How to use gusimplewhiteboard with ROS

Vladimir Estivill-Castro
MiPal

October 28, 2014

IMPORTANT: We will be using catkin, but this is just for package management. We will actually not use ROS (none of the middleware for ROS is required here).

Examples of logic-labelled finite-state machines using gusimplewhiteboard

To help you use clfsm with ROS we have two examples.

1. The first example is simply a publisher and a subscriber. This will enable to see how to set-up a new message-class, and use the gusimplewhiteboard.

2. The second example consist of creating the driver for this messages to be used in creating a behavior for a ePuck under Webots [1] with ROS.

3. A third example is a a machine for an ePuck that follows a colored line in Webots. The example uses a cleaner API from the gusimplewhiteboard.

Extending the messages and a publisher/subscriber example

The first thing to extend the *MiPal* messages already available is to download the package guwhiteboardtypegenerator.
This will enable you to extend the class-oriented messages that gusimplewhiteboard knows about and thus, you can create new message types. We also are going to need the file guwhiteboardtypelist.tsl. In summary, from the *MiPal* website, in the download section, you will need.

1. The package guwhiteboardtypegenerator that constructs the infrastructure for class-oriented messages

[1] We require some knowledge with Webots. From the *Webots User Guide* you should read Chapter 1 *Installing Webots* (www.cyberbotics.com/guide/). **Warning;** Webots documentation is inaccurate and has inconsistencies from version to version; you just simply have to work around this. Then, from the *Webots User Guide* we recommend Chapter 2 *Getting Started with Webots*. Then, we believe you can jump to Chapter 7 *Tutorials* and complete all of these. It may be the case you occasionally need to go back to earlier chapters as reference. Also useful is complete all the *Beginner programming Exercises*. Pay particular attention to building your knowledge and to the exercise Line following.

2. The sample guwhiteboardtypelist.tsl file which is the input to the executable of the package guwhiteboardtypegenerator.

Place the package guwhiteboardtypegenerator in your catkin workspace. Assuming your catkin workspace is

```
$HOME/catkin_ws/src
```

You place guwhiteboardtypegenerator so that

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

results in at least the following packages

```
CMakeLists.txt
gusimplewhiteboard
libclfsm
wbperf
clfsm
guwhiteboardtypegenerator
r2d2mipal
```

Also, place the example guwhiteboardtypelist.tsl in

```
$HOME/catkin_ws/src/gusimplewhiteboard/include/
```

Make sure you can build the package guwhiteboardtypegenerator. You do it as is customary with ROS. Assuming $HOME/catkin_ws is your workspace.

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

Look at the compilation messages and your executable is built in

```
cd $HOME/catkin_ws/devel/lib/guwhiteboardtypegenerator/guwhiteboardtypegenerator
```

A simple message for a differential robot

We use the standard ROS approach.

```
cd $HOME/catkin_ws/src
catkin_create_pkg webots_epuck_nxt_differential_robot roscpp
```

Remember, in ROS, packages prefer names with lower case. Also, you may need to edit the file
webots_epuck_nxt_differential_robot/package.xml
for at least placing an e-mail address.

Now, we proceed with the *MiPal* approach to create class-oriented C++11 messages. The first thing is to create a plain C data structure for consistent cross platform alignment in memory. In *MiPal*, this is a file where the practice is use all lower case letter, and starting with wb_. Also, the name of the struct that would be used from there on is the same as the name of the file. Thus, since the struct is
wb_webots_epuck_NXT_differential_robot,
the name of the file is
wb_webots_epuck_NXT_differential_robot.h
We also place it in the include section of our package.

```
cd $HOME/catkin_ws/src/webots_epuck_nxt_differential_robot/include/webots_epuck_nxt_differential_robot
vi wb_webots_epuck_NXT_differential_robot.h
```

We will provide you this file, but is important if you actually input the lines yourself into an editor.

Note the use of the *MiPal* macro `PROPERTY` which will generate appropriate setters and getters for this attribute of the objects of your class. Also note that we can create constructors for `C++11` although not for the `C`-language. Note also that we use types for integers whose layout as bits is unambiguous; that is, do not use plain `int`.

`wb_webots_epuck_NXT_differential_robot.h`:

```
#ifndef _WWEBOTSEPUCKDIFFERENTIALNXT_

#define _WWEBOTSEPUCKDIFFERENTIALNXT_


#include <sys/types.h>

#include <gu_util.h>


        /** Enumeration of the Sensor ports available in the NXT differential robot
     *         */
        enum NXT_Sensor_Ports {
                        NXT_PORT_1 = 0,
                        NXT_PORT_2 = 1,
                        NXT_PORT_3 = 2,
                        NXT_PORT_4 = 3
                };


        /** Enumeration of the Colour Channels available in camera of the E-Puck differential robot
     *         */
        enum CAMERA_E_PUCK_CHANNELS {
                        BLUE_CHANNEL = NXT_PORT_1,
                        RED_CHANNEL = NXT_PORT_2,
                        GREEN_CHANNEL = NXT_PORT_3,
                        GREY_CHANNEL = NXT_PORT_4,
                                CHANNEL_SENTINEL=4
                };


/* The data structure for the values     of a webotsEPuckNXTdifferentialRobot OO-message
 *
 */


struct wb_motor
{
    /********** The motor Power : a percentage from [-100,100], signed, negative is backwards */
    PROPERTY(int16_t, motor_power)
#ifdef __cplusplus
        /** default constructor */
        wb_motor() :  _motor_power(0) {}
        /** copy constructor */
        wb_motor(const wb_motor &other) :  _motor_power(other._motor_power) {}
```

```
        /** assignment operator */
    wb_motor &operator=(const  wb_motor &other) {_motor_power=other._motor_power;  return *this;}
#endif
};

struct wb_button
{
    /* true = pressed, false, no contact */
    PROPERTY(bool,pressed)
    /* true = post regularly to the whiteboard, false, no reporting, sensor is off */
    PROPERTY(bool,on)
#ifdef __cplusplus
        /** default constructor */
        wb_button() : _pressed(false), _on(false)  {}
        /** copy constructor */
        wb_button(const wb_button &other) :  _pressed(other._pressed), _on(other._on) {}
        /** assignment operator */
    wb_button &operator=(const  wb_button &other) {_pressed=other._pressed; _on=other._on;  return *this;}
#endif
};

struct wb_distance
{
    /* NXT is mm, Webots ePuck not know TODO */
    PROPERTY(int16_t,distance)
    /* true = post regularly to the whiteboard, false, no reporting, sensor is off */
    PROPERTY(bool,on)
#ifdef __cplusplus
        /** default constructor */
        wb_distance() :  _distance(0), _on(false)  {}
        /** copy constructor */
        wb_distance(const wb_distance &other) :  _distance(other._distance), _on(other._on) {}
        /** assignment operator */
    wb_distance &operator=(const  wb_distance &other) {_distance=other._distance; _on=other._on;  return *this;}
#endif
};

struct wb_encoder
{
    /* Turns of a motor */
    PROPERTY(int16_t,rotations)
    /* true = post regularly to the whiteboard, false, no reporting, sensor is off */
    PROPERTY(bool,on)
#ifdef __cplusplus
        /** default constructor */
```

```
        wb_encoder() :  _rotations(0), _on(false) {}
        /** copy constructor */
        wb_encoder(const wb_encoder &other) :  _rotations(other._rotations), _on(other._on) {}
        /** assignment operator */
    wb_encoder &operator=(const  wb_encoder &other) {_rotations=other._rotations; _on=other._on;  return *this;}

#endif
};

struct wb_camera
{
    /* camera width */
    PROPERTY(u_int8_t,camera_width)
    /* camera height */
    PROPERTY(u_int8_t,camera_height)
    /* camera camera_threshold */
    PROPERTY(u_int8_t,camera_threshold)
    /* camera median per channel */
    ARRAY_PROPERTY(u_int8_t, median, (GREY_CHANNEL+1))
    /* camera total per channel */
    ARRAY_PROPERTY(u_int8_t, totalpixel_count, (GREY_CHANNEL+1))
    /* true = post regularly to the whiteboard, false, no reporting, sensor is off */
    PROPERTY(bool,on)
#ifdef __cplusplus
        /** default constructor */
        wb_camera() :  _camera_width(0), _camera_height(0), _camera_threshold(0), _on(false) {}
        /** copy constructor */
        wb_camera(const wb_camera &other) :     _camera_width(other._camera_width),
                        _camera_height(other._camera_height),
                    _camera_threshold(other._camera_threshold),
                        _on(other._on) {
                        for (CAMERA_E_PUCK_CHANNELS channel_i= BLUE_CHANNEL;
                                channel_i != GREY_CHANNEL ;
                                channel_i = static_cast<CAMERA_E_PUCK_CHANNELS>(static_cast<int>(channel_i) + 1))
                                { _median[channel_i]=other._median[channel_i];
                                  _totalpixel_count[channel_i]=other._totalpixel_count[channel_i];
                                }
                    }
        /** assignment operator */
    wb_camera &operator=(const  wb_camera &other) { _camera_width=other._camera_width;
                            _camera_height=other._camera_height;
                        _camera_threshold=other._camera_threshold;
                        for (CAMERA_E_PUCK_CHANNELS channel_i= BLUE_CHANNEL;
                                channel_i != GREY_CHANNEL ;
                                channel_i = static_cast<CAMERA_E_PUCK_CHANNELS>(static_cast<int>(channel_i) + 1))
```

```
                                        { _median[channel_i]=other._median[channel_i];
                                          _totalpixel_count[channel_i]=other._totalpixel_count[channel_i];
                                        }
                                                    _on=other._on;   return *this;}
#endif
};



struct wb_webots_epuck_NXT_differential_robot
{
        PROPERTY(int16_t ,robot_ID)
    /* Differential robot has two motors with encoders */
        PROPERTY(struct wb_motor, left_motor)
    PROPERTY(struct wb_motor, right_motor)
        PROPERTY(struct wb_encoder, left_encoder)
    PROPERTY(struct wb_encoder, right_encoder)
    /* for the ePuck */
        PROPERTY(int16_t ,max_times_radians_speed)
    /* Also left and right touch sensors */
        PROPERTY(struct wb_button, left_button)
    PROPERTY(struct wb_button, right_button)
        /* For now one sonar sensor*/
    PROPERTY(struct wb_distance, sonar_sensor)
        /* For now one camera*/
    PROPERTY(struct wb_camera, the_camera)
#ifdef __cplusplus
        /** copy constructor */
    wb_webots_epuck_NXT_differential_robot() :
        _robot_ID(),
            _left_motor(), _right_motor(),  _left_encoder(), _right_encoder(), _max_times_radians_speed(),  _left_button(), _right_button(),
_sonar_sensor(), _the_camera() {}
        /** copy constructor */
        wb_webots_epuck_NXT_differential_robot(const wb_webots_epuck_NXT_differential_robot &other) :
        _robot_ID(other._robot_ID),
        _left_motor(other._left_motor), _right_motor(other._right_motor),
        _left_encoder(other._left_encoder), _right_encoder(other._right_encoder),
        _max_times_radians_speed(other._max_times_radians_speed),
        _left_button(other._left_button), _right_button(other._right_button),
        _sonar_sensor(other._sonar_sensor),
        _the_camera(other._the_camera)
        {}
        /** assignment operator */
    wb_webots_epuck_NXT_differential_robot &operator=(const  wb_webots_epuck_NXT_differential_robot &other) {
        _robot_ID=other._robot_ID;
        _left_motor=other._left_motor; _right_motor=other._right_motor;
```

```
        _left_encoder=other._left_encoder; _right_encoder=other._right_encoder;
        _max_times_radians_speed=other._max_times_radians_speed;
        _left_button=other._left_button; _right_button=other._right_button;
        _sonar_sensor=other._sonar_sensor;
        _the_camera=other._the_camera;
            return *this;}
#endif
};


#endif //_WWEBOTSEPUCKDIFFERENTIALNXT_
```

Once we have the C-language structure we use it for a C++11-class. There are several important aspect of the practice and layout of these classes so they can be used as gusimplewhiteboard messages.

Again, in the include directory of our ROS package we create the C++11-class.

```
cd $HOME/catkin_ws/src/webots_epuck_nxt_differential_robot/include/webots_epuck_nxt_differential_robot
vi WebotsEPuckNXTdifferentialRobotControlStatus.h
```

The name of the file is the same as the name of the class, and the convention here for *MiPal* is that C++11 classes are upper-case. We have ControlStatus added, because one class can be used as both, a *status* message or a *control* message, Typically, a status message is posted periodically by the driver of the robot, and reflects that status of the device/robot. The control message is typically posted by a controller, who wishes to change the status of the robot to the content of the control message.

There are several important aspects of the C++11-message.

1. First, they are part of the guWhiteboard name space. **However, it is best practice if include files, do not use namespaces**.

2. They are built by sub-classing the C-language data structure.

   ```
   class WebotsEPuckNXTdifferentialRobotControlStatus:  public wb_webots_epuck_NXT_differential_robot
   ```

3. Because of the above point, they must include the file detailing the C-language data structure.

4. In order to be properly integrated among the gusimplewhiteboard messages they must be wrapped-up in a #ifndef that has the form

   ```
   <class_name>_DEFINED
   ```

   In our case, this will be

   ```
   #ifndef WebotsEPuckNXTdifferentialRobotControlStatus_DEFINED
   #define WebotsEPuckNXTdifferentialRobotControlStatus_DEFINED
   ...
   #endif // WebotsEPuckNXTdifferentialRobotControlStatus_DEFINE
   ```

We will discuss more details later, for now, again, we provide the entire file. However, it is better if you type this file in an editor. You will learn from errors if you cannot compile it. WebotsEPuckNXTdifferentialRobotControlStatus.h:

```
/***************** IMPORTANT    */
/* is <class_name>_DEFINED                  */
```

```
/************************************/

#ifndef WebotsEPuckNXTdifferentialRobotControlStatus_DEFINED
#define WebotsEPuckNXTdifferentialRobotControlStatus_DEFINED

#include <cstdlib>
#include <sstream>
#include <gu_util.h>

/******************IMORTANT     */
/* The corresponding C structure        */
/************************************/
#include "webots_epuck_nxt_differential_robot/wb_webots_epuck_NXT_differential_robot.h"

namespace guWhiteboard {

    class WebotsEPuckNXTdifferentialRobotControlStatus:   public wb_webots_epuck_NXT_differential_robot
    {
    public:
        /** designated constructor **/
    WebotsEPuckNXTdifferentialRobotControlStatus(): wb_webots_epuck_NXT_differential_robot() {}

    /** copy constructor **/
    /*
      WebotsEPuckNXTdifferentialRobotControlStatus(const WebotsEPuckNXTdifferentialRobotControlStatus &other):
            _left_motor._motor_power(other._left_motor._motor_power)
              {}
        */

    /** copy assignment operator **/

    /** copy assignment operator **/
     /**    INTERESTING !!!! */

#ifdef WHITEBOARD_POSTER_STRING_CONVERSION

#define SEPARATOR_IS_COMMA ','
#define SEPARATOR_IS_COLON ':'
#define IS_VISIBLE_ID 'I'

    std::string description() {
        std::ostringstream ss;
        ss << "a test of description";
        return ss.str();
    }
```

```
#endif // WHITEBOARD_POSTER_STRING_CONVERSION

    }; // class

} //namespace

#endif // WebotsEPuckNXTdifferentialRobotControlStatus_DEFINED
```

Now that you have a class-oriented-message for *MiPal*, it must be incorporated into the known messages of the `gusimplewhiteboard`. Here is where the file
`guwhiteboardtypelist.tsl`
and the package
`guwhiteboardtypegenerator`
play a role.

We first need to edit `guwhiteboardtypelist.tsl`, which should be in

```
$HOME/catkin_ws/src/gusimplewhiteboard/include/
```

We need to add two lines at the end that incorporate our new status message and control message. `P3.d/additional_guwhiteboardtypelist.tsl`:

```
class:WebotsEPuckNXTdifferentialRobotControlStatus, nonatomic, WebotsEPuckNXTdifferentialRobotStatus,                  "WebotsEPuckNXTdifferentialRobo
class:WebotsEPuckNXTdifferentialRobotControlStatus, nonatomic, WebotsEPuckNXTdifferentialRobotControl,                 "WebotsEPuckNXTdifferentialRobo
```

Thus, append these to lines to the file `guwhiteboardtypelist.tsl`. We recommend that you see the date and time of last modification of files.

```
ls -la
```

Files like
`WBFunctor_types_generated.h`,
`guwhiteboardtypelist_c_typestrings_generated.c`,
`guwhiteboardgetter.cpp`,
`guwhiteboardtypelist_generated.h`,
`guwhiteboardposter.cpp`,
`guwhiteboardtypelist_tcp_generated.h`, and
`guwhiteboardtypelist_c_generated.h`
are going to be replaced.

Once you have these two lines, pale yourself inside the directory `typeClassDefs`. If you are already in

```
$HOME/catkin_ws/src/gusimplewhiteboard/include/
```

it is simply

```
cd typeClassDefs
```

but the absolute path is

```
cd $HOME/catkin_ws/src/gusimplewhiteboard/include/typeClassDefs
```

We will execute the program from the package `guwhiteboardtypegenerator`.

```
../../../../devel/lib/guwhiteboardtypegenerator/guwhiteboardtypegenerator
```

Alternatively you can provide the absolute path

```
cd $HOME/catkin_ws/devel/lib/guwhiteboardtypegenerator/guwhiteboardtypegenerator
```

You should see that the two lines you added are posted as p=being processed.

Now, however, you need to build a new `gusimplewhiteboard` and it library. The easiest way is to recompile all packages. So remove everything that is being constructed.

```
cd $HOME/catkin_ws
rm -fr build/ devel/
catkin_make
```

A simple publisher

We are now in a position to build a program that posts to the whiteboard messages of our newly created class. This will be our first executable of the new package. It will be the publisher.

```
cd $HOME/catkin_ws/src/webots_epuck_nxt_differential_robot/src
vi webots_epuck_nxt_differential_robotPublisher.cpp
```

The program is rather simple. After a small banner, it performs 3 things.

1. Creates an object of the recently defined class of messages.

   ```
   WebotsEPuckNXTdifferentialRobotControlStatus a_robot
   ```

2. Crete a `gusimplewhiteboard` *control* handler.

   ```
   WebotsEPuckNXTdifferentialRobotControl_t control_Handler;
   ```

3. Place the message in the communication middleware:

   ```
   control_Handler.set(a_robot);
   ```

The complete code is the following.
`P3.d/webots_epuck_nxt_differential_robotPublisher.cpp`:

```
#include <iostream>

#include "webots_epuck_nxt_differential_robot/WebotsEPuckNXTdifferentialRobotControlStatus.h"
#include "gugenericwhiteboardobject.h"
#include "guwhiteboardwatcher.h"

int main(void) {

    using namespace  std;
    using namespace  guWhiteboard;

    cout << "This is an example of a publishing using the Class-oriented whiteboard "  << endl;

    WebotsEPuckNXTdifferentialRobotControlStatus a_robot;

    WebotsEPuckNXTdifferentialRobotControl_t control_Handler;
```

```
    control_Handler.set(a_robot);

}
```

To compile this program we need to configure the
`CMakeLists.txt`
of the package
`/webots_epuck_nxt_differential_robot`.
Thus we need to edit
`$HOME/catkin_ws/src/webots_epuck_nxt_differential_robot/CMakeLists.txt`.

The changes are simple. We recommend that you search for the initial string and some of the changes are produced by editing. Most changes are in the `Build` section.

(search for "`find_package(catkin`") Correct that it only needs the `gusimplewhiteboard`. Thus replace

```
find_package(catkin REQUIRED COMPONENTS
roscpp
)
```

by

```
find_package(catkin REQUIRED COMPONENTS gusimplewhiteboard)
```

(search for "`catkin_package(`") Those dependencies are to be reflected as well in the `catkin_package()` section. Non commented lines should look like this.

```
catkin_package(
INCLUDE_DIRS include
# LIBRARIES webots_epuck_nxt_differential_robot
# CATKIN_DEPENDS gusimplewhiteboard
# DEPENDS system_lib
)
```

You can achieve this by a bit of un-documenting and a bit of editing.

(search for "`Build`") We strongly recommend that in the build section you place some important flags.

```
set(CMAKE_C_FLAGS "-std=c99")
set(CMAKE_CXX_FLAGS "-std=c++11 -DWHITEBOARD_POSTER_STRING_CONVERSION=1")
set(CMAKE_SHARED_LINKER_FLAGS "-Wl,-rpath -Wl,${CMAKE_INSTALL_PREFIX}/lib")
```

(search for "`include_directories(`") Define the actual paths to include directories.

```
include_directories(include)
include_directories(${gusimplewhiteboard_INCLUDE_DIRS})
```

(search for "`add_executable`") We actually place our executable.

```
add_executable(webots_epuck_nxt_differential_robotPublisher
src/webots_epuck_nxt_differential_robotPublisher.cpp)
```

(search for "`target_link_libraries(`") We need to specify that in order to build our executable, it needs to link against the `gusimplewhiteboard` library.

```
target_link_libraries(webots_epuck_nxt_differential_robotPublisher
gusimplewhiteboard
${CMAKE_DL_LIBS}
${LIBDISPATCH_LIBRARIES}
${CMAKE_THREAD_LIBS_INIT})
```

If all this editing is correct you actually can build and execute the publisher.

```
cd $HOME/catkin_ws
scriptsize catkin_make
./devel/lib/webots_epuck_nxt_differential_robot/webots_epuck_nxt_differential_robotPublisher
```

And the output will be something like the output of the banner.

```
This is an example of a publishing using the Class-oriented whiteboard
```

A simple subscriber: Strongly Discouraged

Of course things are more interesting with a subscriber. We prefer idempotent messages and minimization of threads. We can provide several arguments why rather than use an event-driven approach we prefer the use of `get_Message`. In summary, subscribers are strongly discouraged. But in a sense, we present it here for completeness.

An elegant form of the subscriber is to have an `_interface` class register the call-back in its constructor. Thus, in the `include` directory of our new package, we place the header of such a class.

```
cd $HOME/catkin_ws/src/webots_epuck_nxt_differential_robot/include/webots_epuck_nxt_differential_robot
vi webots_epuck_nxt_differential_robotSubscriber_interface.h
```

This is also a simple interface. It has three methods and one private member, which is the `watcher`. The `watcher` is declared as follows.

```
whiteboard_watcher *watcher;
```

The signatures are for three public functions are as follows.

1. A constructor

   ```
   WebotsEPuckNXTdifferentialRobotSubscriber_interface();
   ```

2. The banner to invoke after the constructor has provided us an object of this class.

   ```
   std::string banner()
   ```

   Note we do not use namespaces in header files.

3. The actual call-back that will be started in a different thread when a message of this type is received din the middleware.

   ```
   void callback(guWhiteboard::WBTypes t, guWhiteboard::WebotsEPuckNXTdifferentialRobotControlStatus
   &aControlStatus);
   ```

`webots_epuck_nxt_differential_robotSubscriber_interface.h`:

```
#include <iostream>


#include "webots_epuck_nxt_differential_robot/WebotsEPuckNXTdifferentialRobotControlStatus.h"


#include "guwhiteboardwatcher.h"


class WebotsEPuckNXTdifferentialRobotSubscriber_interface
```

```
{
        whiteboard_watcher *watcher;

    public:
            ///< constructor
    WebotsEPuckNXTdifferentialRobotSubscriber_interface();

        std::string banner() { return std::string("(c) Vlad Estivill_Castro, demo subsriber class-oriented whiteboard");}
        /// callback method

    void callback(guWhiteboard::WBTypes t, guWhiteboard::WebotsEPuckNXTdifferentialRobotControlStatus  &aControlStatus);
};
```

With this interface, its actual methods are in the `src` directory.

```
cd $HOME/catkin_ws/src/webots_epuck_nxt_differential_robot/src
vi webots_epuck_nxt_differential_robotSubscriber_interface.cpp
```

`webots_epuck_nxt_differential_robotSubscriber_interface.cpp`:

```
#include "webots_epuck_nxt_differential_robot/webots_epuck_nxt_differential_robotSubscriber_interface.h"

using namespace std;
using namespace guWhiteboard;

WebotsEPuckNXTdifferentialRobotSubscriber_interface::WebotsEPuckNXTdifferentialRobotSubscriber_interface()
{
    watcher = new whiteboard_watcher();

    // Note we are demonstrating the same call abck for two messages types

    SUBSCRIBE(watcher, WebotsEPuckNXTdifferentialRobotControl,
                WebotsEPuckNXTdifferentialRobotSubscriber_interface,
            WebotsEPuckNXTdifferentialRobotSubscriber_interface::callback);

}

void WebotsEPuckNXTdifferentialRobotSubscriber_interface::callback(guWhiteboard::WBTypes,
            guWhiteboard::WebotsEPuckNXTdifferentialRobotControlStatus  &aControlStatus)
{
    cout << " We received a control message " << aControlStatus.description() <<  std::endl;
}
```

The most important aspect is that the constructor initializes the watcher and subscribes the callback to the *control* message. The callback t this stage is very simple, it just prints a banner indicated a message has been received. Note that the callback runs in a different thread.

Thus, we are ready to construct the subscriber

```
cd $HOME/catkin_ws/src/webots_epuck_nxt_differential_robot/src
vi webots_epuck_nxt_differential_robotSubscriber.cpp
```

webots_epuck_nxt_differential_robotSubscriber.cpp:

```
#include "webots_epuck_nxt_differential_robot/webots_epuck_nxt_differential_robotSubscriber_interface.h"

using namespace std;
using namespace guWhiteboard;

int main(void) {

        cout << "This is an example of a simple subcriber using the Class-oriented whiteboard " <<  endl;
        WebotsEPuckNXTdifferentialRobotSubscriber_interface  *subscriber = new WebotsEPuckNXTdifferentialRobotSubscriber_interface();
        cout << subscriber->banner() << endl;

            while (1) { sleep(1); }

}
```

The subscriber just creates an object of the interface class, calls the banner of such object and then spins for ever in a sleep so it consumes minimal CPU cycles.

Thus, we now have to add this subscriber to the `CMakeLists.txt` of the package. In the same `CMakeLists.txt` we just need now even fewer changes.

(search for "`add_executable`") We actually place or executable. Note we have one target and two dependencies.

```
add_executable(webots_epuck_nxt_differential_robotSubscriber
src/webots_epuck_nxt_differential_robotSubscriber_interface.cpp
src/webots_epuck_nxt_differential_robotSubscriber.cpp)
```

(search for "`target_link_libraries(`") We also need the corresponding description at link time. In order to build our executable for the subscriber, it needs to link against the `gusimplewhiteboard` library.

```
target_link_libraries(webots_epuck_nxt_differential_robotSubscriber
gusimplewhiteboard
${CMAKE_DL_LIBS}
${LIBDISPATCH_LIBRARIES}
${CMAKE_THREAD_LIBS_INIT})
```

Controlling a robot

Here, we will demonstrate the preferred mode of receiving a control message. We use `get_Message` in the above infrastructure rather than a subscriber. The problem derives from all the concurrency issues that arise from the multiple threads that our module would need to handle for each invocation of the callback. Specially if we are to adjust our status for the corresponding control message.

The infrastructure described above which communicates modules via class-oriented messages is sufficient to create a driver for a robot. The driver receives *control* messages. A *control* messages is to be understood as a request to change the status of the robot. The driver also regularly publishes a *status* message and in doing so refreshes the belief of its status.

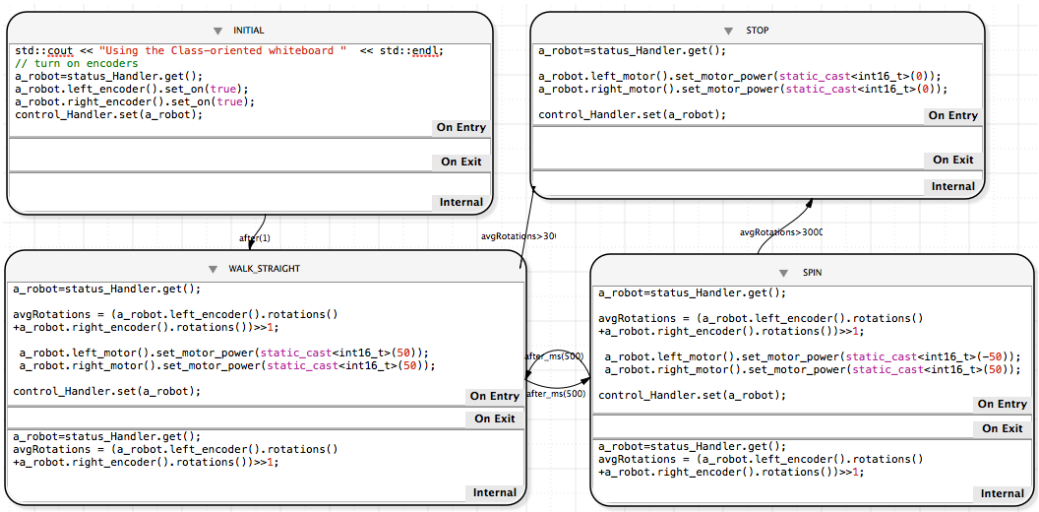

Figure 1: The `ePuckBehavior.machine` is a simple *llfsm*. While the encoders do not reach a high value it alternates between spinning and walking straight.

Thus, the driver-module is an interface but *w*ill not be coded as a subscriber. Any *llfsm* that is controlling the robot interacts with the driver-module as follows. It uses the status message on `gusimplewhiteboard` to learn about the status, and thus, sensor values in the robot. It uses a simple `get_Message` to obtains sensor values in a status message. On the other hand, when the *llfsm* publishes a control message, it commands the robot to some action. Since we are using essentially the same class for the status-message and the control-message, when an *llfsm* is to issue a control message, the standard approach is to first use `get_Message` to obtain the status object, change only a few things, like a few values, and re-send it as the control message.

The machine we will use in this demo is for illustration only can be built with MIEDITLLFSM (see Figure 1). This behavior is an alternation of the state WALK_STRAIGHT and the state SPIN until the average rotation of the motor encoders is above 3,000. When that happens, the *llfsm* goes into the state STOP. The state INITIAL, in the **OnEntry** section, besides printing a message to standard output, issues a control message to turn the motor encoders on.

```
std::cout « "Using the Class-oriented whiteboard " « std::endl;
// turn on encoders
a_robot=status_Handler.get();
a_robot.left_encoder().set_on(true);
a_robot.right_encoder().set_on(true);
control_Handler.set(a_robot);
```

Here, the `get_Message` that obtains the status of the robot is `a_robot=status_Handler.get()`. Note already the advantage of using object-oriented messages. Among the message-properties that message of class robot have are encoders; which themselves have the property of whether they are on or not. Using OO-notation, we set such properties on

```
a_robot.left_encoder().set_on(true);
a_robot.right_encoder().set_on(true);
```

. The last thing to do in the INITIAL is post the control message.

```
control_Handler.set(a_robot);
```

Lets now look at the STOP state. It only has an **OnEntry** section.

```
a_robot=status_Handler.get();
a_robot.left_motor().set_motor_power(static_cast<int16_t>(0));
a_robot.right_motor().set_motor_power(static_cast<int16_t>(0));
control_Handler.set(a_robot)
```

Again, the first line of code uses a `get_Message` to obtain the status of the robot.

```
a_robot=status_Handler.get();
```

Then, it uses object-orientation, as the getter `a_robot.left_motor()` is the left motor, while the getter `a_robot.right_motor()` is the right motor. To each, we access the property `motor_power` and we use a setter, to change the value of both motors to zero. It may look a bit strange to use `static_cast<int16_t>(0)` instead of plain 0, but recall that our structures have specific length in bits. The last step is to send the control message.

```
control_Handler.set(a_robot)
```

The **OnEntry** sections of the state WALK_STRAIGHT and the state SPIN are analogous to the **OnEntry** section fo the STOP state. At least with respect to modifying the values of the motor, they all use a `get_Message` to obtain the status, they use object-orientation o get the motors out of the robot-message and the setters to put appropriate values to the motor power. And finally, they post the control message. The only difference is that, in the **OnEntry** section and in the **Internal** the `get_Message` obtain the most recent values of the encoders and an average value is computer. For example, the **Internal** section of the WALK_STRAIGHT and the state SPIN is

```
a_robot=status_Handler.get();
avgRotations = (a_robot.left_encoder().rotations()
+a_robot.right_encoder().rotations())»1;
```

It is important to refresh the value of the sensor in the **Internal** section that is executed every time the transitions out fail. Both states have the same transitions out ot STOP: `avgRotations>3000` and a transition to each other `after_ms(500)`.

This machine needs thus the variables to access the messages through `gusimplewhiteboard`.

```
WebotsEPuckNXTdifferentialRobotControl_t control_Handler;
WebotsEPuckNXTdifferentialRobotStatus_t status_Handler;
```

A variable for the OO-message

```
WebotsEPuckNXTdifferentialRobotControlStatus a_robot;
```

and finally a variable for the average distance measured in both encoders

```
int16_t avgRotations;
```

These 4 variables should be placed in the variables section fo the machine. The `include` section fo the machine is as follows.

```
#include <iostream>
#include "CLMacros.h"
#include "webots_epuck_nxt_differential_robot/WebotsEPuckNXTdifferentialRobotControlStatus.h"
#include "gugenericwhiteboardobject.h"
#include "guwhiteboardwatcher.h"
using namespace guWhiteboard;
```

This is essentially all that is required to construct a behavior that uses sensors and actuators with `Webots` and the *MiPal* `gusimplewhiteboard` as the middleware. However, there are a few things that need to be set up to actually run it.

The driver for Webots

We will demonstrate the construction of a driver for a robot constructing a *controller* for the Webots ePuck. The Webots-controller typically has a constructor that is executed when we revert the world, then it has a run() method executed in an endless loop when the simulation is playing. The loop is paused if the simulation is paused.

Our controller thus will use an interface class GUWebotsDriver, and then execute the run() method. We also place this as part of the current catkin package.

```
cd $HOME/catkin_ws/src/webots_epuck_nxt_differential_robot/src
vi guwebotsdrivermodule.cpp
```

guwebotsdrivermodule.cpp:

```
#include "webots_epuck_nxt_differential_robot/GUWebotsDriver.h"


/*
 *   Main program that should be placed as a controller. It should accept an argument
 *   (parameter) which would be the robot number, so it ignores commands to other
 * robot-IDs, and publishes sensor data with the given Robot-ID.
 *
 * Uses and object of the class GUWebotsDriver to initialize subscriptions, and run
 *
 */


static __attribute__((__noreturn__)) void usage(const char *cmd)
{   std::cerr << "Usage: " << cmd << " [-p  robotNumber ] " << std::endl;
    exit(EXIT_FAILURE);
}


int main(int argc, char *argv[])
{
    int aRobotID=0;


    int ch;


    while ((ch = getopt(argc, argv, "p:")) != -1) {
        switch (ch) {
        case 'p':         // robot number
            std::cout << optarg << std::endl;
            aRobotID = atoi(optarg);
            break;
        case '?':
        default: usage(argv[0]);
        }
    }


    GUWebotsDriver* controller = new GUWebotsDriver(aRobotID);
    controller->run();
```

```
    delete controller;
    return 0;
}
```

We observe from this code that the class `GUWebotsDriver` has basically 3 methods.

1. A constructor `GUWebotsDriver(int8_t theRobotID);` that receives a robot identification number. The idea is we can run `Webots` with several ePuck in the same environment (nut possibly different behavior). We have done this two study collaborative planning and formations.
2. A banner `std::string banner()` to display output once indicating execution.
3. The `run` method that contains the `Webots` simulation loop.

These methods are defined in a header file placed in the `include` directory of the package.

```
cd $HOME/catkin_ws/src/webots_epuck_nxt_differential_robot/include/webots_epuck_nxt_differential_robot
vi GUWebotsDriver.h
```

`GUWebotsDriver.h`:

```
#ifndef GUWEBOTSINTERFACE
#define GUWEBOTSINTERFACE

#include <sstream>

#include <webots/Robot.hpp>
#include <webots/DifferentialWheels.hpp>
#include <webots/Camera.hpp>

/**
 * the definition of the status/control messages for differential robot
 */
#include "webots_epuck_nxt_differential_robot/WebotsEPuckNXTdifferentialRobotControlStatus.h"

// we will use the class oriented whiteboard
#include "guwhiteboardwatcher.h"

const float TOP_MOTOR_POWER =100.0;
/**
 *   Webots constants
 */
const int  TIME_STEP= 32; // [ms] // time step of the simulation
const int WHEEL_SAMPLING_PERIOD= 30;
const int  WEBOTS_PIXELS_MAX_VALUE=255;
const int  TIME_STEP_CAM=64;

/**
```

```
*   Class that acts as a webots controller (a driver).
*   Webots uses the term controller, we use the term driver.
*   We will use it mainly for the e-puck
*   It does inherit from a robot of type     webots::DifferentialWheels
*/
class GUWebotsDriver : public webots::DifferentialWheels {

public:
    /**
     * GUWebotsDriver constructor
     */
    GUWebotsDriver(int8_t theRobotID);

    /**
     * GUWebotsDriver destructor
     */
    virtual ~GUWebotsDriver();

    /**
     * infinite loop that reads sensor values, report them if necessary and sends commands.
     * General architecture is to read all sensors with the WEBOTS API
            * Check which ones to report, and build the whiteboard message needed to post
     */
    void run();

    /**
    * announce we are running
    */
    std::string banner() { return std::string("(c) Vlad Estivill_Castro, webots driver/controller differential robot e-Puck");}

    /*!
     * Better to use a method to analyze control message with a get-Message in out time-triggered approach
     * Will update out status for turning sensor on off, or signals to actuators
     */
    virtual void analyzeControl ();

private :
    bool isRunning;

    /* to keep my status */

    guWhiteboard :: WebotsEPuckNXTdifferentialRobotControlStatus a_robot_status;
    guWhiteboard :: WebotsEPuckNXTdifferentialRobotStatus_t status_Handler;

    /*
```

```
     * Always take the parameter as a value in the range [-100,100]
     */
    int16_t bounddParameter(int16_t theParamter) {
         int16_t sign = theParamter >= 0.0 ? 1 : -1;
     int16_t bounded = fabs( theParamter ) > TOP_MOTOR_POWER*1.0 ?  sign * TOP_MOTOR_POWER : theParamter;
     return bounded;
    }


    /**
     * Webots parameters
     */
    int speedLeft, speedRight;
    int wheelEncoderPeriod;
     /**
       *        * The speed of the e-Puck Webot
       */
    //int timesRadiansSpeed;



        /**
     *         The camera of the e-Puck Webot
          */
        webots::Camera * myCamera;
        // buffers    to read the pixels
        int *imagePerChannel[CHANNEL_SENTINEL];
    const unsigned char *the_image;



    void read_image( )
    {
        the_image= myCamera->getImage();
        for (int i = 0; i < a_robot_status.the_camera().camera_width(); i++)
             {    // imageGetBlue selects yellow on the ground very well
                  // imageGetGreen selects magenta very well
                  // imageGetRed selects blue/cyan very well
                    imagePerChannel[BLUE_CHANNEL][i] = WEBOTS_PIXELS_MAX_VALUE-myCamera->imageGetBlue(the_image, a_robot_status.the_camera().camera_width()
i, 0);
                    imagePerChannel[RED_CHANNEL][i] = WEBOTS_PIXELS_MAX_VALUE-myCamera->imageGetRed(the_image, a_robot_status.the_camera().camera_width(),
i, 0);
                    imagePerChannel[GREEN_CHANNEL][i] = WEBOTS_PIXELS_MAX_VALUE-myCamera->imageGetGreen(the_image, a_robot_status.the_camera().camera_width
i, 0);
                    imagePerChannel[GREY_CHANNEL][i] = WEBOTS_PIXELS_MAX_VALUE-myCamera->imageGetGrey(the_image, a_robot_status.the_camera().camera_width()
i, 0);
                }
    }
```

```
/**
 * Image analysis routine.    This function returns the amount of visible
 * pixels of the given channel, aims at producing a color detection
 * routine
 */
int16_t find_total(int tab[], int sizeTab, int threshold){
    int16_t count=0;
    for (int i=0; i<sizeTab;i++){
        if (tab[i]>threshold) count++;
    }
    return count;

}
/**
 * Image analysis routine.    This function returns the position
 *   This function returns the position of the peak contained in the array given
 * in argument (I have hesitation of what it actually does, Vlad)
 */
int16_t find_middle(int tab[], int sizeTab){

    int i,j;
    int *copy = (int *)malloc(sizeof(int)*sizeTab);
    int mean=0;
    int nb_best = sizeTab/10;
    int *index_bests = (int *)malloc(sizeof(int)*nb_best);

    // copy the tab, calculate the mean and
    // test if all the values are identical
    int identical=1;
    for (i=0; i<sizeTab;i++){
      copy[i]=tab[i];
      mean+=tab[i];
      if (tab[i]!=tab[0]) identical=0;
     }

    // exist with middle point if identical
     if (identical) {   free(copy);    free(index_bests); return (int16_t) (sizeTab/2);          }

  mean/=sizeTab;

   // take the best values of the tab
       for (i=0; i<nb_best; i++){
                    index_bests[i]=-1;
                    int index=-1;
```

```
                        int max=0;
                        for (j=0; j<sizeTab; j++){
                                if (max<copy[j] && copy[j]>mean){
                                        max=copy[j];
                                        index=j;
                                    }
                          }
                        index_bests[i]=index;
                        copy[index]=0;
                }
            free(copy);
    //
            // calculate the position mean of th best values
            int firstMean=0;
            int count=0;
            for (i=0; i<nb_best; i++){
              if(index_bests[i]!=-1) {
                    firstMean+=index_bests[i];
                          count++;
                    }
            }

    // count does not change send the middle
            if (count==0) { free(index_bests); return (int16_t) (sizeTab/2); }
                      firstMean/=count;

           // eliminate extreme values
           int secondMean=0;
           count=0;
           for (i=0; i<nb_best; i++){
                if (index_bests[i]<firstMean+sizeTab/10 && index_bests[i]>firstMean-sizeTab/10) {
                    count++;
                     secondMean+=index_bests[i];
                  }
            }
           free(index_bests);
           if (count==0) return (int16_t) (sizeTab/2);
      return (int16_t) (secondMean/count);
       }

};

#endif // GUWEBOTSINTERFACE
```

Thus, the real work is done in the constructor. Here, the corresponding initialization for a controller in `C++11` for `Webots` is performed (see the tutorials in the documentation for `Webots`).

The status of the robot is saved in an object of the class `WebotsEPuckNXTdifferentialRobotControlStatus`, it is convenient to make use of this class as well here, and will be reflected as atatus and control. In particular, during the constructor, the first status message is posted. We make an exception of posting a control message once, so that we do not read later in the `run()` a message left on the whiteboard by accident.

The other part where the work happens is in the `run` method. This is also the standard structure of the `Webots` controller for `C++11` (again, we assume some familiarity at the level of the initial tutorials for `Webots`). The source code of the class `GUWebotsDriver` is placed in the `src` directory of the package.

```
cd $HOME/catkin_ws/src/webots_epuck_nxt_differential_robot/src
vi GUWebotsDriver.cpp
```

The run method simply loops at the rate of the `Webots` simulator, and the first thing it does is is to extract the control-message with a `get_Message`. This happens in the function `analyzeControl()`. Here, signals that encoders, camera or motors should be turn on are copied to the internal status message. The rest of the while loop in the `run()` methods collects sensor data for those which our status tells us we are on. And if the is the case that a sensor is on, it reads the values and stores it in the status message. The status message is also posted at the end of every iteration of the simulation. In every iteration of the simulation, the speed of the motors is adjusted to the value in the internal status message.

`GUWebotsDriver.cpp`:

```
#include "gu_util.h"   // for DBG = DEBUG
#include <iostream>
#include <stdlib.h>

#include "webots_epuck_nxt_differential_robot/GUWebotsDriver.h"

// All the webots classes are defined in the "webots" namespace
using namespace webots;

using namespace std;
using namespace guWhiteboard;



// GUWebotsDriver constructor
GUWebotsDriver:: GUWebotsDriver(int8_t aRobotID): DifferentialWheels()  {

    a_robot_status.set_robot_ID( aRobotID );
    cerr << banner() <<  endl;
    cerr << "ROBOT ID: " << a_robot_status.robot_ID() << endl;

    isRunning=false;

    /********** start with no speed on the motor */
    speedLeft=0; speedRight=0;
    wb_motor my_left_motor; my_left_motor.set_motor_power(static_cast<int16_t>(0));
```

```
    wb_motor my_right_motor; my_right_motor.set_motor_power(static_cast<int16_t>(0));
    a_robot_status.set_left_motor(my_left_motor);
    a_robot_status.set_right_motor(my_right_motor);
    a_robot_status.set_right_motor(my_right_motor);
    int16_t theMaxSpeed=static_cast<int16_t>(        round (getMaxSpeed()/3.14159265) ) ;
    a_robot_status.set_max_times_radians_speed ( theMaxSpeed   );

    /********** Set up the camera */
     myCamera= getCamera ("camera");
     myCamera->enable(TIME_STEP_CAM);
    a_robot_status.the_camera().set_camera_width(myCamera->getWidth());
    a_robot_status.the_camera().set_camera_height(myCamera->getHeight());
    a_robot_status.the_camera().set_camera_threshold(0);
   for (CAMERA_E_PUCK_CHANNELS channel_i= CAMERA_E_PUCK_CHANNELS::BLUE_CHANNEL;
                          channel_i != CAMERA_E_PUCK_CHANNELS::CHANNEL_SENTINEL ;
                          channel_i = static_cast<CAMERA_E_PUCK_CHANNELS>(static_cast<int>(channel_i) + 1))
                          {
                imagePerChannel[channel_i]= (int *)malloc(sizeof(int)*a_robot_status.the_camera().camera_width());
            }
    a_robot_status.the_camera().set_on(false);

    /********** start with motor in zero and off */
    wheelEncoderPeriod = WHEEL_SAMPLING_PERIOD;
    enableEncoders(wheelEncoderPeriod);
    setEncoders(0.0,0.0);
    wb_encoder my_left_encoder; my_left_encoder.set_rotations(static_cast<int16_t>(0)); my_left_encoder.set_on(false);
    wb_encoder my_right_encoder; my_right_encoder.set_rotations(static_cast<int16_t>(0)); my_right_encoder.set_on(false);
    a_robot_status.set_left_encoder(my_left_encoder);
    a_robot_status.set_right_encoder(my_right_encoder);

    /************ clean the whiteboard with first status message ***********/
   status_Handler.set(a_robot_status);
    /************ clean the whiteboard with a command equal to the first status message ***********/
    WebotsEPuckNXTdifferentialRobotControl_t        control_Handler;
    WebotsEPuckNXTdifferentialRobotControlStatus    theCommand;
    theCommand=a_robot_status;
    control_Handler.set(theCommand);

    cerr <<  " CONSTRUCTOR FINISHED :" << endl;
}

// GUWebotsDriver destructor
GUWebotsDriver :: ~GUWebotsDriver() {
    // Enter here exit cleanup code
    //
```

```
    for (CAMERA_E_PUCK_CHANNELS channel_i= CAMERA_E_PUCK_CHANNELS::BLUE_CHANNEL;
                            channel_i != CAMERA_E_PUCK_CHANNELS::CHANNEL_SENTINEL ;
                            channel_i = static_cast<CAMERA_E_PUCK_CHANNELS>(static_cast<int>(channel_i) + 1))
                            { free ( imagePerChannel[channel_i] );
                }
}

// User defined function for initializing and running
// the GUWebotsDriver class
void GUWebotsDriver ::run() {

    // Main loop
    int result;
    do {
        //Better to eliminate subscribe and facilitate true idempotent messages
    //Check if there is any control
    analyzeControl ();
        // Code to handle the rotation sensors (encoders):
    //
    // Test if encoder is ON
    if ( a_robot_status.left_encoder().on() )
          a_robot_status.left_encoder().set_rotations( static_cast<int16_t>(getLeftEncoder()));

    if ( a_robot_status.right_encoder().on() )
          a_robot_status.right_encoder().set_rotations( static_cast<int16_t>(getRightEncoder()) );
        // READ the Camera
    read_image();
    if (a_robot_status.the_camera().on())
    { for (CAMERA_E_PUCK_CHANNELS channel_i= CAMERA_E_PUCK_CHANNELS::BLUE_CHANNEL;
                    channel_i != CAMERA_E_PUCK_CHANNELS::CHANNEL_SENTINEL ;
                    channel_i = static_cast<CAMERA_E_PUCK_CHANNELS>(static_cast<int>(channel_i) + 1))
         { a_robot_status.the_camera().set_totalpixel_count(
            find_total( imagePerChannel[channel_i],a_robot_status.the_camera().camera_width(),a_robot_status.the_camera().camera_threshold()
),
            channel_i
            );
                  a_robot_status.the_camera().set_median(
            find_middle( imagePerChannel[channel_i],a_robot_status.the_camera().camera_width() ),
             channel_i
            );
         }
    }

    // Post new status
         status_Handler.set(a_robot_status);
```

```
    //Set the speed of the motors
        setSpeed(speedLeft,speedRight);

        // Perform a simulation step of TIME_STEP milliseconds and leave the loop when the simulation is over
        result = step(TIME_STEP);
        usleep(10); //give wb message      a chance to enable/disable devices

    } while (result != -1);
}

void
GUWebotsDriver:: analyzeControl ()
{
    // Lets do a get-Message on our Control and update out internal status accordingly

    WebotsEPuckNXTdifferentialRobotControl_t        control_Handler;
    WebotsEPuckNXTdifferentialRobotControlStatus    theCommand;
    theCommand=control_Handler.get();

    DBG(cout << " We received a control message " << theCommand.description() <<  std::endl;)

    // we get motor as a power in [0,100] with a sign for direction
    speedLeft= (        bounddParameter( theCommand.left_motor().motor_power()   ) /100.0 ) *(getMaxSpeed()/getSpeedUnit()) ;
    speedRight= (       bounddParameter(theCommand.right_motor().motor_power()   ) /100.0)*(getMaxSpeed()/getSpeedUnit()) ;
  // Turn encoder on and off
    a_robot_status.left_encoder().set_on(   theCommand.left_encoder().on()  );
    a_robot_status.right_encoder().set_on(  theCommand.right_encoder().on() );
  // Turn the camera on and off
    a_robot_status.the_camera().set_on(  theCommand.the_camera().on() );
    /*
    if ( theCommand.the_camera().on() )
                                    cerr <<   " Turn the camera on" << a_robot_status.the_camera().on()   << endl;
    else
                                    cerr <<   " Turn the CAMERA OFF" << a_robot_status.the_camera().on()   << endl;
      */
}
```

Compiling the package, the *llfsm* and running the robot in `Webots`

Naturally, we need to update the file `CMakeLists.txt` of our package to incorporate the recent changes. We require to include the corresponding `Webots` definitions for a robot, and we need to link against the `Webots` libraries.

1. It depends where you installed `Webots`. The `Webots` installation usually set an environ-

ment variable. So, we need to pass this to the `catkin` descriptions. In the `CMakeLists.txt`, very early, include the line

```
set(WEBOTS_HOME $ENV{WEBOTS_HOME})
```

2. To provide the path for the `include` files of `Webots`, add the following line to the other definitions of paths for `include` files.

```
include_directories(${WEBOTS_HOME}/include/controller/cpp)
```

3. The driver is a new executable, although it will be ran by the `Webots` simulator, it will not run by itself.

```
add_executable(guwebotsdrivermodule
src/guwebotsdrivermodule.cpp
src/GUWebotsDriver.cpp)
```

4. We need to provide the path for linking the `Webots` libraries.

```
find_library(CPP_CONTROLLER_LIBRARY CppController ${WEBOTS_HOME}/lib)
```

5. We also need to indicate to link against such libraries.

```
target_link_libraries(guwebotsdrivermodule gusimplewhiteboard ${CMAKE_DL_LIBS}
${LIBDISPATCH_LIBRARIES} ${CMAKE_THREAD_LIBS_INIT} ${CPP_CONTROLLER_LIBRARY})
```

This changes complete the modification to the file `CMakeLists.txt`. You should check that all executables can be built using the `catkin_make` command from the directory of your workspace.

Now, we need to place the "executable" driver in the corresponding directory for the `Webots` simulator to run it. This typically involves opening a `Webots` world with an ePuck (like in the tutorials). In `Webots`, there is an option to click on the properties of the ePuck and to select a new controller. You must identify the path `CONTROLLERS` to the `controllers` directory of you `Webots` world. There you have to create a directory named `guwebotsdrivermodule` and copy the executable there.

```
cd $CONTROLLERS
mkdir guwebotsdrivermodule
cd guwebotsdrivermodule
cp $HOME/catkin_ws/devel/lib/webots_epuck_nxt_differential_robot/guwebotsdrivermodule .
```

Revert the wold in `Webots`, and you should see in the console of `Webots` the banner of the class `GUWebotsDriver`. Do not start the simulation yet, we will now describe how to run the `ePuckBehavior.machine`.

Now what is left is to execute the *llfsm* `ePuckBehavior.machine`. This requires a `Makefile` similar to the one in the illustration of `clfsm` with `ROS`. The machine and all the files we have have discussed are part of a download. But you learn more by entering the code yourself. If you are into the details of `bmake`, the changes are similar to the necessary changes for `catkin`.

1. We need to say where to find the includes that define the class-oriented messages we have built.

```
EPUCKNXT_DIR?=../src/webots_epuck_nxt_differential_robot/include
```

2. We need some include files from `gusimplewhiteboard`, for example those for the watcher.

```
GUSIMPLEWB_DIR?=../src/gusimplewhiteboard/include
```

These new variables are used in the linking flags

```
 ${machine}_cppflags=-I${${machine}_dir} -I${CLFSM_DIR:Q} -I${LIBCLFSM_DIR:Q} -I${ROS_CONVERTED_ICLD}
-I${EPUCKNXT_DIR} -I${GUSIMPLEWB_DIR}
```

Other changes are removing all linking against `ROS` libraries.

Once you have this, in the `machines` directory you compile the machine with `bmake` and the you execute it with `clfsm`.

```
cd $HOME/catkin_ws/machines
bmake
../devel/lib/clfsm/clfsm ePuckBehavior.machine
```

You should observe the behavior in the ePuck for a while until its encoders grow to a value that the machine stops.

An elegant line follower

We will use the camera infrastructure, to build another rapid illustration of using the `gusimplewhiteboard` and out *MiPal* class oriented messages, to reconstruct the behavior of following the line. This is a cleaner presentation of the machine, as opposed to the presentation in the documentation of MIEDITLLFSM. This presentation is cleaner, because we have structured the messages cleanly using the `C`-structure and the classes. This enables much clearer use of the Object-Oriented nature of the messages, We will illustrate this shortly.

Thus, we use MIEDITLLFSM to construct a new machine `FollowLine.machine`. Place this machine just beside the earlier `ePuckBehavior.machine` in the directory `$HOME//catkin_ws/machines`. The include section is essentially the same but to see some output we will add `#define DEBUG`. So in the `include` you type the following.

```
#include <stdio.h>
#include "CLMacros.h"
#include "webots_epuck_nxt_differential_robot/WebotsEPuckNXTdifferentialRobotControlStatus.h"
#include "gugenericwhiteboardobject.h"
#include "guwhiteboardwatcher.h"
using namespace guWhiteboard;
#define DEBUG
```

We will sue the following variables.

```
Print_t print_ptr;
int robotID;
WebotsEPuckNXTdifferentialRobotControlStatus a_robot;
WebotsEPuckNXTdifferentialRobotControl_t control_Handler;
WebotsEPuckNXTdifferentialRobotStatus_t status_Handler;
int speedToUse;
int rightSpeed;
int leftSpeed;
int cameraWidth;
CAMERA_E_PUCK_CHANNELS theChannel;
CAMERA_E_PUCK_CHANNELS leftChannel;
CAMERA_E_PUCK_CHANNELS rightChannel;
float maxSpeed;
int delta;
```

We are ready to define the states. The first state is INITIAL. It only has an **OnEntry** section for some initialization.

```
#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
leftChannel=GREEN_CHANNEL;
//Follow blue
rightChannel=RED_CHANNEL;
//Follow yellow
theChannel=BLUE_CHANNEL;

a_robot=status_Handler.get();
a_robot.the_camera().set_on(true);
control_Handler.set(a_robot);
```

We add code to the **Internal** section as well.

```
statusCheck=status_Handler.get();
a_robot=status_Handler.get();
a_robot.the_camera().set_on(true);
control_Handler.set(a_robot);
```

The machine moves from this state to the state TURN_ON_ENCODERS with the transition labeled with `statusCheck.the_camera().on()`. The behavior does not start unless the camera is on The turning of encoders follows the standard pattern of retrieving the status message, using it to change the desired setting, and reposting it as a control. So the **OnEntry** section is as follows.

```
#ifdef DEBUG
std::string stateName("STATE: "); stateName+=state_name(); print_ptr(stateName);
#endif
// turn on encoders
a_robot=status_Handler.get();
a_robot.left_encoder().set_on(true);
a_robot.right_encoder().set_on(true);
control_Handler.set(a_robot);
```

Note the more elegant object-oriented notation we now get with constructions like `a_robot.left_encoder().set_on(true);`.
It almost reads itself what is happening. This state has a transition to GET_MAX_SPEED_AND_CAMERA_WIDTH: namely, `after_ms(30)`.

Now, lets look at the **OnEntry** section of the state GET_MAX_SPEED_AND_CAMERA_WIDTH:

```
#ifdef DEBUG
std::string stateName("STATE: "); stateName+=state_name(); print_ptr(stateName);
#endif
a_robot=status_Handler.get();
maxSpeed=M_PI * a_robot.max_times_radians_speed();
cameraWidth=a_robot.the_camera().camera_width();
#ifdef DEBUG
```

```
fprintf(stderr,"maxSpeed Read %f cameraWidth %d\n",maxSpeed,cameraWidth);
#endif
```

This state has an **OnExit** where we turn off the encoders. If fact, you can now analyze he driver and realize that you could structure it so that there is no need to turn the encoders in order to get the maximum speed. The maximum speed is actually now set in the constructor in `GUWebotsDriver.cpp`. So, this machine can now be simplified, but we are just imitating the one described in the *How to* for MIEDITLLFSM. But one improvement we have already performed here is to obtain the `cameraWidth` in the state GET_MAX_SPEED_AND_CAMERA_WIDTH and not on the control loop. The **OnExit** is a s follows.

```
// turn off encoders
a_robot=status_Handler.get();
a_robot.left_encoder().set_on(false);
a_robot.right_encoder().set_on(false);
control_Handler.set(a_robot);
```

From GET_MAX_SPEED_AND_CAMERA_WIDTH we go to the state FEEDBACK_CONTROL with one transition labeled `after_ms(30)`.

The state FEEDBACK_CONTROL has only an **OnEntry** section and it is a clear display of a *control loop feedback mechanism*, at least the calculation of the error, which will placed in the variable `delta`.

```
#ifdef DEBUG
std::string stateName("STATE: "); stateName+=state_name(); print_ptr(stateName);
#endif
delta = a_robot.the_camera().median(theChannel) -cameraWidth/2;
// set the speeds
leftSpeed= speedToUse -4*abs(delta)+4*delta;
rightSpeed=speedToUse -4*abs(delta)-4*delta;
#ifdef DEBUG
fprintf(stderr,"delta %d\n",delta);
#endif
```

The control loop now moves to the correction to the power of the motors, so this is a derivative control loop. This is achieved with transition with label `after_ms(10)` to SET_MOTORS_SPEED. The state SET_MOTORS_SPEED has a transition back to FEEDBACK_CONTROL with label `after_ms(10)` as well. Thus, we only need the **OnEntry** section for state SET_MOTORS_SPEED.

```
#ifdef DEBUG
std::string stateName("STATE: "); stateName+=state_name(); print_ptr(stateName);
#endif
a_robot=status_Handler.get();
a_robot.left_motor().set_motor_power(static_cast<int16_t>(leftSpeed/maxSpeed));
a_robot.right_motor().set_motor_power(static_cast<int16_t>(rightSpeed/maxSpeed));
control_Handler.set(a_robot);
```

With your experience from the earlier machines you should be able to add this one to the `Makefile` and compile it with `bmake`. Then test it.