can do while waiting for the response. Of more serious consequences is that in ROS-2, the scheduling of the client is forced to be under the thread management of ROS-2 Executors. These ROS-2 *Executors* have a complex semantics. Nevertheless, we are able to use the round-robin deterministic scheduler of *MiPal*'s `clfsm` here.

2.5.1 The sample ROS-service to publish the position of the turtle

Therefore, arrangements of *llfsms* have predefined schedules, that enable deterministic scheduling [7]. The execution of each state is a deterministic ringlet that evaluates the transition out of the current state and decides whether transitions T fires (which result of the execution of the **OnExit** section of the source state followed by the **OnEntry** section of the target state of T). If no transition fires, the running the **Internal** section completes the ringlet and the turn passes to the next *llfsms* in the arrangement in round-robin fashion.

For this model of execution, the machine that needs the sensor information cannot subscribe and enable call-backs, but it queries an *adapter* of the position of the ROS-2 `trutlesim`. Thus, this *adapter* will provide as a ROS-service the position of the turtle in the middleware enabled by ROS. The workspace `turtle_sensor_poster_ws` provides two packages.

turtle_sensor_interface : This provides a custom data type for the client-server relationship the *llfsm* is going to have with the posture of the `turtlesim`. The custom data type is in the file `TurtlePosition.srv`.

```
---
float64 x
float64 y
float64 theta
```

turtle_sensor_poster : This is indeed a ROS-2 node that is both, a subscriber (to the topic of the `turtlesim` that indicates the pose) and a service so it can answer request from clients about the pose of the `turtlesim` as a `TurtlePosition`[3].

With ROS-1 and `catkin`, it was easier to place and interface inside a package that was a node. It seems more difficult with `colcon`; that is why we have two packages. For compiling the *llfsms* only the interface would be a dependency.

The workspace `turtle_sensor_poster_ws` is distributed in the *MiPal* downloads for ROS-2. Familiarity with the tutorials for ROS:`services` will facilitate understanding what this does. Download the package `turtle_sensor_poster.tar.gz`. You can copy each package to the workspace where you are working (say `ros2_ws`) or you can leave them in their workspace.

Here, we show the steps to build and run in the workspace `turtle_sensor_poster_ws`. We recommend that you perform the build in order.

```
cd turtle_sensor_poster_ws
colcon build --packages-select turtle_sensor_interface
colcon build --packages-select turtle_sensor_poster
```

You can test this sensor-wrapper. Run the `turtlesim` in in another terminal.

```
ros2 run turtlesim turtlesim_node
```

Also in another terminal install the package and run it.

```
cd turtle_sensor_poster_ws
source install/setup.bash
ros2 run turtle_sensor_poster turtle_sensor_poster_node
```

You can test that is working using the ROS-2 tools (also in another terminal)

[3]We invite you to inspect the code, which places the call-back for the service outside the class (this is consistent with the ROS-2 tutorials 2 Write the service node. However, despite declaring it a `static` function in the class we could not get the compiler to accept the function when passed to the construction of the client. This was possible with ROS-1.

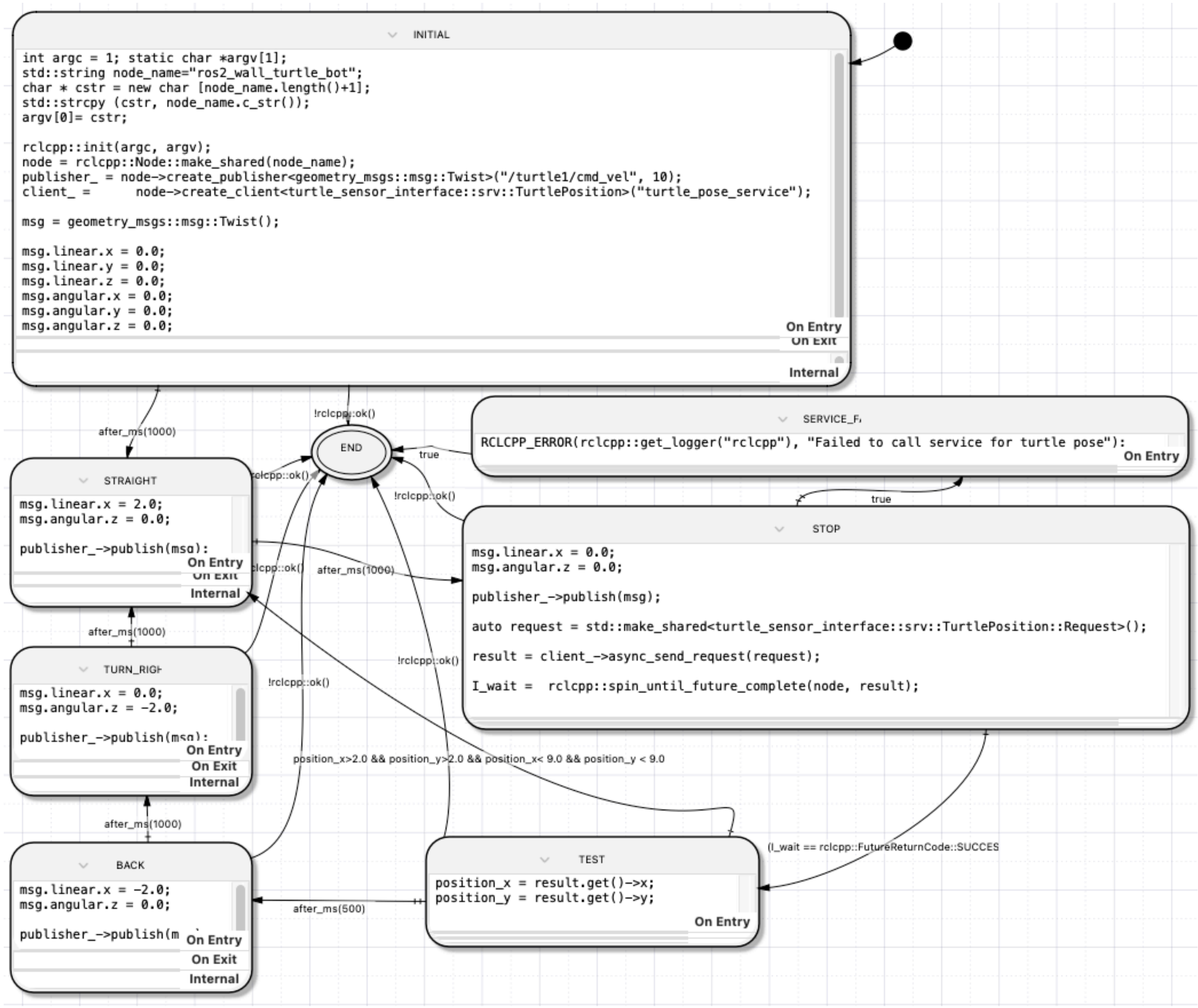

Figure 4: ROS2_WALL_TURTLE_BOT.MACHINE is a *llfsm* implementing a simple reactive behaviour. You can see this behaviour in action in the video youtu.be/4txscEXN8lQ.

```
ros2 service list
ros2 service type /turtle_pose_service
ros2 service call /turtle_pose_service turtle_sensor_interface/srv/TurtlePosition
```

You should see the (x, y) coordinates of the position of the turtle simulation. If you move the turtle and call the service again, the coordinates will be updated.

Just recall that, in the terminal where you would compile the *llfsm*, it is going to be necessary to install the interface.

2.5.2 The example *llfsm* where the turtle reacts to its position to the wall

This new machine (ROS2_WALL_TURTLE_BOT.MACHINE) appears in Figure 4. This *llfsm* will depend on the interface package `turtle_sensor_interface` because the code will be the client. The behaviour is sensing where the turtle is.

```
ros2 pkg create --build-type ament_cmake --license Apache-2.0 ros2_wall_turtle_bot --dependencies rclcpp clfsm libclfsm geometry_msgs turtle_sensor_interface
```

Recall it is crucial to establish the dependencies correctly, and in particular in the file `package.xml` of the package for the machine. The script that assist building the `CMakeLists.txt` file reads `package.xml`.

Thus, this machine requires the following includes.

```
#include "rclcpp/rclcpp.hpp"
#include "geometry_msgs/msg/twist.hpp"
#include "CLMacros.h"
#include "turtle_sensor_interface/srv/turtle_position.hpp"
```

Note that the `turtle_sensor_interface` is here so the *llfsm* can use calls to the corresponding service.

The *llfsm* uses thr following variables.

```
//
//ros2_wall_turtle_bot_Variables.h
//
//Automatically created through MiEditCLFSM -- do not change manually!
//
std::shared_ptr<rclcpp::Node>   node;   ///<
rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr publisher_;    ///<
geometry_msgs::msg::Twist       msg;    ///<
rclcpp::Client<turtle_sensor_interface::srv::TurtlePosition>::SharedPtr client_;       ///<
rclcpp::FutureReturnCode        I_wait; ///<
rclcpp::Client<turtle_sensor_interface::srv::TurtlePosition>::SharedFuture     result; ///<
float    position_x;     ///<
float    position_y;     ///<
```

The variable `client_` is the client service, and it used as demonstrated in the ROS-2 tutorials. Certainly different to ROS-1 is the variable `result` that in ROS-2 has the type

`rclcpp::Client<turtle_sensor_interface::srv::TurtlePosition>::SharedFuture`

since now we issue asynchronous calls. The variable `I_wait` is used to collect the status of the asynchronous call (with some opaque semantics since it is not clear what the client can do between the `sync_send_request` call and the `spin_until_future_complete` call[4].

This machine has been kept simple, because as per the ROS-2 tutorials the code should check the service is available and take some action (perhaps wait and try again several times). We avoid in this example using `wait_for_service(`*`timeout_sec`*`)`, since this blocks the client for as long as *timeout_sec* (or for ever).

Most of the states and transitions should not as surprising given the previous machine. In fact, the INITIAL is almost the same. The name of the node has changed and we have the the initialisation of the client object.

```
int argc = 1; static char *argv[1];
std::string node_name="ros2_wall_turtle_bot";
char * cstr = new char [node_name.length()+1];
std::strcpy (cstr, node_name.c_str());
argv[0]= cstr;

rclcpp::init(argc, argv);
node = rclcpp::Node::make_shared(node_name);
publisher_ = node->create_publisher<geometry_msgs::msg::Twist>("/turtle1/cmd_vel", 10);
client_ = node->create_client<turtle_sensor_interface::srv::TurtlePosition>("turtle_pose_service");

msg = geometry_msgs::msg::Twist();

msg.linear.x = 0.0;
msg.linear.y = 0.0;
msg.linear.z = 0.0;
msg.angular.x = 0.0;
msg.angular.y = 0.0;
msg.angular.z = 0.0;
```

States STRAIGHT and TURN_RIGHT are also just as before, and the state STOP just sets both speeds to zero.

[4]The semantics of `spin_until_future_complete` is opaque indicating some form of blocking.

```
msg.linear.x = 0.0;
msg.angular.z = 0.0;

publisher_->publish(msg);

auto request = std::make_shared<turtle_sensor_interface::srv::TurtlePosition::Request>();

result = client_->async_send_request(request);

 I_wait =  rclcpp::spin_until_future_complete(node, result) ;
```

The state BACK sets the linear forward/backwards speed of the turtle to a negative value (remember linear x is in the reference frame of the robot and is straight).

```
msg.linear.x = -2.0;
msg.angular.z = 0.0;

publisher_->publish(msg);
```

Thus, the only trick is in the state TEST, where the position of the turtle in the space is recuperated.

```
position_x = result.get()->x;
position_y = result.get()->y;
```

We arrive to this state after a successful retrieval of the data from the position service. That is, the transition between STOP and TEST is

```
(I_wait == rclcpp::FutureReturnCode::SUCCESS)
```

Note that if this transition fails then execution will end but first visiting the state StateSERVICE_FAILS. Read about ROS::`services` in the ROS tutorials if this is not clear.

The other interesting transition is the transition going out of TEST back to STRAIGHT.

```
position_x>2.0 && position_y>2.0 && position_x< 9.0 && position_y < 9.0
```

This checks that the recent read positions for the turtle are well within the [0,10]×[0,10] environment. Thus, when the position is central to the environment, the turtle goes back to another straight trajectory. Otherwise, after half a second, it performs the step-back (BACK) and turn (TURN_RIGHT)before going back to STRAIGHT.

2.5.3 Running the machine ROS2_WALL_TURTLE_BOT.MACHINE

Thus, we are almost ready to run ROS2_WALL_TURTLE_BOT.MACHINE. It is compiled the same way as the previous ones. You can use the approach of building a `colcon` package with the script `machine_colcon_setup.sh`.
However, in this case we depend on one more package, so create the package for the machine as follows.

```
ros2 pkg create --build-type ament_cmake --license Apache-2.0 ros2_wall_turtle_bot --dependencies rclcpp clfsm libclfsm geometry_msgs turtle_sensor_interface
```

This will create the necessary dependencies list in the file `package.xml`. Follow the same process as in Section 2.4.1; however, there is one more thing, we need to enable the compilation find the includes for `turtle_sensor_interface`. You will see that the script with create a suggested `CMakeLists.txt` that has a lines for

```
(find_package turtle_sensor_interface REQUIRED)
```

and

```
include_directories(${turtle_sensor_interface_INCLUDE_DIRS})
```

However, in the terminal where you are about to issue the `colcon build` command we recommend that you visit first the turtle_sensor_poster_ws workspace and do the following.

```
cd turtle_sensor_poster_ws
```

and the build and install

```
colcon build --packages-select turtle_sensor_interface
source install/setup.bash
```

Then go back to the workspace where the machines are.

```
cd $HOME/ros2_ws
```

and build here (after you ran the script and overwrote `CMakeLists.txt` with the suggested version.

```
colcon build --packages-select ros2_wall_turtle_bot
```

Them, the `colcon build`, as usual, should compile this machine. Now just do the placing of the result in

`install/ros2_wall_turtle_bot/lib/libiros2_wall_turtle_bot.`*`some-extension`*

to the target in `install/ros2_wall_turtle_bot/lib/`ROS2_WALL_TURTLE_BOT.MACHINE`.machine`. Recall, this is still inside a directory with the name of the OS-type and without the prefix `lib` and with the extension `.so` of it is Ubuntu.

```
mv install/ros2_wall_turtle_bot/lib/libros2_wall_turtle_bot.so install/ros2_wall_turtle_bot/l
```

Now, open several terminals. In one, run the turtle simulator.

```
ros2 run turtlesim turtlesim_node
```

On a second one, we run the service (we have not said where you downloaded this, but go to the workspace (the `cd` may require some prefix). Maybe build it, if already build just install.

```
cd turtle_sensor_poster_ws
colcon build --packages-select turtle_sensor_poster
source install/setup.bash
```

Now run.

```
ros2 run turtle_sensor_poster turtle_sensor_poster_node
```

Finally, the machine is executed in the third terminal.

```
cd $HOME/ros2_ws
source install/setup.bash
ros2 run clfsm clfsm install/ros2_wall_turtle_bot/lib/ros2_wall_turtle_not.machine
```

You should observe the behaviour as in the video youtu.be/4txscEXN8lQ.

2.6 A machine to suspend and re-start ROS2_WALL_TURTLE_BOT.MACHINE

Several *llfsms* can be executed concurrently in `clfsm`. When they are grouped this way they are called an *arrengement*. Also, they can be suspended, restarted and resumed. One example is ROS2_TURTLE_SUSPEND_RESUME.MACHINE. The diagram of the machine appears in Fig. 5. We emphasize that this machine makes use of

```
#include "CLMacros.h"
```

in its includes. This is important, observe the transitions like

```
is_suspended("RosBlindTurttleBot")
```

and

```
is_running("RosBlindTurttleBot")
```

. Also, we see the call in the state INITIAL to suspend the other machine

```
suspend("RosBlindTurttleBot");
```

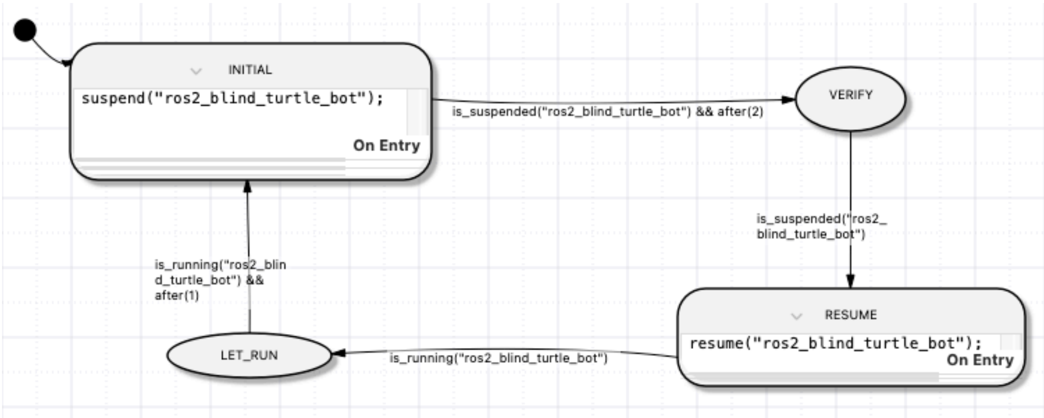

Figure 5: ROS2_TURTLE_SUSPEND_RESUME.MACHINE is a *llfsm* that suspends and starts ROS2_WALL_TURTLE_BOT.MACHINE.

While in the state RESUME

```
resume("RosBlindTurttleBot");
```

enables the other machine to continue. There are some important details about how scheduling with `clfsm` happens of the **OnEntry** and **OnExit** of *llfsms* under these utilities and in general for an arrangement of *llfsms*. The *llfsms* in the arrangement are executing in round-robin fashion, each machine having a turn to the token of execution of a ringlet of its current state. A ringlet is to execute the **OnEntry** section provided, execution is coming from another state, to evaluate all transitions out in sequence and if one fires, the **OnExit** runs and the ringlet stops here. If no transition fires the **Internal** section is executed and the ringlet stops.

If we are not coming from another state, the **OnEntry** does not get executed, the ringlet resumes from evaluating the sequence of transitions.

This *llfsm* shows that all machines have a state SUSPENDED, and that any execution of a ringlet in `clfsm` consists of checking if the machine with the token has been asked to be suspended. In that case, the machine performs a transition to the SUSPENDED state as if it were any other state. However, it will not get a turn on the round-robing until it moves out fo the SUSPENDED state. The `resume` sends the machine back to the state from which it was suspended and re-executes its **OnEntry** section. When suspended, a machine does not execute its **OnExit**. That is the only exception of what suspension causes to a machine.

There are some important aspects of the execution of arrangements of *llfsm* with `clfsm`. To use the features to suspend or resume (which means to go back to the state where the machine was suspended), `clfsm` does not handle full ir relative paths. The machine alone has to be in the command line. Therefore, to observe the effect of the suspend and resume you need to install in the workspace directory.

```
cd $ros2_ws
source install setup.bash
```

We assume you copied the produced linked libraries as indicated. Thus, you need to get into the directory where the machine that is to be installed is.

```
cd install/ros2_blind_turtle_bot/lib
```

Then you issue the run command with an absolute path for the first machine and a name only for the second one.

```
ros2 run clfsm clfsm -v ../../../install/ros2_turtle_suspend_resume/lib/ros2_turtle_suspend_r
ros2_blind_turtle_bot.machine
```

You should oberve the `turtlesim` perform triangles of double the length as if `ros2_blind_turtle_bot` was not suspended. That is, re-running with the following command

```
ros2 run clfsm clfsm -v ros2_blind_turtle_bot.machine
```

produces trajectories that are smaller triangles. This is because when a machine is sustpended and the resumes, it is as if a nes **OnEntry** is executed in the arrival back to the state where it was suspended. Thus, with the sustpended and resume, the `turlesim` receives double the comamnds to go straigth and turn.

References

[1] V. Estivill-Castro and R. Hexel. Simple, not simplistic - the middleware of behaviour models. In J. Filipe and Leszek A. Maciaszek, editors, *ENASE 2015 - Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering, Barcelona, Spain, 29-30 April, 2015*, pages 189–196. SciTePress, 2015.

[2] V. Estivill-Castro and R. Hexel. Run-time verification of regularly expressed behavioral properties in robotic systems with logic-labeled finite state machines. In *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots, SIMPAR*, pages 281–288. IEEE, December 13th-16th 2016.

[3] Vladimir Estivill-Castro, René Hexel, and Carl Lusty. High performance relaying of C++11 objects across processes and logic-labeled finite-state machines. In Davide Brugali, Jan F. Broenink, Torsten Kroeger, and Bruce A. MacDonald, editors, *Proceedings of the International Conference on Simulation, Modelling, and Programming for Autonomous Robots (SIMPAR 2014)*, Lecture Notes in Computer Science, vol 8810, pages 182–194, Cham, 2014. Springer International Publishing.

[4] F. Grubb, V. Estivill-Castro, and R. Hexel. LLFSMs on the PRU: Executable and verifiable software models on a real-time microcontroller. In L. Borzemski, editor, *28th Annual International Conference on Systems Engineering (ICSEng)*, LNNS, pages 391–402. Springer, December 14th-16th 2021.

[5] Barbara Hayes-Roth. A blackboard architecture for control. *Artificial intelligence*, 26(3):251–321, 1985.

[6] C. McColl, V. Estivill-Castro, M. McColl, and R. Hexel. Verifiable executable models for decomposable real-time systems. In L. Ferreira Pires, S. Hammoudi, and Seidewitz. e., editors, *Model-Driven Engineering and Software Development - 9th International Conference, MODELSWARD 2022*, pages 182–193. SCITEPRESS, February 6th-8th 2022.

[7] Callum McColl, Vladimir Estivill-Castro, Morgan McColl, and René Hexel. Decomposable and executable models for verification of real-time systems. In Luís Ferreira Pires, Slimane Hammoudi, and Edwin Seidewitz, editors, *Revised papers form Model-Driven Engineering and Software Development - 9th International Conference, MODELSWARD 2021*, volume 1708 of *Communications in Computer and Information Science*, pages 135–156. Springer, 2022.