Getting to Use MiPal's MiEditLLFSM

Vladimir Estivill-Castro
Machine Intelligence and Pattern Analysis Laboratory
Griffith University, Institute for Integrated and Intelligent Systems

September 18, 2021

Abstract

This document provides an introduction to the editor MiEditLLFSM. This is an editor of logic-labeled finite state machines. These machines can be compiled and executed with the MiPal tool clfsm. Logic-labeled finite-state machines are models of computation derived from state charts. The computation progresses from state to state when transitions fire. Transitions are logic expressions, while states have executable code in three sections **OnEntry**, **OnExit** and **Internal**.

Contents

1 Introduction

MiEditLLFSM was developed in `java` using `NetBeans IDE 7.4`. It should execute with `java 1,6` or higher. This document provides an introduction to the editor MiEditLLFSM. This is an editor of of logic-labeled finite state machines (LLFSMs). These machines can be compiled and executed with the MiPal tool clfsm. Logic-labeled finite-state machines are models of computation derived from state charts. The computation progresses from state to state when transitions fire. Transitions are logic expressions, while states have executable code in three sections **OnEntry**, **OnExit** and **Internal**. FSMs are a very useful instrument to describe the behavior of an automata. They can also be illustrated as state-diagrams. Here, we describe the approach of the MiPal team for FSMs.

The FSMs of the MiPal team closely follow UML 2 and Harel's state-charts while taking into consideration many widely used approaches. Historically, the almost de-facto standard for FSMs is derived from the *STATEMATE* model and tool [**?**, **?**] but there have been many alternative proposals [**?**]. There are several commercial products including *QP*™[**?**], *BotStudio* [**?**] *StateWORKS* [**?**] and *MathWorks*® *StateFlow*. The robotics standard `ROS` has a tool named `smach` (see: `http://wiki.ros.org/smach`). The UML form of FSMs derives from OMT [**?**, Chapter 5], and the MDD initiatives of Executable UML [**?**].

MiPal state machines are also in close proximity with the mathematical model of behavior, the so called finite-state automata [**?**] that produce output, also known as transducers. That is, our FSMs consist of a set S of **states**, and a transition function $T : S \times E \to S$. There is a distinguished state $s_0 \in S$, named the initial state. Typically, the set E is a set of events, or a set of input symbols, but in the case of MiEditLLFSM this is a set of **Boolean expressions**[1]

MiPal's FSMs will be of a synchronous type. The set E are expressions. Moreover, T is usually a *partial function*, that is, there are pairs (s_i, e_t) for which T is not defined; so, T is usually called the transition table. The standard general description of the semantics for $T(s_i, e_t) = s_j$ is that when the machine is in state $s_i \in S$ and if the expression e_t evaluates to true, the machine will move to the state s_j. However, this

[1] In fact, it would be desirable that a transition be labeled by any call (local,remote) to an agent/code that returns a Boolean value and that LLFSMs be generic with respect to the language that once evaluated determines a Boolean value for the expression labeling the transition. For example, in many case studies MiPal has used Defeasible Logic (DPL) [**?**], this is common sense logic. This is an aspect that characterizes MiPal's FSMs; as we mentioned earlier. This is very useful for declarative aspects, for example, we have used models of logic to express what is the off-side rule in soccer [**?**], when it is dangerous for a senior lady to face a stranger [**?**], and to describe hands of poker [**?**]. That is, logic becomes a very useful tool for communicating what criteria classify a soccer player's position as valid, or a set of 5 cards as a full-house.

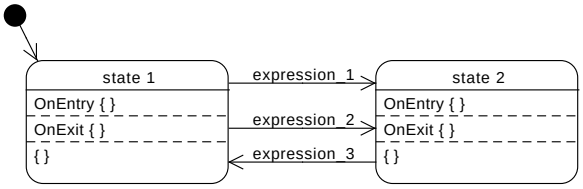

Figure 1: Basic elements of the notation.

requires that, (for any $t \neq s$), if $T(s_i, e_t)$ and $T(s_i, e_s)$ are defined and $T(s_i, e_t) \neq T(s_i, e_s)$, then e_t and e_s never be true simultaneously (unless one is modeling completely non-deterministic behavior).

We simplify the burden for the behavior designer by making the projection of T on each state a sequence instead. That is, $T(s_i, e_t) = s_j$ will cause a transition to state s_j if e_t evaluates to true and no previous expression e_s evaluates to true ($\forall s < t$ in the sequence $T(s_i, \cdot)$). Note that while this is simply syntactic sugar, it does make the task of the behavior designer a lot simpler. Thus, in MiEditLLFSM, each states has associated with it a sequence of pairs (e_s, s_i) denoting the transitions out of the state. The state s_i is the target state when e_s evaluates to `true`.

The enforcement that all transitions out of a state be in sequence avoids some problems that emerge with other languages. In UML 2, and other FSMs languages that enable guard conditions, a need appears to recommend best practices [**?**], where the exclusive disjunction of all guard conditions out of a state shall always be true. Also, in UML 2 and *STATEMATE*, two transitions from a single state that evaluate to true represents a conflict and an invalid configuration.This is not a concern in MiPal's modeling language. Expressions out of a state form a list (in Fig. **??** we have indicated this by a sequence numeral) and thus, the second expressions can be seen as the conjunction with the negation of the first. *MathWorks*®, *StateFlow* with *SymLink* concurs with MiPal's approach and specifies a sequential evaluation of only one event at a time and a mechanism to specify priorities in transitions but its larger set of primitives and its semantics requires complex translations for performing model-checking [**?**].

In a FSM, each state models a period in time where an action[2] takes place. However, there are three sections where actions are grouped. An **OnEntry** section is executed upon arrival to the state, while actions in the **OnExit** section are executed as the machine departs that state. Thus, the actions in these two sections are executed once and only once (and of course upon arrival and upon departure, respectively, in the corresponding state). The third section is a section for internal actions[3] that are executed only if none of the transitions fires. When the internal actions are completed, execution returns to evaluate the sequence of expressions that label transitions out of the state and the cycle is repeated. We refer to one pass over the cycle as a *ringlet*.

2 The `Hello World` LLFSMs

We now construct the very first LLFSM and run it. Simply start the MiEditLLFSM; you can double click or in a terminal window with `java` installed and configured type the following command.

```
java -jar MiEditLLFSM
```

You should get a screen that looks as in Fig. **??**. We will now use the editor to create a new machine. In the menu bar File chose the New option and use the file chooser to navigate to a directory where you will create a LLFSM. Each machine is physically stored in a directory with extension `.machine`. In fact, a LLFSMs for MiEditLLFSM and for CLFSM are directories with extension `.machine`.

So, when you are satisfied with the location of the new LLFSM hit the Open button and you will be queried for the name of the new machine. Type `HelloWorld` and hit the OK button. Many buttons and panels of MiEditLLFSM will become active as a machine with one state has been created. The state is named

[2] We make no distinction between actions and activities, and more on this will be discussed later.
[3] In UML known as the `do` section.

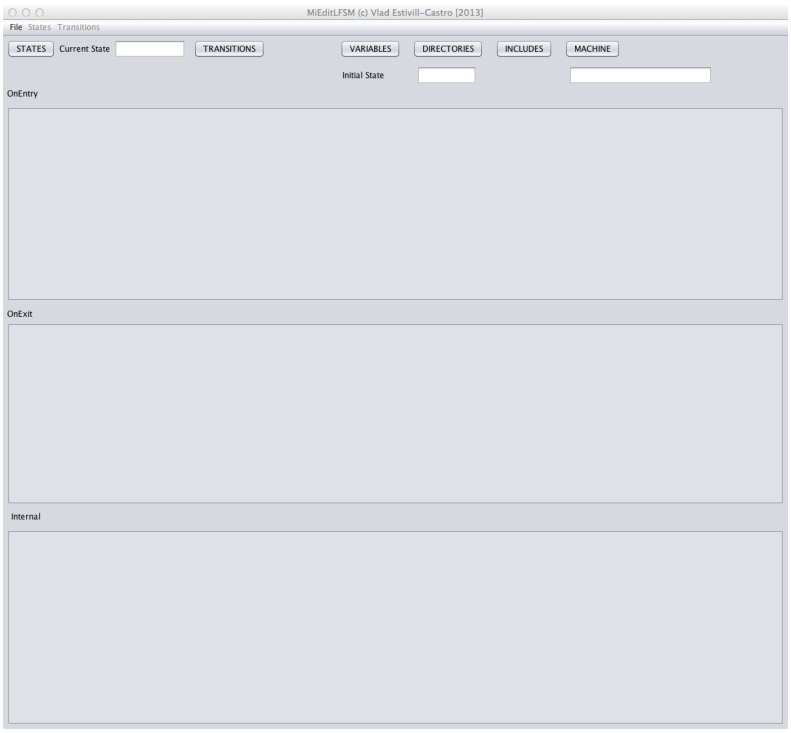

Figure 2: The initial and main screen of MiEditLLFSM.

INITIAL, and it is also the current state. The INITIAL state is where execution starts. In MiEditLLFSM you can edit only one machine at a time and only one state of that machine. That is called the current state.

The name of the machine you are currently editing should appear on the right top corner of the main screen of MiEditLLFSM and a little to the left of it, you will also see the name of the initial state.

2.1 Entering code

Click the first large section labeled **OnEntry**. You are free to enter any text in any formatting that you wish here. LLFSMs are meant to be as language agnostic as possible. Enter the following code.

```
fprintf(stderr,"STATE: %s\n",state_name());
```

Now, go to the States menu and chose the Save option. In MiEditLLFSM changes to a LLFSM are not saved unless you explicitly use the Save option in the States menu. After saving, you should also go to the File menu and chose the Exit option. This terminates MiEditLLFSM.

Note: the File menu, Save is completely equivalent to the. using Save in the States menu.

2.2 Opening an existing machine

Open MiEditLLFSM and in the File menu select the option Open. Navigate until you can select again the `HelloWorld` machine. Once selected, hit Open and the previous disabled buttons will become active on MiEditLLFSM. Also, the **OnEntry** section should have the text of your previous section. In the **OnExit** section add the following code.

```
fprintf(stdout,"Hello\n");
```

Do not forget to go to the State menu and to use the menu item Save to save this change.

3 A first execution, and include files

You could potentially think you can execute this simple machine that you created. We assume you have installed CLFSM. Lets assume that `$CLFSM_HOME` is the path to the CLFSM executable and that `$CLFSM_INCLUDE` is the path to the CLFSM includes. You can test `$CLFSM_HOME` by typing on your shell prompt the following.

```
$CLFSM_HOME/clfsm
```

The CLFSM runs without any output. To test `$CLFSM_INCLUDE`, type

```
ls $CLFSM_INCLUDE
```

You should see the file `CLMachine.h` among a list of files that start with `CLAction.h`.

Thus running the `HelloWorld` machine just need the command

```
$CLFSM_HOME/clfsm -I$CLFSM_INCLUDE HelloWorld
```

This will attempt to compile the LLFSM `HelloWorld`. You should get an error of the form

```
error:  use of undeclared identifier 'stderr'
```

This is because MiEditLLFSM is agnostic to any language. We need to place proper include files. To place include files for the entire machine, click on the `INCLUDES` button. A new free text editing box appears labeled `Machine INCLUDE file`. Type

```
#include <stdio.h>
```

Make sure that you do not leave any blank-space characters between the sharp sign # and the start f a line. Hit the *SAVE* button to close the free text input and then try to run th machine again. This time the error would be of the form

```
error:  use of undeclared identifier 'state_name'
```

There is two solutions to this. One is to remove the code from the **OnEntry** section. If you do this, your machine shall run; however, we recommend you leave something to print on the **OnEntry** as the **OnExit** will not be executed since there are no transitions that fire.

The second solution is to actually include the file named `CLMacros.h` among the includes of the machine. This is because `state_name()` is part of the CLFSM infrastructure. If you tried the first solution and your machine ran, and then you edit it to carry out the second solution, nothing may happened. This is because CLFSM does not recompile. You need to remove the corresponding directory. On Mac-OS this usually requires you to do the following

```
rm -fr HelloWorld.machine/Darwin-x86_64
```

Implement the second solution by adding the line

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

using the Includes button and the free text box for `Machine INCLUDE file` (again, we emphasize that you should not leave any blank-space characters between the sharp sign # and the start of a line). Now, when you run the machine you should get in the output the name of the current state.

3.1 Adding another state

You can add a new state by going to the *State* menu and selecting the *Add* menu item. A box will query you for the name of the new state, Type `SECOND`. This will become the current state in MiEditLLFSM. On the **OnEntry** section type again

```
fprintf(stderr,"STATE: %s\n",state_name());
```

while in the **OnExit** section type

```
fprintf(stdout,"World\n");
```

3.2 Transitions

Now we will add two transitions. First, from the INITIAL state to the SECOND state. Go to the Transitions menu and select the menu item Add. Since the current state is SECOND, let the target state be the default, namely the INITIAL state. Click the OK button in the "Selecting a destination for the transition" and you will be asked to enter a Boolean expression. leave the default expression (the value `true`).

Now we add a transition from INITIAL to SECOND. For this, we make the current editing state the initial state by clicking on the STATES button on the top right corner of the GUI. A box allows you to select a state, so chose INITIAL. You will see that the **OnExit** has changed to printing `Hello` to the `stdout`. Now, again, in the Transitions menu select the Add option. Change the default expression to

```
after(1)
```

This is a CLFSM macro that waits for a second. That is, it evaluates to true once a second has passed. Save the state and execute this machine. You should see the output of each of the states after one second.

3.3 Editing a transition

Lets change the delay from one second to two seconds. For this, click the top button called TRANSITIONS. A list of all transitions out of the current state will appear. In fact is a list of pairs (target state, expression). This is because the expression could potentially be duplicated with a different target state; and also, there could be two transitions to the same target state but with different expression. At the moment there is only one (target state, expression) pair, so simply click the OK and a window that allows you to edit the corresponding expression will appear. Edit the expression to `after(2)` and click OK. A warning will appear reminding you that, for this change to have effect, the current state must be saved. Save the state as before and run the machine. Convince yourself you can edit transitions in both directions and to different delay values.

4 Variables

Because our models consist of arrangements of FSMs, variables used in each of the sections above are of 3 types. The first type, **local variables**, are exclusive to one and only one FSM (that is their scope is only the states of one FSM). **Internal variables** are shared by all the FSMs in the arrangement (that is, their scope is all the states of the FSMs in the arrangement). Finally, **external variables** are variables whose scope goes even beyond the arrangement of the FSMs and in embedded system are variables that are set by external sensors or are set to activate effectors and actuators. The environment that holds the variables is named the *whiteboard* [?], but it also correspond to the software architecture pattern of a repository [?].

5 Concurrency model

By design, in one ringlet execution there is only one **read** operation by which a local copy of external and internal variables in the scope of the current FSMs is made before the execution of any section or the evaluation of any expression labeling any transition. That is, all execution in a ringlet is in the same context that is not modified by any other concurrent FSM or any external event (a new sensor reading, for example). If no transition fires and the internal actions complete, when a new ringlet commences, a new read of the external scope will take place. All writes of external or internal variables by a FSM take place immediately in the shared context. Our choice to place only one **read** instance of the variables per ringlet may seem to contradict the *STATEMATE* "execution time" requirement that suggest changes in any point in time should be reflected in the next. However, this creates serious problems in robotics applications where there is an *open environment* [?], and languages like RFSM also take an approach to evaluate the set of transitions out of a state in the same context.

6 Arrangements of machines

The arrangement of FSMs is executed by a round-robin switch from one ringlet of one FSMs to the next one in the arrangement. Thus, the arrangement of FSMs is a single sequential execution, executed by one thread that interprets the semantics described above. It is possible also to indicate a relative frequency for each FSMs enabling different rates of progress which are implemented by each FSMs having a certain number of ringlets performed before passing the execution token to the next FSM in the arrangement. Note that this style of execution is very much in line with the time-triggered architecture [?] (as opposed, as we mentioned earlier, to an event-driven architecture).

The use of a single-tread execution for the several FSMs in the arrangement, **as opposed to the parallelization** by a semantics that just specifies concurrency (that is arbitrary rate of progress for each, as each is executed within an independent thread) brings several advantages. It has been argued that from the design point of view, open concurrency (where the management of switches between threads is left to the system) represents an unnecessary cognitive load in the model designer [?] as it opens all sorts of needs for communication, synchronization and consideration of communication delays. There is added complexity in ensuring properties like fairness, management of critical sections, no deadlock, and extermination of starvation. It is also the case that the execution (that is implementation) is usually less efficient as the concurrency control mechanisms consume CPU cycles and may need to manage context switches and communication primitives with native support from the operating system or the hardware. Perhaps more important is the actual formal verification that the models are correct. Model-checking of models that enable concurrent threads must consider a universe of all possible states of the system and such universe is the Cartesian product of all possible states of each thread. This combinatorial explosion significantly complicates the formal verification of such system. For robotic systems and embedded systems where there may be several timing requirements, sequential execution has been proposed as superior to the multiplication of threads [?].

By using sequential scheduling we maintain concurrency, and the models produced with the logic-labeled FSMs can be verified using public domain model-checking technology (`NuSMV`) within a matter of seconds [?, ?], while for the same case studies, but using Behavior Trees [?] – which have explicit notation for spawning parallel threads – require several days of CPU to verify equivalent properties [?].

It is important to note that the approach presented here is not a departure from the event model of traditional FSMs. In fact, the ability of our FSMs to use statements in a decidable common-sense logic allows for a more complex event definition with clear value and temporal semantics [?]. Sensors that trigger events in an embedded system simply swap a state variable. Such variable is an external status variable, of which a snapshot is taken at the time the token arrives to the machine that evaluates a condition (transition) to the state of the sensor reading.

7 An example with Webots

In this section we will use our editor to create a behavior in a small differential robot named ePuck in the simulator Webots. We will also use the MiPal object-oriented whiteboard and the bridge between the MiPal object-oriented whiteboard and Webots.

We will start by creating a very simple one-state machine to check that all the infrastructure you need is in place. Look at MiPal *Getting Started* document to set up the MiPal object-oriented whiteboard. Follow the standard installation document of Webots as well.

Start MiEditLLFSM and create a new machine. Lets called `MoveForward`. Recall you do this by selecting the menu item File and choosing the New option. Then, you use the file chooser to navigate to a directory where you will create a LLFSM. Select the directory and a dialog will appear where you enter `MoveForward`.

After the machine has been created, many more buttons become enabled. Chose the VARIABLES button and a dialog with 3 columns shall appear (the aim is to get it to look like Figure ??). In the first column type `int` (as we will be declaring a C++ integer variable. In the second column type `robotID` and as a third column you can enter any string as a comment. Type `the robot ID for Webots`. Unfortunately the dialog requires you to chose another cell before you close this dialog for the changes to be reflected. So, chose another cell and then click on the CLICK IN ANOTHER CELL AFTER EDITING AND THEN SAVE HERE.

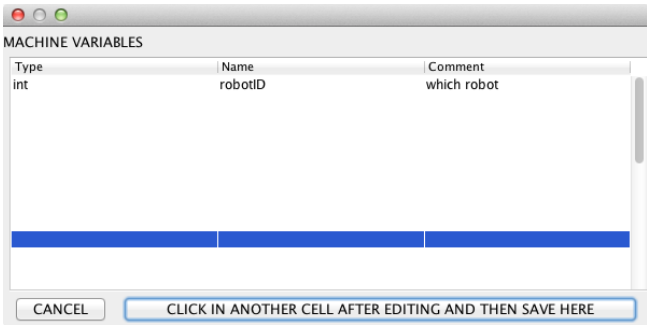

Figure 3: Window to declare variables at the scope of the machine.

This has created an external variable with the scope of all states of the machine. Now, click the Includes. In the dialog type the following text.

```
#include <stdio.h>
#include "CLMacros.h"
#include "typeClassDefs/WEBOTS_NXT_bridge.h"
#include "CLWhiteboard.h"
using namespace guWhiteboard;
#define DEBUG
```

Because we will use `fprintf()` we include

```
#include <stdio.h>
```

.

Because we will use `state_name()` we include

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

Because we will use MiPal's `Webots` bridge we include

```
#include "typeClassDefs/WEBOTS_NXT_bridge.h"
```

Because we will use the handlers of MiPal's object oriented whiteboard, we include

```
#include "CLWhiteboard.h"
```

Because we can refer more easily to names from the bridge in the namespace of the MiPal whiteboard we include

```
using namespace guWhiteboard;
```

Finally, since we may want to turn of debugging output we can use the preprocessor directive

```
#define DEBUG
```

Now we are in a position to enter the code of the **OnEntry** section for the initial state. Type

```
#ifdef DEBUG
fprintf(stderr,"STATE: %s\n",state_name());
#endif

robotID=0;

// START the motors
WEBOTS_NXT_bridge thetMotorCommand(robotID,MOVE_MOTORS, 50, 50,false);
a_Command_Handler.set(thetMotorCommand);
```

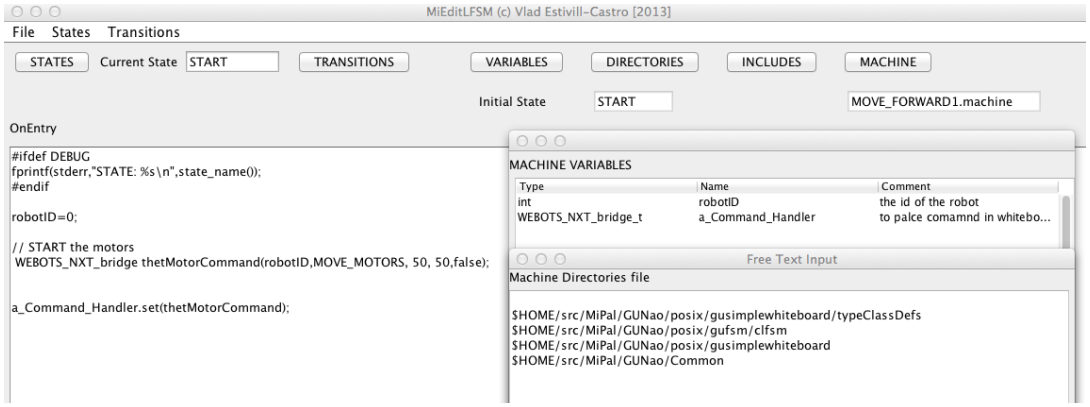

Figure 4: The one state in MiEditLLFSM that sets the e-Puck going.

Do not forget to go to the menu item States and Save this state.

This code of the **OnEntry** section will print the name of the state if the debug flag is enabled. Initialize the external variable `robotID` and use the bridge to start the motors at speed. Before you continue, rename the initial state to `START`. You do this by editing the text field besides the `CurrentState` label.

You can not edit the name of the initial state in the text field for the initial state. You must make it the current state first.

We are almost ready to run this machine. Go back to the VARIABLES and add the variable

```
WEBOTS_NXT_bridge_t a_Command_Handler
```

This variable is used to place the motor command using the MiPal Object-Oriented whiteboard.

The compiler inside `clfsm` needs some paths to find the corresponding code for the MiPal whiteboard as well as for the bridge. Because we are using the MiPal whiteboard and also the current MiPal `WEBOTS_NXT_bridge`, you need to install this components. You may not need to install all the tools described in the MiPal *Getting Started* document. However, you certainly need to set up the environment variable `GUNAO_DIR`. Click on the DIRECTORIES and enter the following paths.

```
$GUNAO_DIR/posix/gusimplewhiteboard/typeClassDefs
$GUNAO_DIR/posix/gufsm/clfsm
$GUNAO_DIR/posix/gusimplewhiteboard
$GUNAO_DIR/Common
```

Then click the SAVE button.

Now, you can go to the directory and run this machine.

```
$CLFSM_HOME/clfsm -I$CLFSM_INCLUDE MOVE_FORWARD1.machine
```

You should see the machine being compiled, executed and then print

```
STATE: START
```

How to check that it actually starts an e-puck inside `Webots` running? You need to compile the bridge and place it as a controller in a `Webots` world. See the document MiPal `Webots` *Modules* document. Typically, you revert the world in `Webots`, that load the bridge as a controller and starts the simulation. Then, with the simulation running you also execute

```
$CLFSM_HOME/clfsm -I$CLFSM_INCLUDE MOVE_FORWARD1.machine
```

The e-Puck will run away. Stop th simulation since the one-state machine will not stop this e-Puck as it has exited. Fig. **??** displays the one state machine that has this simple behavior.

7.1 Machine that follows a line

We now demonstrate a few more features of CLFSM, the MiPal whiteboard and the bridge by showing an example of a LLFSM where the robot follows a line. It is also an example of feedback loop control. If the robot gets an image that indicates the line is to the right of its trajectory, it will accelerate the motor in the right wheel and slow down the motor in the left wheel proportional to the absolute value of the difference between the color-pixels median and the straight direction of movement.

So, we need to create a new machine. Lets call it EpuckFollowsLine. In the initial state, the machine will do some initialization. So, place the following code in the **OnEntry** section of the initial state[4].

```
#ifdef DEBUG
std::string stateName("STATE: "); stateName+=state_name(); print_ptr(stateName);
#endif
robotID=0;
speedToUse=200;
leftSpeed=0; rightSpeed=0;
cameraWidth=0;
delta=0; maxSpeed=0.0;
//Follow magenta
theChannel=GREEN_CHANNEL;
//Follow blue
theChannel=RED_CHANNEL;
//Follow yellow
theChannel=BLUE_CHANNEL;
WEBOTS_NXT_bridge a_Command(robotID,CAMERA,theChannel,1);
a_Command_Handler.set(a_Command);
```

This code needs some variables to be declared. So click on the VARIABLES and introduce the following declarations.

```
Print_t                print_ptr          To display in whiteboard
int                    robotID            The ePuck id
WEBOTS_NXT_bridge_t    a_Command_Handler  to place Webots-bridge commands in the whiteboard
int                    speedToUse         the usual speed
int                    rightSpeed         right wheel speed
int                    leftSpeed          left wheel speed
int                    cameraWidth        the ePuck camera width
CAMERA_E_PUCK_CHANNELS theChannel         to select the color
float                  maxSpeed           ePuck maximum speed
int                    delta              difference between input and output signal
```

So, with the variable declarations we can explain the earlier code for the **OnEntry** section of the initial state. The section around the `#ifdef DEBUG` will display the name of the state being executed as it runs the **OnEntry** section in the whiteboard. You can monitor this by running MiPal's whiteboard monitor (the option `-v` o CLFSM achieves something similar, showing the state names as they re executed, more on this later).

The variable `robotID` determines which robot the messages are for in Webots by the bridge. The `speedToUse` is the recommended speed in an example for this in a tutorial by Webots. The `leftSpeed` and `rightSpeed` would correspond to the speeds of each of the wheels. The `cameraWidth` of the ePuck can be obtained from the bridge, and we will need it to determine the moves.

The variable `delta` denotes the observer discrepancy that will provide the magnitude of the error to the feedback-loop control. The variable `maxSpeed`

The ePuck and the bridge allow to have a 3 colors (channel) in the camera. The variable `theChannel` will indicate which to use. In this case, the last value assigned to it is the constant `BLUE_CHANNEL`, which makes yellow pixels dark and everything else white. The code

[4] The current `WEBOTS_NXT_bridge` interface does not match the new MiPal whiteboard model, where there is no queuing of messages of the same type posted immediately after each other, so figures that show we stop the motors as well in this state are not correct anymore.

```
WEBOTS_NXT_bridge a_Command(robotID,CAMERA,theChannel,1);
a_Command_Handler.set(a_Command);
```

builds an message of the class `WEBOTS_NXT_bridge`. This message sets the camera of the given robotID to use the color channel in `theChannel`.

The last bit of code builds a `WEBOTS_NXT_bridge` message as a instruction (or command) to the ePuck and the command is `MOVE_MOTORS`. because the left speed is set to zero and so is the right speed this stops the motors. The instruction `a_Command_Handler.set(a_Command);` is what actually places the message in the whiteboard. The last parameter of the constructor in `WEBOTS_NXT_bridge` messages usually indicates whether the message is the e-Puck to the outside (in this case set to `true`) or from the outside to the ePuck (then, set to `false`. Thus an instruction to move to motors is from the outside to the ePuck an the last parameter is `false`. If it is a command about the values of the camera will be read (about a sensor), then is set to `true`.

Because MiEditLLFSM/ does not have syntax-checking, it is a good idea to not to write to many states until you are sure you have typed the variables correctly. But before you compile even a machine with only one state, recall you need to set up the includes. Click on the button INCLUDES and type the following.

```
#include "CLMacros.h"
#include "typeClassDefs/WEBOTS_NXT_bridge.h"
#include "CLWhiteboard.h"
using namespace std;
using namespace guWhiteboard;
#define DEBUG
```

The `CLMacros` enable functions like `state_name()`, while `#include "typeClassDefs/WEBOTS_NXT_bridge.h"` enables MiPal's `Webots`-bridge. You need `#include "CLWhiteboard.h"` to use the MiPal's whiteboard. The other are namespace declarations that save typing `string` instead of `std::string`. The `DEBUG` flag enables the code that is between the `#ifdef ... #endif`.

And also you need the path directories used in compilation. So, click on the button DIRECTORIES and input the following.

```
$GUNAO_DIR/posix/gusimplewhiteboard/typeClassDefs
$GUNAO_DIR/posix/gufsm/clfsm
$GUNAO_DIR/posix/gusimplewhiteboard
$GUNAO_DIR/Common
```

Make sure you do not leave any blank-space characters before the actual path and the start of the line. Moreover, please note that these paths are dependent on installing the MiPal whiteboard and clfsm as per the MiPal *GettingStarted.pdf* document.

To check that all you have built works so far execute the machine with the following command.

```
$CLFSM_HOME/clfsm -v -I$CLFSM_INCLUDE EpuckFollowsLine.machine
```

If you have no errors, compilation will complete and the machine will run. Do not forget the `-v` option and then, the output will be something like

```
m 0 s 0 - EpuckFollowsLine.machine - INITIAL
```

The program clfsm would then terminate. If you are running the `guWhiteboardMonitor` you should see output similar to this.

```
Type:  Print             Value:  STATE: INITIAL
Type:  PlayerNumber      Value:  2
Type:  WEBOTS_NXT_bridge Value:  0,CAMERA,0,1,
```

Make sure you understand how these 3 messages ended up in the whiteboard.

Now, rename the initial state to `TURN_CAMERA_ON`. Do you see when we instruct the ePuck to start reporting from the camera?

After that, we going to add another state. So, in the menu for State chose the option Add and create a new state named TURN_ON_ENCODERS. Although the CLFSM -v option is very convenient, paste the following code in the **OnEntry** section of this new state.

```
#ifdef DEBUG
std::string stateName("STATE: "); stateName+=state_name(); print_ptr(stateName);
#endif
```

Also, we put something useful.

```
WEBOTS_NXT_bridge commandLeft(robotID,ROTATION_ENCODER,LEFT_MOTOR_DIFFERENTIAL,1);
WEBOTS_NXT_bridge commandRight(robotID,ROTATION_ENCODER,RIGHT_MOTOR_DIFFERENTIAL,1);
a_Command_Handler.set(commandLeft);
a_Command_Handler.set(commandRight);
```

You can interpret this two messages on the bridge as turning on the reporting on the encoders for each of the wheels. If you now run the machine and the guWhiteboardMonitor as well as the Webots simulation, the whiteboard will be regularly populated with reports on the values of the wheels spinning (or constant values if the wheels are actually not moving). However, to actually test we arrive to this state we need a transition from TURN_CAMERA_ON to TURN_ON_ENCODERS. To do this, you have to make TURN_CAMERA_ON the current state after you **saved** the code you just put into TURN_ON_ENCODERS.

Once TURN_CAMERA_ON is the current state select the menu item Transitions and the option Add. The default expression for a transition is the trivial Boolean expression true, change this to after_ms(20), this is sufficient for messages in the MiPal whiteboard to not get overwritten by another message of the same type posted too fast immediately afterwards. Try running the machine again. Remember that you should remove the subdirectory where files are compiled if you require re-compilation. In Mac-OS it the subdirectory staring with Darwin.

You should now see that the machine goes trough 2 states before exiting, but if you are running the Webots simulator, many messages will be posted and continue to be posted even the machine has exited? Why? Well the bridge was left in a state of reporting the encoders and while the simulation is running this will happen. The encoders must be on to learn the maximum speed, because in the message of the encoder values the maximum speed is also provided.

7.1.1 Obtaining the maximum speed and the camera width

We will now add one more state. Lets call it GET_MAX_SPEED_AND_CAMERA_WIDTH. In the **OnEntry** section of this state we enter the following code.

```
#ifdef DEBUG
std::string stateName("STATE: "); stateName+=state_name(); print_ptr(stateName);
#endif
WEBOTS_NXT_encoders_t encoder_data_ptr;
maxSpeed=M_PI * (encoder_data_ptr.get()).maxSpeed();
#ifdef DEBUG
fprintf(stderr,"maxSpeed Read %f\n",maxSpeed);
#endif
```

In the **OnExit** section we will turn off the encoders as they re not needed any more.

```
 WEBOTS_NXT_bridge commandLeft(robotID,ROTATION_ENCODER,LEFT_MOTOR_DIFFERENTIAL,0);
WEBOTS_NXT_bridge commandRight(robotID,ROTATION_ENCODER,RIGHT_MOTOR_DIFFERENTIAL,0);
a_Command_Handler.set(commandLeft);
a_Command_Handler.set(commandRight); WEBOTS_NXT_camera_t camera_data_ptr; //the Width
```

We need a transition to reach this state. So go back to TURN_ON_ENCODERS and make a transition to GET_MAX_SPEED_AND_CAMERA_WIDTH. Instead of just leaving the default true change it to after_ms(30). This 20ms delay ensures the signal about the encoders is processed and forwarded in the whiteboard.

Figure 5: The transitions and states so far.

7.1.2 Obtaining the difference of desired system-state and current system-state and issuing a proportional correction

Now we create the finite-machine state where we read the image and calculate the error. That is, we calculate the displacement away from the desired line. So, create a new state called FEEDBACK_CONTROL. Here, we will collect the center of mass of the pixels of the color we are following, and then in the variable `delta` we obtain the signed magnitude of error from such center being at the center of the camera. Then, we set speeds for the motors accordingly. The code of the **OnEntry** section in this new state is as follows:

```
#ifdef DEBUG
std::string stateName("STATE: "); stateName+=state_name(); print_ptr(stateName);
#endif
WEBOTS_NXT_camera_t camera_data_ptr;
// the WIDTH is a property of the camera across all channels
// WEBOTS_NXT_camera theActualCameraObject = camera_data_ptr.get();
cameraWidth = (camera_data_ptr.get()).width() ;
// second parameter of a Camera Channel is the value of the middle point
// delta is the error to the desired state, as a feedback loop control model
delta = ((camera_data_ptr.get() ).get_channel(theChannel)).secondParameter() -cameraWidth/2;
// set the speeds
leftSpeed= speedToUse -4*abs(delta)+4*delta;
rightSpeed=speedToUse -4*abs(delta)-4*delta;
```

Again, we need a transition to arrive to this state. Go back to the earlier state GET_MAX_SPEED_AND_CAMERA_WIDTH. Add a transition also with expression `after_ms(30)`. We now have a sequence of transition as per Figure **??**.

7.1.3 Sending the motors their new speed

Now we actually add the loop of the famous *Feedback loop control*. We add a new state SET_MOTORS_SPEED. So, add this state as usual, and place the following code in the **OnEntry** section.

```
#ifdef DEBUG
std::string stateName("STATE: "); stateName+=state_name(); print_ptr(stateName);
#endif
WEBOTS_NXT_bridge
thetMotorCommand(robotID,MOVE_MOTORS, leftSpeed/maxSpeed, rightSpeed/maxSpeed,false);
//post the speed
a_Command_Handler.set(thetMotorCommand);
```

We add two transition. Form this state to the earlier state FEEDBACK_CONTROL we put `after_ms(32)`. This is because the Webots controller (the MiPal `guwebotsinterfacemodule`) has been set to a cycle of 32ms. The transition back is now just the expression `after_ms(10)`. Figure **??** shows this finite state machine and the Webots simulator. The simulator display the ePuck robot executing this program,a nd thus follows a line of color. To run this machine and have this effect on the robot, you need to be running the MiPal `webotsbridge` as a controller for that Webots world. You need to read the MiPal "WebotsGettingStarted.pdf" document [5]. The
tt WEBOTS_NXT_bridge class of the MiPal *whiteboard*is the one used here in the code to create objects of

[5]Under the MiPal infrastructure this usually consists of placing yourself in the directory of the `guwebotsinterfacemodule`. That is `$MIPAL_HOME/src/MiPal/GUNao/webots/webotsbridge`. Then, you execute the following.

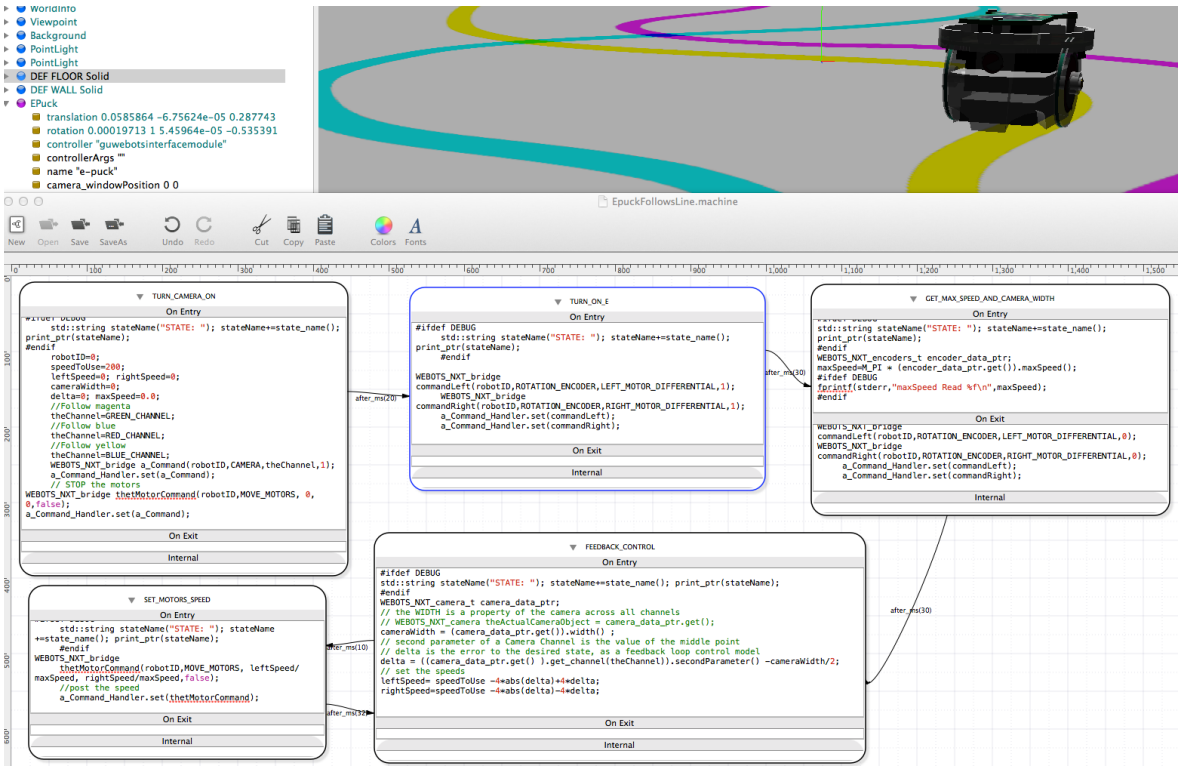

Figure 6: The line follower LLFSM controlling the ePuck in Webots.

the class `WEBOTS_NXT_bridge` that represent messages (information) back and forward between the LLFSM and the execution of the machine in Figure ??.

7.2 Machine to control the line follower — an arrangement of machines

Complexity of behaviors can be created by composing simple LLFSM into more complicated behaviors, and those in turn, into more complicated behaviors. Thus, `clfsm` can execute more that one LLFSM concurrently, and under a sequential, predictable schedule. This is advantageous as many of the complexities of race conditions, concurrent programming and critical section synchronization are avoided. If you provide more than one LLFSM to CLFSM, they become an arrangement. In the arrangement, the execution token is given to the first machine in the arrangement to execute one ringlet. The it passes to the next one for also just one ringlet, and so on, until it reaches the last one, who in round-robin fashion passes the execution token to the first one. Each machine in turn is executed for one and only one ringlet. See Section ?? on concurrency.

Thus, now we are going to create a controller machine for the previous machine. This machine will be very simple as we just want to illustrate the construction of such hierarchy. It is very similar to the famous subsumption architectures of Brooks [?]. The CLFSM tool has the capacity to easily build this with some techniques.

```
export WEBOTS_HOME=/Applications/Webots
make host
cp ./build.host/guwebotsinterfacemodule MiPalDifferentialEPuck/controllers/guwebotsinterfacemodule/
```

The MiPal environment has copied the Webots world of the colored lines and the ePuck in `$MIPAL_HOME/src/MiPal/GUNao/webots/webotsbridge/MiPalDifferentialEPuck/worlds` with the name `novice_linear_camera.wbt`.

7.2.1 The SUSPEND state

For each LLFSM, it is always possible to create a designated state to park the machine. We name the state SUSPEND, although the name of the state is not what determines if it is the state where to park the machine. However, only one designated state of this kind can be in a machine. Similar with the initial state, there can only be one per machine.

If a machine has this state, it also has (implicit) transitions from all of its states to this state. Automaticly, each state has such transition evaluated as the very first transition in its sequence of outgoing transitions. CLFSM macros include the function

`suspend(`*`const string NameOfMachine`*`)`

The execution of a `suspend` (by another machine) enables the transition to the SUSPEND state in the named machine. When the token of execution arrives to the machine named in the suspend, then the transition actually fires, the current state's **OnExit** section runs followed by the **OnEntry** section of SUSPEND.

There are also implicit transitions out of the SUSPEND state. There is a transition to the last state executed previous the migration to the SUSPEND state. This transition is enabled by the following CLFSM macro.

`resume(`*`const string NameOfMachine`*`)`

Again, when another machine executes a `resume`, it enables the transition back to the previous state. The transition does not fire until the token of execution arrives to the named LLFSM. Then, the **OnExit** section of the SUSPEND state is performed and the machine moves to the earlier state as its current state, also executing the **OnEntry** section of the new state. Thus, the SUSPEND is like any other state, with exactly the same semantics. While suspended, the machine will execute its **Internal** section when the token of execution arrives to it.

Currently CLFSM provides no checking and no implementation for hierarchical control of machine who in turn start other machines. That is, each machine, if suspended, must explicitly suspend other sub-machines it has executing (if that is the desirable behavior). This is usually accomplished by the machine itself issuing `suspend` calls to all the machines is has set in running mode from its SUSPEND state.

When CLFSM starts execution of an arrangement, all machines are running and they start in their initial state. A suspended machine can be brought back to its initial state with the `restart` macro from CLFSM.

`restart(`*`const string NameOfMachine`*`)`

This is completely equivalent to a `restart` except that the target state is not the state that took the machine to the SUSPEND state, but the target state is the initial state.

7.2.2 Testing if a machine is running

CLFSM enables also predicates to evaluate whether some machine is in its SUSPEND state or not. The predicate

`bool is_suspended(`*`const string NameOfMachine`*`)`

can be used as part of the code, and in particular as part of a Boolean expression labeling a transition in a machine to make decision based on whether another machine is suspended or not.

7.3 The DRIVERFORFOLLOWER LLFSM

So, lets construct a new machine that will run the earlier line follower machine (EPUCKFOLLOWSLINE) for only a certain amount of time, just enough to perform the trip over the colored lines, and stop it at the end. Open MIEDITLLFSM and open a new machine called DRIVERFORFOLLOWER. Our goal is to build a machine that has for states, as per Figure ??.

Use the Directories to set the directories as follows.

```
$GUNAO_DIR/posix/gusimplewhiteboard/typeClassDefs
$GUNAO_DIR/posix/gufsm/clfsm
$GUNAO_DIR/posix/gusimplewhiteboard
```

Figure 7: A machine that controls the line follower machine.

And use the Includes to set the include directories for this machine as follows.

```
#include <iostream>
#include <stdio.h>
#include <string>
#include "CLMacros.h"
#define DEBUG
```

For this machine we will not require any variables. So the INITIAL state just suspends the EPUCKFOLLOWSLINE machine. Place the following code in the INITIAL of the DRIVERFORFOLLOWER.

```
#ifdef DEBUG
fprintf(stderr, "STATE: %s\n", state_name());
#endif
suspend("EPUCKFOLLOWSLINE");
```

The next state will actually restart the line follower. So, use the States menu item to add a new state, and name it FOLLOW_LINE. This state has also a simple **OnEntry** section of code: it basicly restarts the sub-machine.

```
#ifdef DEBUG
fprintf(stderr, "STATE: %s\n", state_name());
#endif
restart("EPUCKFOLLOWSLINE");
```

Do not forget to save this state using the Save option of the States menu. Now, use the STATES to go back to the INITIAL and we will add a transition from INITIAL to FOLLOW_LINE. The transition we will add is as follows

```
after_ms(20) && is_suspended("EPUCKFOLLOWSLINE")
```

The `after_ms(20)` ensures that we allow for the token of execution to arrive to the EPUCKFOLLOWSLINE which then goes to its SUSPEND state. And then, the controller itself will not move forward unless such machine is indeed suspended. Now we create a third state named STOP_FOLLOWING. Its **OnEntry** section suspend the EPUCKFOLLOWSLINE machine again. It is the same as the INITIAL state.

```
#ifdef DEBUG
fprintf(stderr, "STATE: %s\n", state_name());
#endif
suspend("EpuckFollowsLine");
```

Let save this state and go back to the FOLLOW_LINE state, so we can put a transition to the state just created. Replace the default `true` for the following expression.

```
after_ms(36000)
```

That is, we will follow the lien for 3.6sec.

The last state will be a state with no transitions out of it. This will terminate the execution of the DriverForFollower machine. Lets name this final state STOP. Its **OnEntry** section is just the debug part.

```
#ifdef DEBUG
fprintf(stderr, "STATE: %s\n", state_name());
#endif
```

The transition from STOP_FOLLOWING to STOP is also the transition that allows some time for the token of execution to arrive at the next machine in the arrangement and ensure it is suspended, before the driver machine moves on.

```
after_ms(20) && is_suspended("EpuckFollowsLine")
```

Recall we need to go back and edit EpuckFollowsLine in order to add a SUSPEND where we will halt the motors. The **OnEntry** section of the SUSPEND for EpuckFollowsLine is as follows.

```
// STOP the motors
WEBOTS_NXT_bridge thetMotorCommand(robotID,MOVE_MOTORS, 0, 0,false);
a_Command_Handler.set(a_Command);
```

So, once you have added this state, then use the Choose suspend menu item of the State to select the new state as the suspended state.

The entire machine EpuckFollowsLine is also illustrated in Figure ??.

To test the arrangements of both machines, you need to run Webots with the `guwebotsbride` as the controller in the environment with the lines of color. Then, run CLFSM with both machines as parameters. Typically, in Webots, you revert the world first. Then, you run the command

```
$CLFSM_HOME/clfsm DriverForFollower.machine EpuckFollowsLine.machine
```

You should get and effect similar to what is displayed in the MiPal classification video[6] for 2014 from the 3:34 (3 mins, 14 sec) till 4:08 (4 mins, 8 sec). You will see also a more advanced project called MiCASE. The additional features of MiCASE are

1. graphical layout of LLFSMs,
2. editing more than one LLFSM,
3. syntax checking of C++ code,
4. on-board simulation of CLFSM, and
5. tracking of machine-submachine hierarchies.

[6] The MiPal 2014 classification video is at vimeo.com/mipalgu/robocup2014qual and youtu.be/cSLK5EyKtik.

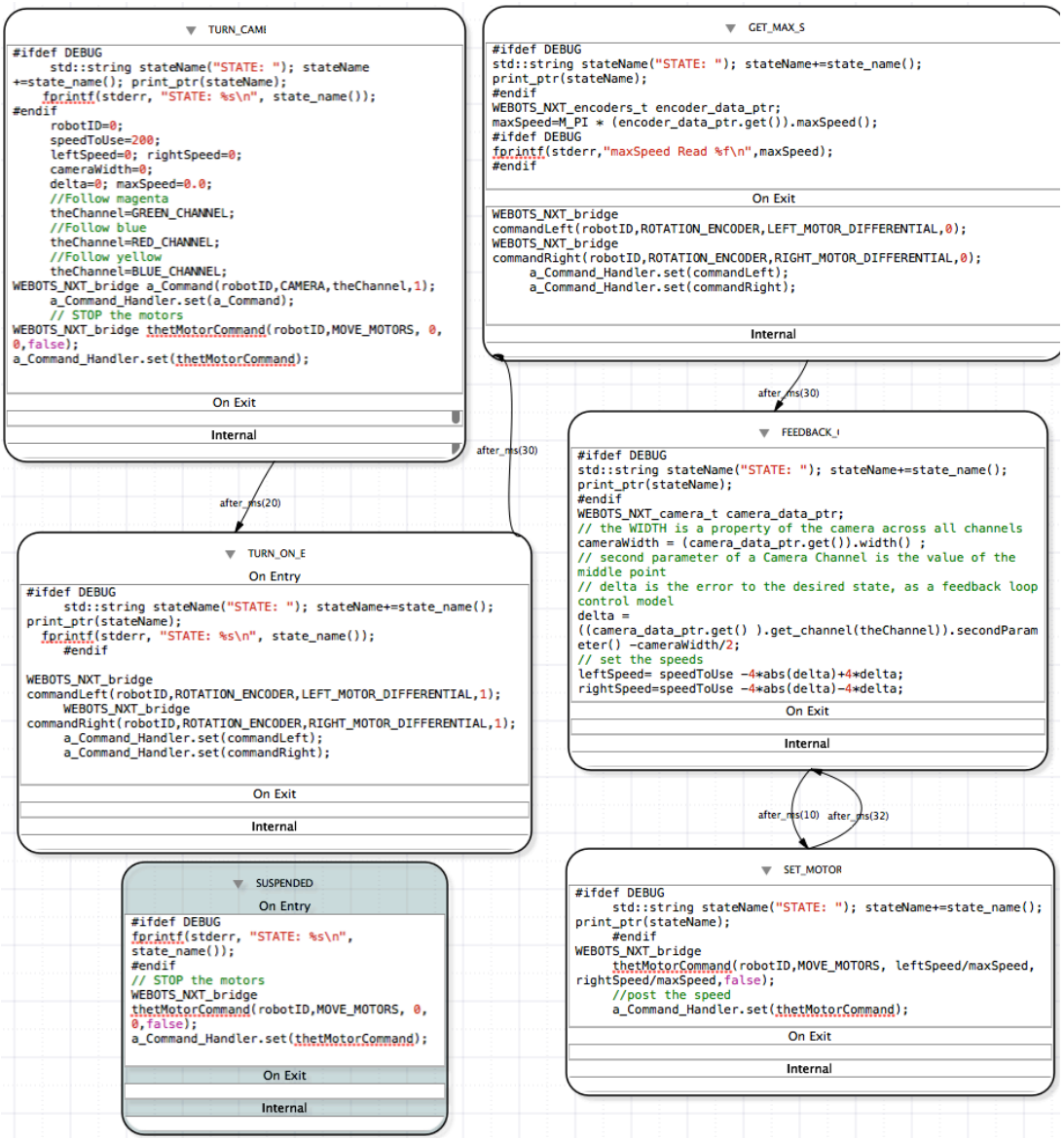

Figure 8: The EpuckFollowsLine with its state for when suspended.

8 A behavior to avoid obstacles on the Nao humanoid robot

In this section we use the MiPal infrastructure for the Nao robot to create a simple behavior that allows the robot to walk about, and in the process avoid obstacles detected by its sonar sensors.

We will need an arrangement of machines.

- BatNaoMoves is the machine that using the sonar decides whether to walk straight or turn as it senses an obstacle. It works in collaboration with another machine: SMButtonChest.
- SMButtonChest is a machine that reads the chest button sensor of the Nao and updates objects of the MiPal-class `NAO_States`. This enables the demonstration to be operated by a person. The walk about will start when the person presses the chest button and it is paused when the button is pressed again. The robot kneels when the demonstration behavior is paused.
- GetUp will run the recorded motions to get up. Selecting different motions according if the robot fell on its back or on its chest.
- RobotPosition reads the inertia sensors and also updates the MiPal-class `NAO_States` to indicate a stance of the robots posture.
- FallManager ensures the posting to the whiteboard of posture information is suspended while the recorded motion to get up is going on.
- BatManController is the top machine that chooses between the walk-about of BatNaoMoves and the FallManager, on the basis of whether a fall has happened or not.

It should be possible to run a Nao just with the two machines BatNaoMoves and SMButtonChest, but then, if the robot falls down, it cannot get up again. So, a human operator has to pick it up.

8.1 The walk about behavior

Thus, lets create a new machine called BatNaoMoves. This is accomplished by starting MiEditLLFSM:

```
java -jar MiEditLLFSM.jar
```

Use the File and select the option New. You can navigate your directories in your file system. You double-click to go into a directory. Once you have selected a directory, you place the machine we are constructing by clicking OK. You will be asked for the name of the machine, type BatNaoMoves. You will be positioned in the INITIAL state of the machine.

We first complete the INCLUDES section fo the machine. These are the necessary `#includes` for any functionality you are to include among the C++ code that will be placed in the executable sections of the machine. For this machine, enter the following lines in the box that appears, and when complete, click on SAVE (take care not to leave black-space characters between the beginning of the line and the # indicating the pre-processor directive).

```
#include <iostream>
#include <stdio.h>
#include <string>
#include <cmath>
#include "CLMacros.h"
#include "CLWhiteboard.h"
#include "typeClassDefs/SENSORS_SonarSensors.h"
#include "typeClassDefs/MOTION_Interface.h"
#include "typeClassDefs/WALK_ControlStatus.h"
#include "typeClassDefs/NAO_State.h"
using namespace guWhiteboard;
#define DEBUG
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wc++98-compat-pedantic"
```

The CLMacros.h enables some functions of clfsm, like testing if a machine is suspended, or using the function after_ms() in a transition so that a certain number of mili-seconds occurs before the transition can fire.

The file CLWhiteboard.h enables access to the MiPal whiteboard, while the includes under typeClassDefs are the message types (classes) for specific MiPal modules to the Alderbaran's Nao.

We also need the paths to the directories to find such includes. For this, we need to use the DIRECTORIES button and open where clfsm searches for compilation of the machine. This machine requires you complete the installation of MiPal tools for the Nao (or you obtain a copy of the Linux 14.3 machine). Those instructions let you know how to set the environment variable GUNAO_DIR. Typically, GUNAO_DIR is set with

```
export GUNAO_DIR=$HOME/src/MiPal/GUNao
```

The content of your DIRECTORIES seciton should be as follows.

```
$GUNAO_DIR/Common
$GUNAO_DIR/posix/gusimplewhiteboard
$GUNAO_DIR/posix/gusimplewhiteboard/typeClassDefs
$GUNAO_DIR/posix/gufsm/clfsm
```

The next step is to declare the variables that are global to all states of this machine. You open the window to declare variables by clicking on the button VARIABLES. A screen that shows a matrix with 3 columns and several rows opens. You enter the type of the variable in the first column, the variable name in the second column and you can add an optional comment in the third. Enter the following variables.

```
QSay_t                SAY
SENSORS_SonarSensors_t sensorHandler      whiteboard handler for sonar
SENSORS_SonarSensors   sensorValues
MOTION_Status_t        motion_status_handler
MOTION_Commands_t      motion_ptr         handler for motion commands
NAO_State_t            nao_state_ptr      handler to obtain the Nao state from the whiteboard
NAO_State              nao_state          message with information on button on the Nao
WALK_Command_t         walk_post
int                    sonarRight
int                    sonarLeft
```

Now that the global element of the machine are defined, we build the machine by adding states and contents to the states. Lets start with the **OnEntry** section of the INITIAL state. Place there the following code.

```
#ifdef DEBUG
fprintf(stderr,"STATE: %s\n",state_name());
#endif
say("Bat Man");
```

This code enables to output into the console the name of the state using the clfsm macro state_name(). This is useful when debugging a machine and you want to trace the states its is going trough. The say("Bat Man"); uses the MiPal interface to the MiPal speech module for the Nao. It is an abbreviated way of using the handler QSay_t say so that the robot speaks the words *Bat Man* when reaching this state.

Now, use the Add option of the States to add the second state of this machine. We will call it START_SONAR. Although, arrival into this state will cause the robot to get up. Place the following code in the **OnEntry** section of this new state.

```
#ifdef DEBUG
fprintf(stderr,"STATE: %s\n",state_name());
#endif
say("Sonar values");
MOTION_Commands motion;
motion.GoToStance(Motions::Kneeling_stance, Motions::Standing_stance);
motion.set_head_stiffness(true);
motion_ptr.set(motion);
```

The print statement should now be familiar, it is a debugging tool for tracing the progress of the machine along its states with output to the console. The `say` should not be a surprise, it enables tracking progress as well, but usually we do not do it for every state, as machines progress trough states much faster that the time it takes to say these messages.

The interesting aspect is the use of the MiPal interface for pre-recorded motion. We create a message-type of the class `MOTION_Commands`. We set some properties of this message, like the fact that we desire a change of stance from the `Kneeling_stance` to the `Standing_stance`. We also set the property that the head stiffness should be high. The line `motion_ptr.set(motion);` actually places the command in the whiteboard using the handler we declared in the variables earlier.

We will also place code in the **Internal** section of the START_SONAR. This code will read the sonar values.

```
sensorValues=sensorHandler.get();
sonarLeft = int ( sensorValues.sonar(Sonar::Left0));
sonarRight = int (sensorValues.sonar(Sonar::Right0));
#ifdef DEBUG
fprintf(stderr, "LEFT: %d RIGHT %d\n", sonarLeft, sonarRight);
#endif
```

This code uses the handler of the sonar declared in the machine variables and uses the `get`-message mechanism to obtain a `sensorValues` object from the whiteboard. The values of the left sonar sensor and the right sonar sensor are extracted using the MiPal interface for this message-type. That is, this class has a method `sonar` and using the predefined constants `Sonar::Left0` and `Sonar::Right0`, we obtain the actual values. For debugging purposes, such values are printed to the console. This section will be executed as long as the machine remains in this state.

We now create a third state where we actually set the robot walking forward and we keep track of the sonar values. So, we refer to this state as WALK_ABOUT. Again, in MiEditLLFSM you add a state using the Add option of the menu States. Then, you enter the name of the desired state.

In the **OnEntry** section of WALK_ABOUT you enter the following code.

```
#ifdef DEBUG
fprintf(stderr,"STATE: %s\n",state_name());
#endif
walk_post(WALK_Ready);
protected_msleep(15);
walk_post(WALK_ControlStatus(WALK_Run, 35, 0, 0, 5));
sensorValues=sensorHandler.get();
sonarLeft = int ( sensorValues.sonar(Sonar::Left0));
sonarRight = int (sensorValues.sonar(Sonar::Right0));
#ifdef DEBUG
fprintf(stderr, "LEFT: %d RIGHT %d\n", sonarLeft, sonarRight);
#endif
```

This machine will regularly read from the whiteboard the `nao_state` in order to detect if the chest button has been pushed and we are at the end of the walk.

Thus, the corresponding handler is used to perform a `get`-Message. This is the line `nao_state = nao_state_ptr.get();` which we will repeat in internal sections and on-exit sections in this and further states. We need to set the walk interface ready. This is the `walk_post(WALK_Ready);` followed by a 15 milliseconds pause to enable the DCM cycle (at 10ms) to pick this up. Then, the command actual walk command is placed into the whiteboard with `walk_post(WALK_ControlStatus(WALK_Run, 35, 0, 0, 5));`. The reminder of this sections are debugging commands for the console about arriving to this state and the readings of the sonar values.

It is important that the view of the world is regularly updated, so this state must have the following code in both, its **Internal** section and its **OnEntry** section

```
nao_state = nao_state_ptr.get();
sensorValues=sensorHandler.get();
sonarLeft = int ( sensorValues.sonar(Sonar::Left0));
sonarRight = int (sensorValues.sonar(Sonar::Right0));
```

We have not introduced any transitions. We will leave those till the end. We will now create the state where the robot walks turning left.

So again, using the Add option of the menu States add a new state called TURN_LEFT. There should be no surprises here. The code of the **OnEntry** section is as follows.

```
#ifdef DEBUG
fprintf(stderr,"STATE: %s\n",state_name());
#endif
nao_state = nao_state_ptr.get();
walk_post(WALK_Ready); protected_msleep(15);
walk_post(WALK_ControlStatus(WALK_Run, 0, 0, -1, 5));
sensorValues=sensorHandler.get();
sonarLeft = int ( sensorValues.sonar(Sonar::Left0));
sonarRight = int (sensorValues.sonar(Sonar::Right0));
```

This different parameters on the `WALK_ControlStatus` object result in the robot spinning on its place counter-clockwise. The **OnEntry** and the **Internal** sections of this state are also only updating the sensor values (for sonar and for the chest button)

```
nao_state = nao_state_ptr.get();
sensorValues=sensorHandler.get();
sonarLeft = int ( sensorValues.sonar(Sonar::Left0));
sonarRight = int (sensorValues.sonar(Sonar::Right0));
```

Now add the state TURN_RIGHT, which has the same **OnExit** and **Internal** sections that update the sensors, and also almost exactly the same **OnEntry** state, but now use `WALK_ControlStatus(WALK_Run, 0, 0, 1, 5)`. The change to a positive 1 as the third parameter makes the robot turn right, and spin in clockwise direction.

We will now make a state where the robot will stop and kneel down again. This will happen if the chest button is pressed or if the sonars detect something very very close, so it is actually safer to stop. This state will be named GAME_OVER. The **OnEntry** section is as follows.

```
#ifdef DEBUG
fprintf(stderr,"STATE: %s\n",state_name());
#endif
say("Game Over");
nao_state = nao_state_ptr.get();
walk_post(WALK_ControlStatus(WALK_Disconnect));
protected_usleep(30000);
MOTION_Commands motion;
motion.GoToStance(Motions::Standing_stance, Motions::Kneeling_stance);
motion_ptr.set(motion);
protected_usleep(2000000);
MOTION_Commands motion2(false, false, true);
motion_ptr.set(motion2);
```

The crucial parts of this code are disconnecting the walk engine so we can issue pre-recorded motion. This is done with `walk_post(WALK_ControlStatus(WALK_Disconnect));` followed by a pause of 30 ms or

30,000μs. Then, a motion command is issued to apply pre-recorded motion to make the robot kneel. We also wait approximately 2 seconds for this to complete, before issuing a command to remove stiffness in all motors. Note that is dangerous in the MiPal infrastructure to send to messages of the same class-type in rapid succession, but here the two motion commands are separated by 2 seconds.

The **Internal** and the **OnEntry** sections of this ate are very simple, we just update the status with respect to touch/pressure buttons with

```
nao_state = nao_state_ptr.get();
```

Finally, we will create another state where we can detect another chest button pushed and restart the robot walking about. This state is called END and has a very simple **OnEntry** section.

```
#ifdef DEBUG
fprintf(stderr,"STATE: %s\n",state_name());
#endif
say("The End");
nao_state = nao_state_ptr.get();
```

The **Internal** and **OnExit** sections are also as follows.

```
nao_state = nao_state_ptr.get();
```

The last state we will add is called SUSPENDED. This is a state we will park this amchine when the robot has fallen. That is, a p[rent amchine will suspend this machine and send it to the state SUSPENDED. There can be only one suspend state (same as the initial state), although the name is arbitrary. You add it as any other state using the menu States and then, you can highlight it as the suspend state by the option Choose Suspend, that enables to select a state as a suspended state. This state has a very simple code in its **OnEntry** section:

```
walk_post(WALK_Disconnect);
protected_usleep(30000);
```

Save the statem and now we have all states for this machine.

So, now we are ready to add all the transitions. We will start adding the transitions from the INITIAL state to the START_SONAR state. So, one needs to change what is the current state. Use the button STATES and the list of all states should come on on a dialog window. In this case, leave the INITIAL and chose OK. This should make the INITIAL the current state. To add a transition, use the Transitions and chose the option to Add. A window with potential target states comes up. Chose the START_SONAR as the target. Now, edit the default expression (`true`) to the expression `after_ms(2000)`. This means we will move form INITIAL to START_SONAR after 2 seconds (you can also use `after(2)`.

Now the transition from START_SONAR state to the WALK_ABOUT state. So, we need to use the button STATES and select the state START_SONAR. Then, once this is a current state, we need to select a target state. This requires to use the Add of the Transitions menu. Here we will allow 4 seconds for the robot to reach standing up and we will also test that the status of the motion has completed. So, the transition is

```
after_ms(4000)&& !  motion_status_handler.get().isRunning()
```

This should be the replacement of the default Boolean expression for this transition.

Note the advantages of messages being objects, we take advantage of dynamic binding and to the object we get using the get-Message on the handler, we directly request whether it is running using the method `isRunning()`.

Now, where we have more than one transition out of the same state is for the WALK_ABOUT. We will spin the robot if we are closer than 50cm from an obstacle. So, lets make the WALK_ABOUT state the current state (use the STATE). We first add the transition to the TURN_RIGHT. The transition is `sonarLeft < 50`.

We also add a transition with target the state TURN_LEFT. This transition is when we are closer than 50cm on the other sensor. Namely, `sonarRight < 50`. Finally, we place a third transition out of the WALK_ABOUT state. This time the target is the state GAME_OVER. The expression is as follows.

```
nao_state.chest_pressed() || (sonarLeft < 28) || (sonarRight < 28)
```

This ensures that if we are too close to an obstacle we stop, also, if the chest button is pressed, the robot will stop and kneel.

Change the current state to the TURN_LEFT. In this state we also want to stop if a chest button is pressed of we are too close to an obstacle while spinning. So, with target state the state GAME_OVER we add the following transition.

```
nao_state.chest_pressed() || (sonarLeft < 21) || (sonarRight < 21)
```

We use now 21cm, somewhat less than when walking forward, because spinning around is OK to be a bit closer to obstacles, while walking straight requires slowing down the direction into the obstacle. We need a transition to go back walking straight when the obstacle is no longer in front. So, form the state TURN_LEFT where we adding transitions now, we add a second transition with target WALK_ABOUT.

```
after_ms(500) && ( sonarLeft > 50 )
```

This ensures we always spinning at least half a second when we have detected an obstacle.

Now, if we make the current state the state TURN_RIGHT, then we add two analogous transitions also to the corresponding target states. Thus, from TURN_RIGHT to GAME_OVER we add the following transition.

```
nao_state.chest_pressed() || (sonarLeft < 21) || (sonarRight < 21)
```

And also, from TURN_RIGHT to WALK_ABOUT.

```
after_ms(500) && ( sonarRight > 50 )
```

the transitions that test $sonarRight > 50$ seem to be the other way around in the example code, need tot est this.

There is just two more transitions to go. The next one goes from the state GAME_OVER, so make this the current state by using the States and choosing the option Add again in the Transitions menu. Then, select the state END as the target. Edit the default expression `true` to the following expression

```
!nao_state.chest_pressed() && after_ms(4000) && !  motion_status_handler.get().isRunning()
```

This checks the chest button on the robot is no longer pressed, 4 seconds have gone by, so the robot kneels, and also the motion has stopped (it is not running.

The last transition now has END as a source and INITIAL as a target. It enables a reset of the behavior. Simply, if the chest button is pressed.

```
nao_state.chest_pressed()
```

After all this transitions, the LLFSM called BatNaoMoves should have transitions as per Fig. ??.

It also, should have state activities similar to those of Fig. ??

You can run this machine on its own; without any other machines. You need a Nao set up with the fundamental MiPal modules and be able to cross-compile the machine. Then place the compiled machine into the robot. Read the MiPal *Getting Started* document. Test the machine. Of course, the machine does not end, and without the machine that control the chest button, once the robot kneels again (because you put your hand very very close to its sonar), the behavior is basicly stuck in the END.

So, the next step is to include the machine that collects the inputs of the pressure/touch sensors.

8.2 A machine to post to the whiteboard from the chest button of the Nao

The states and transitions of this state machine are shown in Fig. ??.

Some of the code in the states of the machine shows the philosophy on using the MiPal whiteboard in a manner that uses a get-Message (a pull approach) over using a subscribe (a push-approach). For example, the **OnEntry** section of the Button_Off has the following 4 lines of code.

```
SENSORS_LedsSensors_t leds_ptr;
SENSORS_LedsSensors led = leds_ptr.get();
led.LEDsGroupChange(Chest, Red);
leds_ptr.set(led);
```

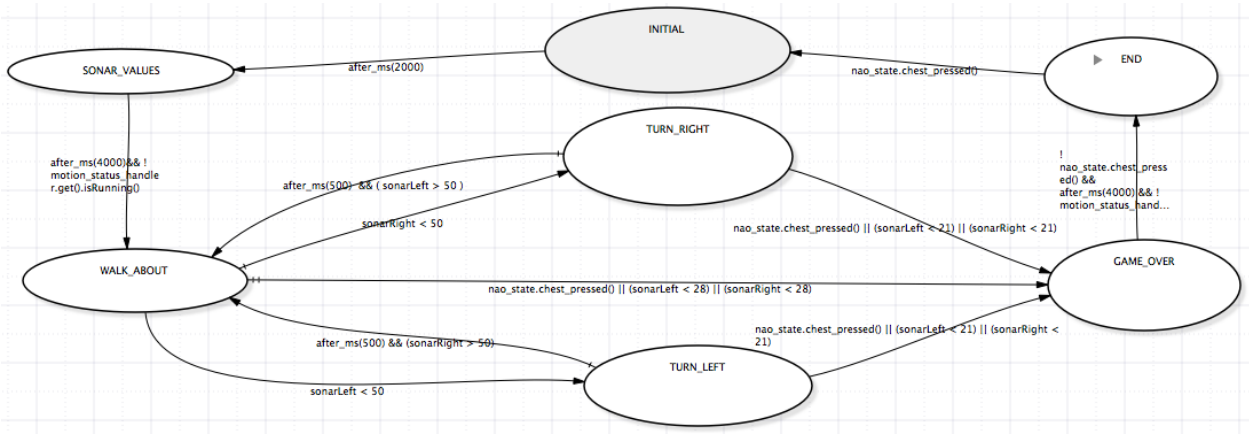

Figure 9: The states and transitions of the BatNaoMoves machine.

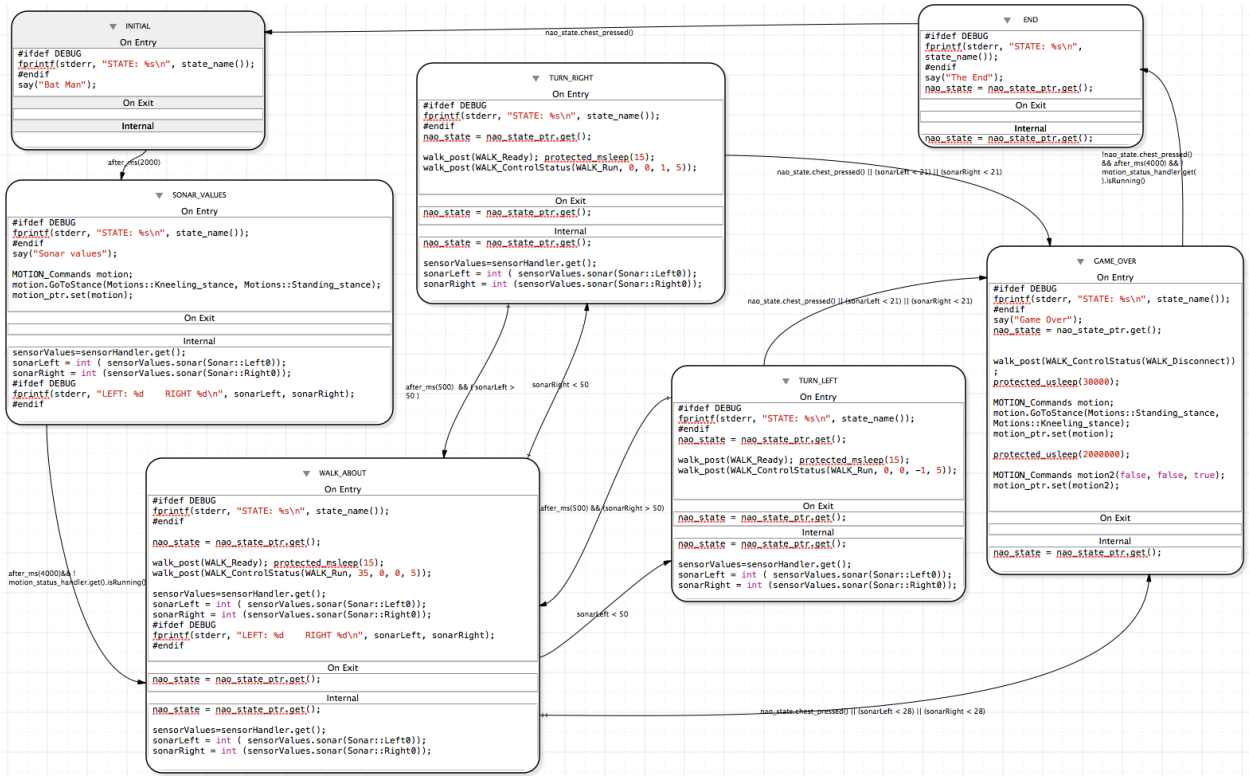

Figure 10: The inside of the states of the BatNaoMoves machine.

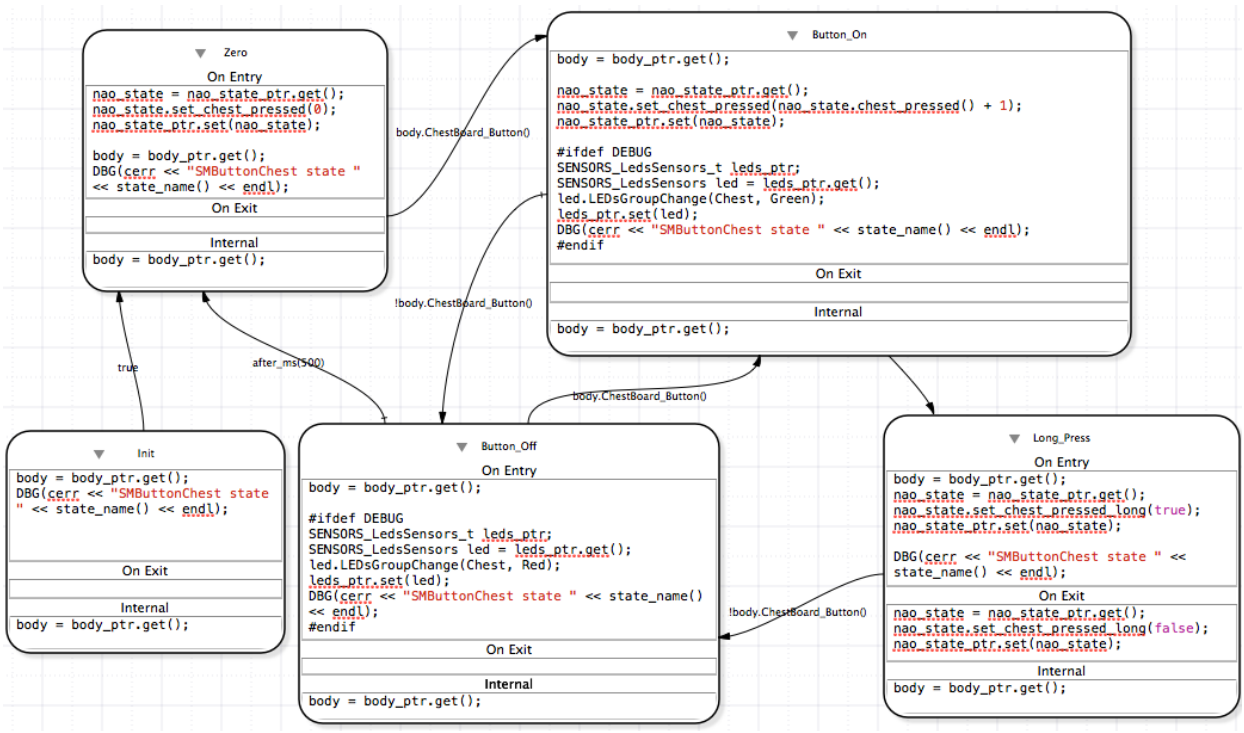

Figure 11: The inside of the states of the SMButtonChest machine.

The first line defines a whiteboard handler for a message of the class-type `SENSORS_LedsSensors`. The second line of code actually carries out the `get`-Message. That is, a copy from the whiteboard is now in the variable of the machine. Then, the third line of code updates the status in the local copy: `led.LEDsGroupChange(Chest, Red);`. Finally, the fourth line moves the local copy using the same handler back into the whiteboard. This technique eliminates all sorts of concurrency issues regarding the same class-type of messages because, as long as it is only LLFSMs operating on the whiteboard, the sequential scheduling by CLFSM guarantees no race conditions for the data/messages on the whiteboard.

We invite you to build this machine yourself. Restart MiEditLLFSM and create a new machine. Use the INCLUDES to create the following includes for compilation of all states in the machine

```
#include <iostream>
#include <stdio.h>
#include <string>
#include <cmath>
#include <gu_util.h>
#include "CLMacros.h"
#include "typeClassDefs/SENSORS_BodySensors.h"
#include "typeClassDefs/SENSORS_LedsSensors.h"
#include "typeClassDefs/NAO_State.h"
#include "CLWhiteboard.h"
using namespace guWhiteboard;
using namespace std;
//#define DEBUG
```

Use the DIRECTORIES to specify the following directories to search for include paths at compilation.

```
$GUNAO_DIR/posix/gusimplewhiteboard/typeClassDefs
$GUNAO_DIR/posix/gufsm/clfsm
$GUNAO_DIR/posix/gusimplewhiteboard
$GUNAO_DIR/Common
```

To complete the elements common to all states you need to include the variables common to all states. Use the VARIABLES button and add the following declarations.

```
SENSORS_BodySensors   body
SENSORS_BodySensors_t body_ptr
NAO_State_t           nao_state_ptr
NAO_State             nao_state
```

Again, the class-type message `NAO_State` is the most crucial as it is the real output of this machine. It is initially set to zero in the **OnEntry** section of the state Zero. It is set in 3 lines of code.

```
nao_state = nao_state_ptr.get();
nao_state.set_chest_pressed(0);
nao_state_ptr.set(nao_state);
```

Again, the methods is to use pull-technology; that is a `get`-Message as opposed to a subscribe. So the first-line obtains the current copy in the whiteboard, and gets a local object. The second line uses essentially a property-setter of the class-type to set the number of presses received in the chest button to zero. The third line, places the object back into the whiteboard using the object handler.

So, the way this machine works is that in the state Button_On, the count is incremented by the **OnEntry** section. This happens when we detect a button pushed (the transitions labeled `body.ChestBoard_Button()`) and we go out of this state when the button is released the transition `! body.ChestBoard_Button()`. However, if more than 500ms (that is half a second) goes by, we go into the state Long_Press. In this state, once again, three lines of code of the its **OnEntry** section modify a property of the `nao_state` object on the whiteboard. This again, is made by pull-technique.

```
nao_state = nao_state_ptr.get();
nao_state.set_chest_pressed_long(true);
nao_state_ptr.set(nao_state);
```

That is, we do a `get`-Message first to obtain information (an object) from the whiteboard into the machine. Then, the object of the class-type `NAO_State` is modified with a setter of such class-type. Then, the handler to the whiteboard is sued to place it back on the whiteboard, for other machines to use. In particular, the earlier BatNaoMoves machine.

Also, this state Long_Press is finished when the button is no longer being pressed. In that case, the information on the whiteboard about a long-press being in effect is offset. This happens on the **OnExit** section of the state Long_Press.

```
nao_state = nao_state_ptr.get();
nao_state.set_chest_pressed_long(false);
nao_state_ptr.set(nao_state);
```

Again, the methods is a pull-approach that gets the object out of the whiteboard, sets a property of the object, and places it back on the whiteboard.

With these two machines, you should be able to create a behavior that is quite fun to watch. The robot walks about avoiding obstacles. As long as it does not fall down, you can stop it by pressing the chest button and also re-start it by pressing the chest button.

8.3 A super-machine to regulate when to run the get up behavior

We will create a new machine that basicly runs the BatNaoMoves machine when the robot is standing, or runs behaviors to get the robot up if it has fallen. The idea is rather simple (see Fig. **??**). In the Init state (which is the initial state) all sub-machines are suspended and after 20ms, which shall be enough to ensure they will not run, `but perhaps the order of the machines in the invocation to` CLFSM `matters`. the order is given to resume the BatNaoMoves machine. However, at all times, the state of the robot is being monitored by obtaining (using a pull-method) the state of the robot; namely an object of the class `NAO_State`. This is why we have, in the **OnEntry** and in the **Internal** sections the following code.

```
nao_state = nao_state_ptr.get();
```

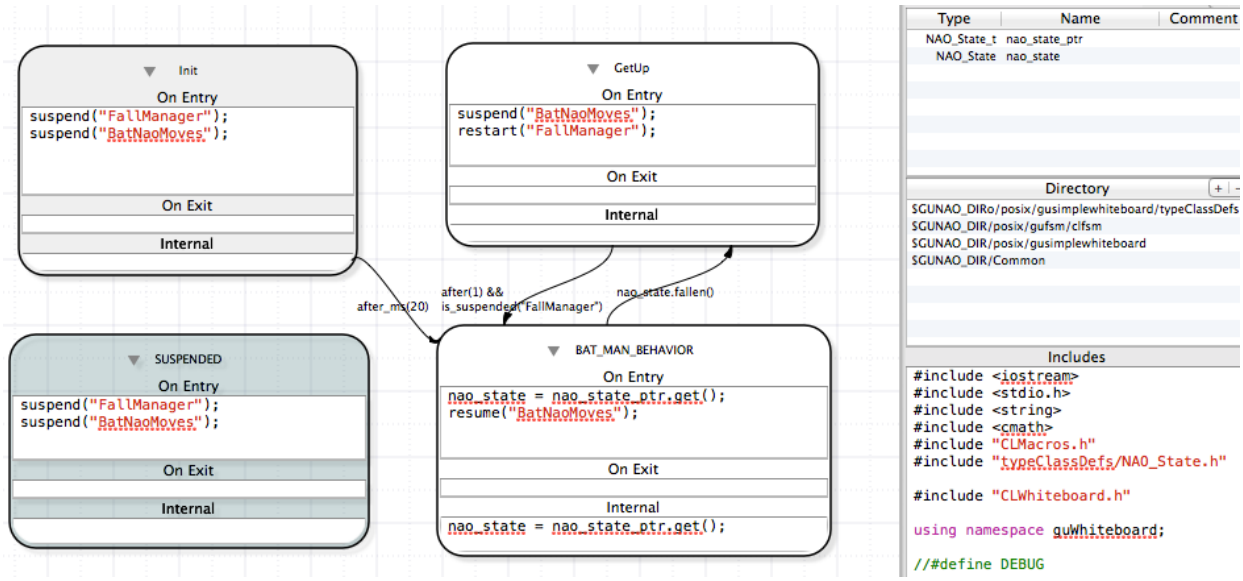

Figure 12: The inside of the states of the BatManController machine.

The transition out of the state BAT_MAN_BEHAVIOR to GetUp is `nao_state.fallen()`. This will be true if the robot is down.

The **OnEntry** section of the GetUp state suspends the BatNaoMoves machine and re-starts the FallManager machine. Thus, we move on to specify the FallManager machine.

8.4 The FallManager machine

This machine in turn manage one other submachine (see Figure ??). There are some important things as you use MiEditLLFSM to build the FallManager machine. First, the initial state should be the INITIAL. Also, very **important**, this machine assumes that some other machine has started the machine RobotPosition. And also interesting, this machine, when it achieves its job, it arrives at its suspend state, where it ensures the GetUp is suspended. Because this machines arrives by itself to is suspend state, we need a transition out-of and into the state SUSPENDED, so this state is not considered a temrinating final state.

The intuition of the FallManager machine is that when it restarts, it is because the robot has already fallen, and the RobotPosition has established in the whiteboard what sort of position the robot is in. Once RobotPosition is suspended, we can restart the GetUp. Once the machine GetUp completes, we shall be in the state GET_UP_FINISHED. The way the machine is drawn, there are 3 passes over the state CheckStance where if the robot is not standing, then the machien will go to the state RunGetUp. However, perhaps this machine can eb simplified, with just waiting for 1.5 seconds instead of 3 times for half a second. Moreover, the `stability` variable is not set to zero in the loop of the state RunGetUp. So, this machine gives up attempting to get up in 3 trials.

8.5 The machine to determine the orientation of the robot

The machine RobotPosition monitors sensors and according to them sets a status variable in the object of class-type `NAO_State`. The machine is rather simple but its illustration (refer to Figure ??) needs some explanation. In its initial state (the state Init) two variables are given values. The intention is to treat these variables as constants in the machine.

The crucial element are the transitions. And also that the transitions must be executed in order. Note that each transition does a get-Message. Perhaps all of these get-Messages should be in the state Check, and updating a couple of variables. You may try that. Each transition verifies a condition of how the robot

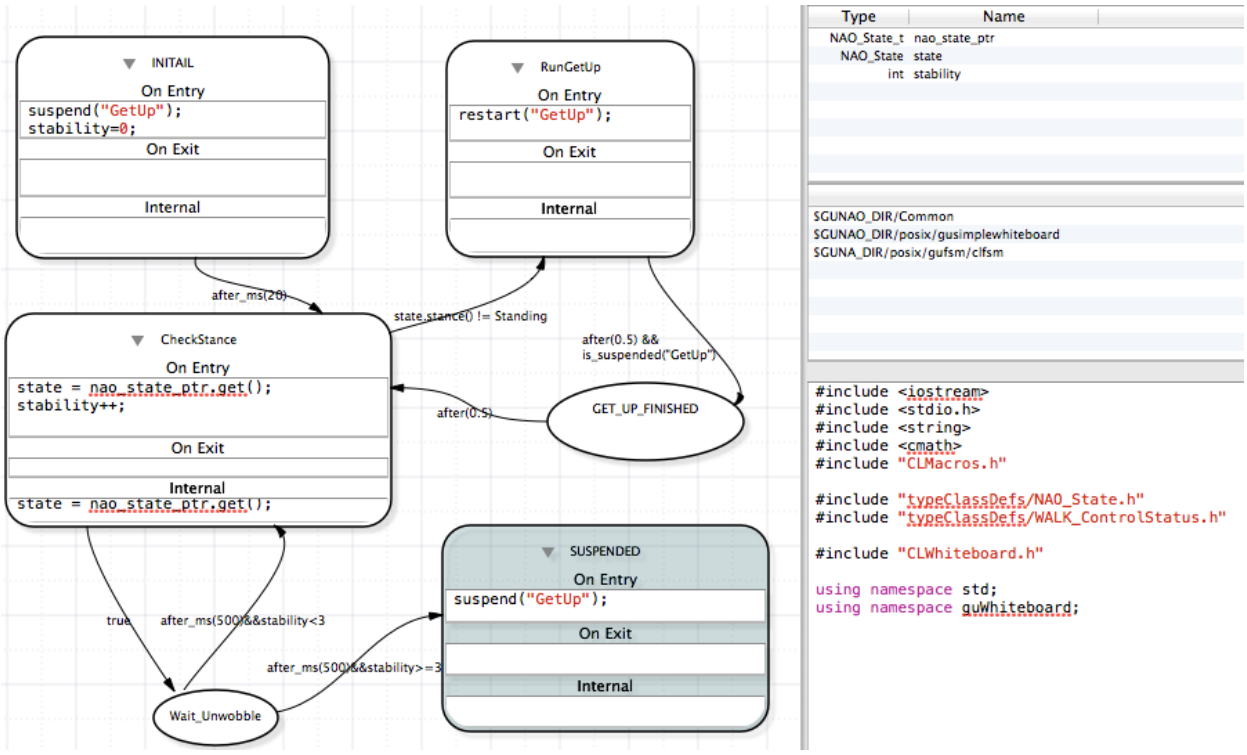

Figure 13: The inside of the states of the FallManager machine.

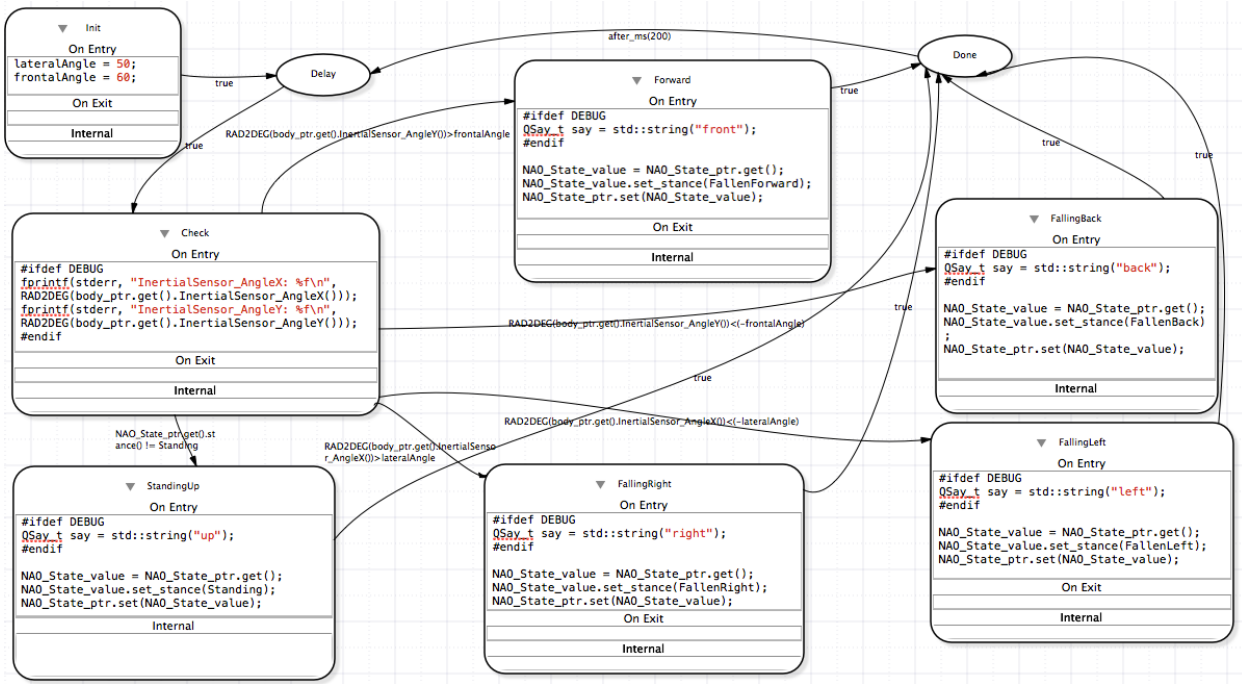

Figure 14: The inside of the states of the RobotPosition machine.

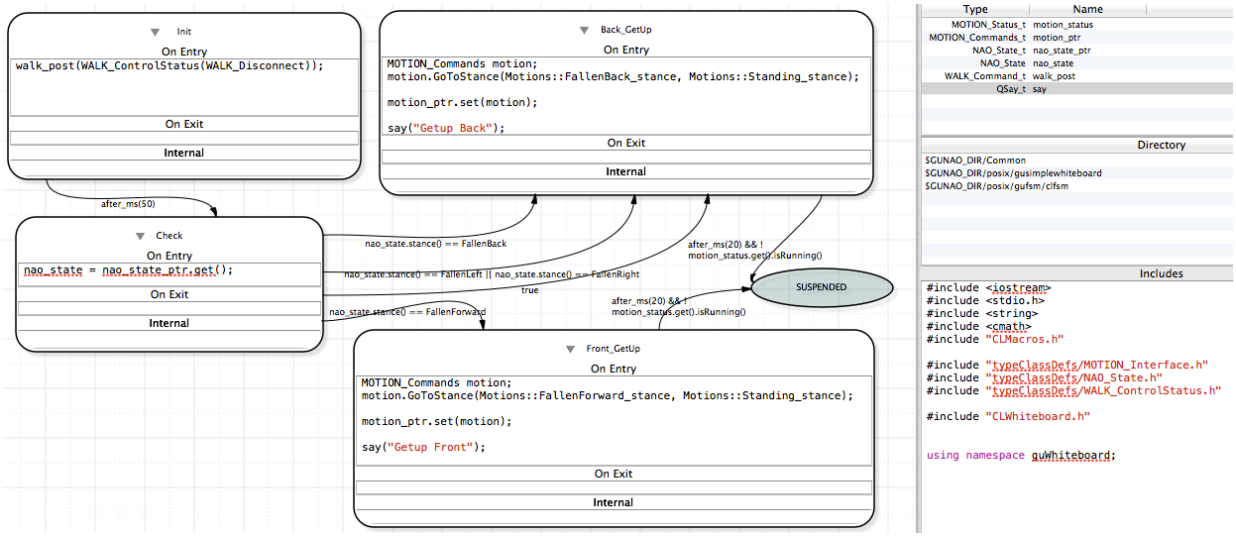

Figure 15: The inside of the states of the GetUp machine.

is lying down. The corresponding state uses the pull-method to update the stance accordingly in the object of class-type `NAO_State` on the robot. For example, the state FallenBack has the following 3-lines of code.

```
NAO_State_value = NAO_State_ptr.get();
NAO_State_value.set_stance(FallenBack);
NAO_State_ptr.set(NAO_State_value);
```

That is, we use the handler `NAO_State_ptr` to obtain an object of class-type `NAO_State` into the variable `NAO_State_value`. We then use a setter in this class to change the property:

```
NAO_State_value.set_stance(FallenBack);
```

The third line of code places the object back into the whiteboard. All other states are analogous here. What is crucial is the order of the transitions. The last transition to evaluate is the transition from the state Check to the state StandingUp. It has the form

```
NAO_State_ptr.get().stance() != Standing
```

And thus, if none of the other transitions has fired, then it declares the robot standing! Thus, the order of the transitions is as follows.

```
RAD2DEG(body_ptr.get().InertialSensor_AngleY())<(-frontalAngle)
RAD2DEG(body_ptr.get().InertialSensor_AngleX())>lateralAngle
RAD2DEG(body_ptr.get().InertialSensor_AngleX())<(-lateralAngle)
RAD2DEG(body_ptr.get().InertialSensor_AngleY())>frontalAngle
NAO_State_ptr.get().stance() != Standing
```

The target states are FallingBack, FallingRight, FallingLeft, FallingForward, and StandingUp.

8.6 The machine to get up

This machine is also rather simple. However, somewhat delicate. `this machine has true transition if the robot has not fallen forward, making some of the other transitions to Back_GetUp very redundant.` Its initial state is Init. Here, the walk engine is disconnected so we can execute pre-recorded motions. The motions to get-up from falling on its back, or get up, from falling on its chest are pre-recorded motions. The state Check obtains using a `get`-Message a `NAO_state` object. The `stance` property is tested in order to decide how to get up. Until the motion does not complete, which is the predicate

```
!motion_status.get().isRunning()
```

this machine does not achieve its suspended state. To build this machine correctly you need to label the last state as the suspended state.

```
it is unclear when the walk engine is connected again.
```

9 Further work

There should be a way to declare constants that have the scope of all states. Perhps ina seciton similar to the variables that have scope of the states of a machine.