How to use clfsm with ROS

Vladimir Estivill-Castro
MiPal

October 24, 2016

Abstract

This document gets you started on using clfsm with ROS. It can be used as a tutorial to gain an understanding of very basic behaviors defined with logic-labeled 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 we have four examples. You can see the first two examples in the video www.youtube.com/watch?v=AJYA2hB4i9U&feature=youtu.be.

1. The first *llfsm* is a simple machine that publishes ROS:messages. It has one state where we see the count, and another state that actually performs the publishing. Figure 1 shows a picture of this machine.

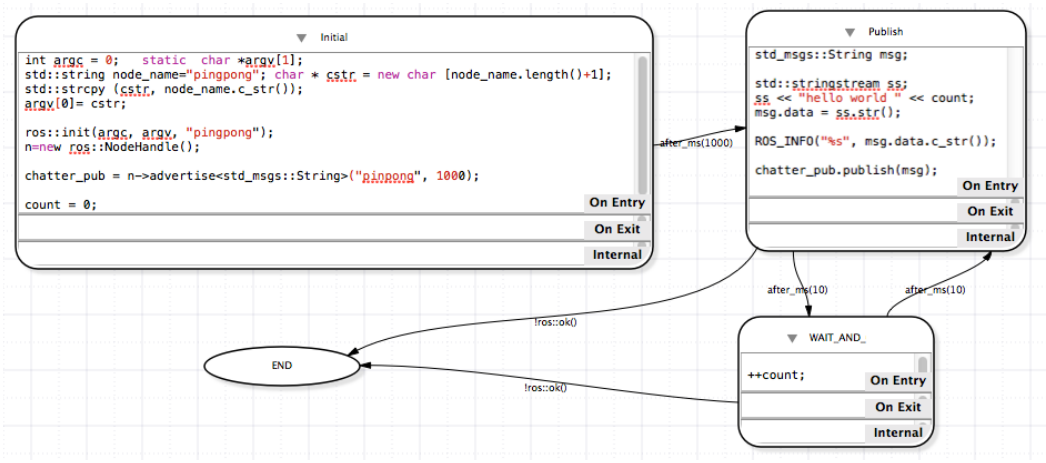

Figure 1: `RosPingPong` is a simple *llfsm*. If you have completed the `ROS` tutorials and understand the semantics of *llfsms* you should be able to describe what it does before you actually run it.

2. The second machine is a machine that actually instructs the TurtleBot to walk around. This is a simple behavior but has no use of sensors. The behavior controls actuators but does not collect any information from the environment.

3. The third example does control the TurtleBot with a reactive behavior. 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 video of this behavior is available at youtu.be/4txscEXN8lQ.

4. We can create an arrangement of *llfsms* where two machines execute concurrently (in fact in a sequential schedule), and one suspends and resumes the other.

You can use this document as a tutorial the material to follows and the goal is to make sure you can execute these 4 *llfsms*.

1.1 The setup

This section assumes you have read the current "How to use" of MIEDITLLFSM" so you have an understanding of the structure of *MiPal*'s *llfsms*. It has been tested on Ubuntu 14.04 64 bits and `ROS`-Indigo. Only the first machine has been tested on MacOS-Mavericks and `ROS`-hydro. Also, we assume you have successfully installed `ROS` (wiki.ros.org/indigo/Installation/Ubuntu) and completed the beginners `ROS` tutorials (http://wiki.ros.org/ROS/Tutorials). In particular, you should have a `catkin` workspace and be able to build `ROS` modules like in the tutorials with `catkin_make`.

Moreover, since we are not using *MiPal*'s "Getting Started" document, at least in Ubuntu you have to perform the following. For the `gusimplewhiteboard` module:

```
sudo apt-get install libdispatch-dev
```

The catkin workspace with sources of clfsm, and the other modules comes as clfsm.tar.bz2.
You should extract the files with

```
tar xjvf clfsm.tar.bz2
```

Place the code in the corresponding src directory of your catkin workspace. Thats is assuming you have a catkin_ws under your home directory as per the ROS tutorials, then

```
cd $HOME/catkin_ws
ls src
```

should produce

```
clfsm
CMakeLists.txt
gusimplewhiteboard
libclfsm
```

A catkin_make should compile all modules.

2 Machines with the ROS turtlesim

2.1 Building the machine RosPingPong using MIEDITLLFSM

Please attempt to construct RosPingPong using MIEDITLLFSM although is provided with the download instructions. For the RosPingPong machine (Figure 1) you need the following includes, in the include section of the machine.

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "CLMacros.h"

#include <sstream>
```

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

```
ros::NodeHandle* n
int count
ros::Publisher chatter_pub
```

The machine RosPingPong can be built with 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++11 strings. This code is C++11 compatible.

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

ros::init(argc, argv, "pingpong");
n=new ros::NodeHandle();

chatter_pub = n->advertise<std_msgs::String>("pinpong", 1000);

count = 0;
```

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

```
std_msgs::String msg;

std::stringstream ss;
ss << "hello world " << count;
msg.data = ss.str();

ROS_INFO("%s", msg.data.c_str());
chatter_pub.publish(msg);
```

The PUBLISH state alternates with the WAIT_AND_COUNT state with transitions of 10ms; that is, respective transitions `after_ms(10)`. The state WAIT_AND_COUNT only has a simple **OnEntry** section to increment the counter and to the ROS spin.

```
++count;
```

There is an accepting final state called END. The transitions to it are the test that ROS has finished `!ros::ok()`.

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

The logic-labeled finite-state machines of `clfsm` come in a directory `<machine_name>.machine`. That is, they have an extension `.machine`.

We are going to explain here the assistance we provide so you can set a machine as a `catkin` package and compile them with the command `catkin_make`. This will be more familiar to you if you are a ROS user.

We place each machine as a package. So we recommend that you do so in your `catkin` workspace. For example for the ROSPINGPONG *llfsm* we recommend the following.

```
cd $HOME/catkin_ws/src
catkin_create_pkg RosPingPong std_msgs roscpp clfsm libclfsm
```

You notice that if you now go an edit the file `package.xml` you already find there the required packages.

```
<build_depend>libclfsm</build_depend>
<build_depend>roscpp</build_depend>
<run_depend>clfsm</run_depend>
<run_depend>libclfsm</run_depend>
```

Thus, in general, you create a package for an *llfsm* as follows.

```
cd $HOME/catkin_ws/src
catkin_create_pkg machineName std_msgs roscpp clfsm libclfsm
```

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/catkin_ws/src
cd machineName
mkdir -p machine
mv machineName.machine machine
```

Copy the assisting script `machine_catkin_setup.sh`[1] into the `machine` directory as well. Depending of where you download `machine_catkin_setup.sh`, 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_catkin_setup.sh machine
cd machine
chmod ugo+x machine_catkin_setup.sh
./machine_catkin_setup.sh machineName.machine
```

The script will populate the `src` and `include` folders of your package. If you edit the machine, that is, if you modify states, change the code, or anything that varies the machine, you need to run the script again. You can check that you have files in `src` and `include` by executing the following commands inside the directory `machine`.

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

More importantly, the script creates a file `CMakeLists_machineName.txt` that provides you with a suggestion for the file `CMakeLists.txt` of the package of the machine in order to complete configuring it for `catkin`. The output highlights the main points and hints what are the issues in constructing the `CMakeLists.txt` for `catkin`. Although the script inspects whether you are running MacOS or Ubuntu, it will give you the suggestions so the resulting `catkin` package would be portable to both.

You can revise the suggestions in the corresponding sections, the output file

```
CMakeLists_machineName.txt
```

gives you an initial `CMakeLists.txt` and the output of the script indicates which sections require these additions. Some may not be necessary, like the hint

```
find_package(catkin REQUIRED COMPONENTS clfsm libclfsm)
```

This is just a remainder in case you forgot listing these two packages on creation of the package for the machine.

Important: You will need to check the hint

```
catkin_package( LIBRARIES libmachineName)
```

This means that the configuration includes the following aspects:

1. to uncomment the corresponding line
2. and the string `lib` before *machineName*.

It is necessary to do this because we need to separate the compilation of the source code of the machine itself, with the linking to produce the input for `clfsm`.

Also you must make sure that `catkin` knows to link against `libclfsm`. So you need to make sure the line

```
CATKIN_DEPENDS clfsm libclfsm roscpp std_msgs
```

is not commented. So at a minimum, that section should look something like this:

```
 catkin_package(
# INCLUDE_DIRS include
LIBRARIES libmachineName
CATKIN_DEPENDS clfsm libclfsm roscpp std_msgs
# DEPENDS system_lib )
```

[1] see the *MiPal* downloads mipal.net.au/downloads.php

You can use

`CMakeLists_`*`machineName`*`.txt`

as the file `CMakeLists.txt` for `catkin` to compile your package.

```
cd $HOME/catkin_ws/
catkin_make
```

This will produce your machine in

`$HOME/catkin_ws/devel/lib/lib`*`machineName`*`.`*`some-Extension`*

where the extension depends on the operating system. Moreover, the script
`machine_catkin_setup.sh`
should have also created a directory

`$HOME/catkin_ws/devel/lib/`*`machineName`*`.machine/`*`some-OS-description`*

You must copy the compiled machine in the file

`$HOME/catkin_ws/devel/lib/lib`*`machineName`*`.`*`some-Extension`*

into the directory

`$HOME/catkin_ws/devel/lib/`*`machineName`*`.machine/`*`some-OS-description`*

twice; once with the name

`$HOME/catkin_ws/devel/lib/`*`machineName`*`.machine/`*`some-OS-description/machineName`*

and also

`$HOME/catkin_ws/devel/lib/`*`machineName`*`.machine/`*`some-OS-description/machineName`*`.so`

2.3 How to run the machine

If all the previous steps succeed you should have an executable of `clfsm` in

`$HOME/catkin_ws/devel/lib/clfsm/clfsm`

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

`$HOME/catkin_ws/devel/lib/clfsm/clfsm`

providing as argument the directory

`$HOME/catkin_ws/devel/lib/`*`machineName`*`.machine`

If you place the option `-v` (for verbose) to `clfsm` before the arrangements of *llfsms*, you should see the execution of the states.

2.4 How to compile the machine `RosPingPong` with `bmake`

We can run *llfsms* outside the `catkin` environment and with `ROS`. We will use `bmake` and a `Makefile` to compile the machine `RosPingPong` before we execute it with `clfsm` (the `Makefile` for this is supplied with the download, just modify the early line that defines the variable `MACHINES` to the arrangement of *llfsms* you want to compile without the `.machine` extension).

Therefore, we need to install `bmake`. In Ubuntu, this is simply

```
sudo apt-get install bmake
```

We also need a directory for construction of libraries

```
mkdir -p /usr/local/lib
```

.

Download the `Makefile`. You must have created a `catkin` workspace for the `ROS` tutorials; suppose as per the tutorials is called `catkin_ws`. Typically,

```
cd $HOME/catkin_ws
ls
```

gives something like

```
build
devel
src
```

Place a directory `machines` in your `catkin_ws`.

```
mkdir -p machines
```

Thus, now

```
cd $HOME/catkin_ws
ls
```

gives something like

```
build
devel
machines
src
```

Place the `Makefile` inside the directory `machines` and also place `RosPingPong` there. It must be the case that

```
cd $HOME/catkin_ws
ls machines
```

gives

```
Makefile
RosPingPong.machine
```

Now, simply go inside the `machines` directory and compile them with the `Makefile` script using `bmake`.

```
cd $HOME/catkin_ws/machines
bmake
```

There may be some warnings, but an executable machine is produced [2]. In another terminal start `roscore`. Remember that the path to the executable of `clfsm` is

```
/catkin_ws/devel/lib/clfsm/clfsm
```

If you compiled with `bmake`, in the `machines` folder run the machine using `clfsm`.

```
/catkin_ws/devel/lib/clfsm/clfsm RosPingPong
```

Remember that the `-v` (verbose) option for `clfsm` shows the execution of the states.

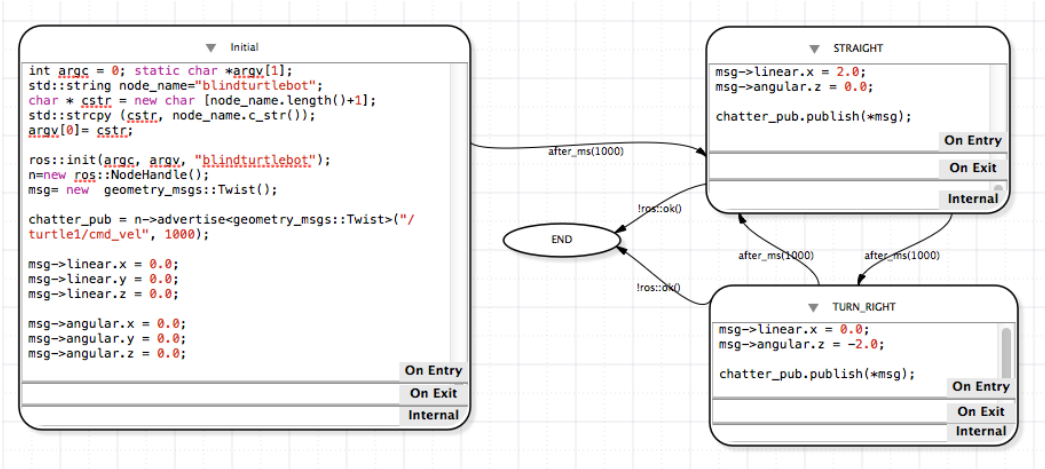

Figure 2: `RosBlindTurttleBot` is a simple *llfsm*. If you have completed the `ROS` tutorials and understand the semantics of *llfsms* you should also be able to describe what it does before you actually run it.

2.5 A machine that controls actuators: `RosBlindTurttleBot`

Now we demonstrate sending messages so that a robot does something; that is, a very simple machine that control the walking behavior 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 `RosBlindTurttleBot`. Figure 2 shows the schematics of it. The behavior is very simple, walk straight for a bit, then turn for a bit, and repeat these two sates.

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 `ROS` environment.

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

ros::init(argc, argv, "blindturtlebot");
n=new ros::NodeHandle();
msg= new geometry_msgs::Twist();

chatter_pub = n->advertise<geometry_msgs::Twist>("/turtle1/cmd_vel",
1000);

msg->linear.x = 0.0;
msg->linear.y = 0.0;
msg->linear.z = 0.0;

msg->angular.x = 0.0;
```

[2]For some `ROS` versions you may need to edit out linking against some libraries: `rosconsole_log4cxx`, `rosconsole_backend_interface` `boost_thread`

```
msg->angular.y = 0.0;
msg->angular.z = 0.0;
```

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.

```
ros::NodeHandle* n
ros::Publisher chatter_pub
geometry_msgs::Twist * msg
```

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

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "CLMacros.h"

#include <sstream>
```

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 message and advances one spin of ROS.

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

chatter_pub.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;

chatter_pub.publish(*msg);
```

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

You can compile and build the machine using the catkin package approach and the script machine_catkin_setup.sh (see Section 2.2).

Alternatively, you can use the bamke approach and make sure that the Makefile includes the name of the machine:

RosBlindTurttleBot

in the assignment to the variable MACHINES but without the extension .machine. Then simply compile it.

```
cd $HOME/catkin_ws/machines
bmake
```

There may be again some warnings, but an executable machine is produced. In another terminal start roscore. In a third terminal start ros::turtlesim.

```
rosrun turtlesim turtlesim_node
```

In the machines folder run the machine using clfsm.

```
 /catkin_ws/devel/lib/clfsm/clfsm RosBlindTurttleBot
```

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

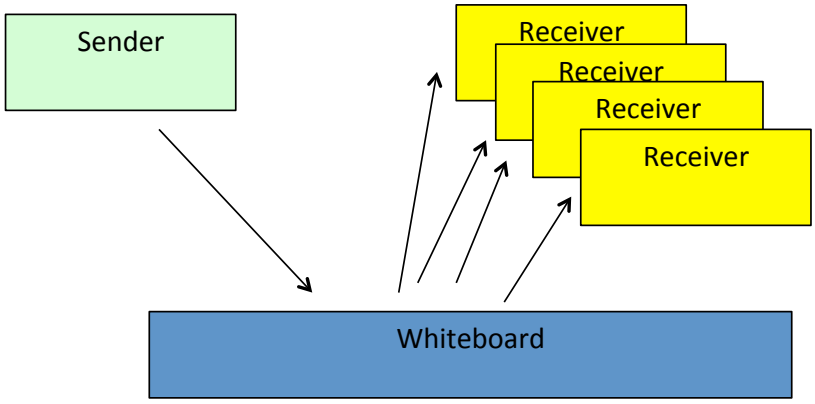

Figure 3: The role of some middleware (or whiteboard) is to simplify the APIs of communication between a sender of a message and the receivers.

2.6 A machine for reactive behavior: sensors and actuators

This is the third demonstration machine. It will require a bit more work, as with machines, we use an approach to messages that gives preference to the `get_Message` approach, rather than a publisher-subscriber approach. 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" by Vlad Estivill-Castro, Rene Hexel and Carl Lusty *International Conference on Simulation, Modelling, and Programming for Autonomous Robots (SIMPAR 2014)* Bergamo, Italy. October 20-23. In Brugali, D. et al. (Eds.): Lecture Notes in Artificial Intelligence LNAI 8810, pp. 182-194. Springer International Publishing Switzerland (2014). 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.

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, 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)**:** 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 *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 (`client.call` which is blocking) 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)

2.6.1 The sample `ROS`-service to publish the position of the turtle

Thus, this needs the position of the turtle in the middleware enabled by `ROS`. The program
`turtlesimlistener.cpp`
is such a `ROS` publisher and is distributed as the only source program of the package
`turtle_sensor_poster`.
Familiarity with the tutorials for `ROS:services` will facilitate understanding what this does. It publishes the position fo the turtle as messages basicly defined in
`turtle_sensor_poster/srv/TurtlePosition.srv`.
Download the package `turtle_sensor_poster.tar.gz` and place it in your `catkin_ws`. It should compile with the standard `catkin_make`.

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

This new machine (`RosWallTurttleBot`) appears in Figure 4. This machine requires the following includes.

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "turtle_sensor_poster/TurtlePosition.h"
#include "CLMacros.h"
#include <sstream>
```

And we make use of the following variables.

```
ros::NodeHandle* n
ros::ServiceClient client
ros::Publisher chatter_pub
geometry_msgs::Twist* msg
ros::NodeHandle* pos_n
long pos_x
long pos_y
turtle_sensor_poster::TurtlePosition srv
```

Most of the states and transitions are not as surprising given the previous machine. In fact, the Initial is almost the same and we have used the same string to name this `ROS` node (technically an

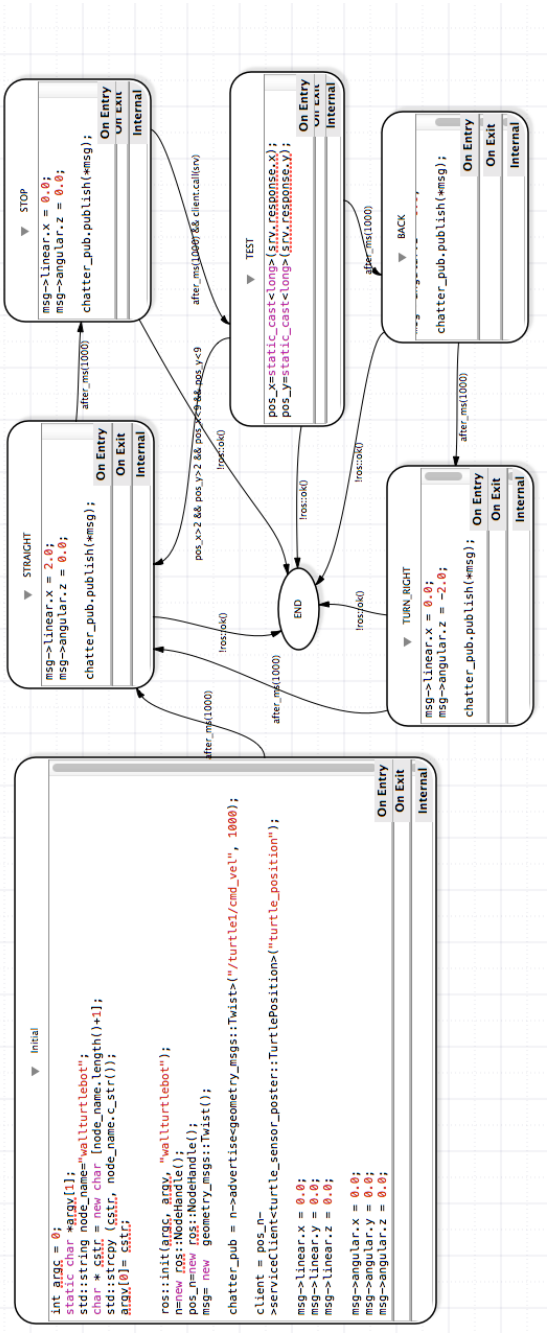

Figure 4: `RosWallTurttleBot` is a *llfsm* implementing a simple reactive behavior. You can see this behavior in action in the video youtu.be/4txscEXN8lQ.

error, but as long as you do not run both machines, there should be no problem). What you need to add is the initialisation of `pos_n` and the client[3].

```
pos_n=new ros::NodeHandle();
client = pos_n->serviceClient<turtle_sensor_poster::TurtlePosition>("turtle_position");
```

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

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

chatter_pub.publish(*msg);
```

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;

chatter_pub.publish(*msg);
```

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

```
pos_x=static_cast<long>(srv.response.x);
pos_y=static_cast<long>(srv.response.y);
```

We arrive to this state only after one second and a successful retrieval of the data from the publisher. That is, the transition between STOP and TEST is

```
after_ms(1000) && client.call(srv)
```

Read about `ROS::services` in the `ROS` tutorials if this is not clear, but a bit more discussion will follow when we present the service.

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

```
pos_x>2 && pos_y>2 && pos_x<9 && pos_y<9
```

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 a second, it performs the step-back (BACK) and turn (TURN_RIGHT)before going back to Straight.

2.6.3 Running the machine `RosWallTurttleBot`

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

[3]The diagram (Figure 4) is inconsistent, the placement of the service for the position is `turtle_sensor_poster`.

```
cd $HOME/catkin_ws/src
catkin_create_pkg RosWallTurttleBot std_msgs roscpp clfsm libclfsm
turtle_sensor_poster
```

This will create the necessary dependencies list in the file `package.xml` and in the `CMakeLists.txt`. Follow the same process as in Section 2.2; however, there is one more thing, we need to specify to find the includes for `turtle_sensor_poster`. Add

```
include_directories(${turtle_sensor_poster_INCLUDE_DIRS})
```

at the end of the other includes recommended by the script but before

```
include_directories(
${catkin_INCLUDE_DIRS}
)
```

Them, a `catkin_make` as usual should compile this machine. Now just do the placing of the result in `devel/lib/libRosWallTurttleBot.`*`some-extension`* to the target in `devel/RosWallTurttleBot.machine`.

Alternatively, run `bmake` on the `machines` directory.

```
cd $HOME/catkin_ws/machines
bmake
```

Now, open four terminals. In one, run `roscore`, the communication middleware. In another, run the turtle simulator.

```
rosrun turtlesim turtlesim_node
```

On a third one, we run the service.

```
cd $HOME/catkin_ws
./devel/lib/turtle_sensor_poster/turtlesimlistener
```

Finally, the machine is executed in the fourth terminal.

```
cd $HOME/catkin_ws/machines
/catkin_ws/devel/lib/clfsm/clfsmRosWallTurttleBot
```

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

2.6.4 A machine to suspend and re-start `RosWallTurttleBot`

Several *llfsms* can be executed concurrently in `clfsm`. Also, they can be suspended, restarted and resumed. One example is `TurtleSuspendResume`. The diagram of the machine appears in Fig. 5. We emphasize that this machine makes use in its includes of

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

This is important, observe the transitions like

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

and

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

. Also, we see the calls in the state Initial to suspend the earlier machine

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

While in the state RESTART

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

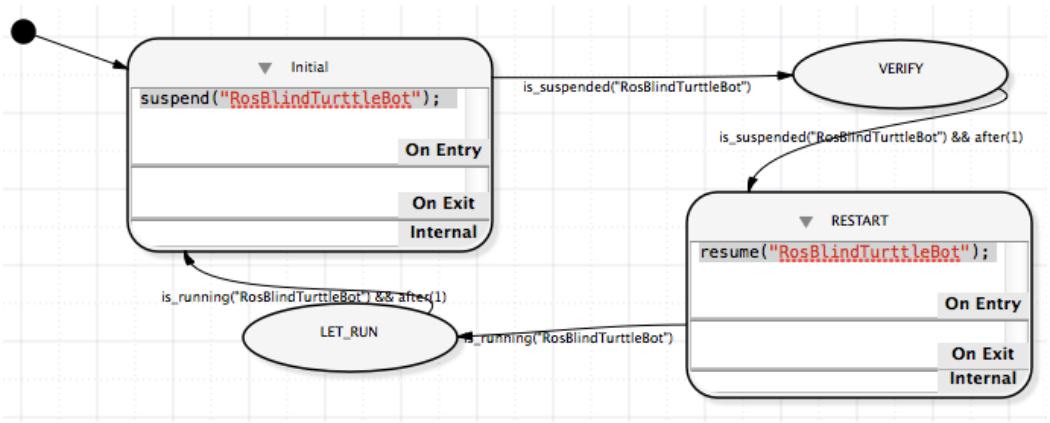

Figure 5: `TurtleSuspendResume` is a *llfsm* that suspends and starts `RosWallTurttleBot`.

enables the other machine to continue. There are some important details about the scheduling `clfsm` does 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 we are coming from another state, to evaluate all transitions out in sequence and if one fires, the **OnExit** run 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* it 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.