OPEN-R Essentials

by Ricardo A. Téllez, v1.0

4th September 2004

Contents

1 Introduction and description

OPEN-R SDE is the C++ environment provided by Sony to program their Aibo robots. It consists of a group of APIs, file structures and compilation tools that allow the creation of control programs for the robot from a Windows, Macintosh or Linux platform. An OPEN-R program consists of a set of objects that are executed concurrently on the robot, plus a set of configuration files that specify how those objects must interact. The objects are cross-compiled in the host machine producing Aibo code. This code is then transferred to a memory stick that is inserted on the Aibo robot and executed on it. OPEN-R SDE provides of all the tools required for all those tasks.

Before continuing, please install the OPEN-R SDE on your computer. This text does not include an installation guide of the OPEN-R SDE and it will only focus on the explanation of the OPEN-R interface. This text will concentrate on development from a Linux machine, but all platforms follow almost the same steps. Refer to the official OPEN-R documentation or to the *Aibo quick-start guide*[1] for installation instructions and additional information for other platforms.

1.1 Objects description

An **OPEN-R object** could be defined as a process running on Aibo. An **OPEN-R program** is just a set of those objects running concurrently (in parallel). OPEN-R objects **communicate** with each other in order to coordinate. Thus, a program to control Aibo will consist of a set of OPEN-R objects, each object performing its own job but coordinating with the other objects by using message passing. An OPEN-R object corresponds to one executable file for Aibo created at compile-time. When Aibo boots, all the compiled objects are loaded into memory and started as concurrent processes. Their message interchange will determinate the flow of action.

OPEN-R objects behave in a similar way to that of a finite state automaton: each object is composed of a set of **internal states**. **Transitions** govern the change between states. Having this idea in mind, François Serra and Jean-Christophe Baille at ENSTA developed a useful mechanism for a graphical description of the workings of an object. We will borrow here their methodology in order to describe the insights of OPEN-R objects in a graphical way[2].

During our OPEN-R description and explanation, we will describe and use the BallTrackingHead7 sample program as an example. This program, included in the OPEN-R SDE makes Aibo look for its pink ball and play some sound when detected or lost. On that example, the program is composed of four main objects: BallTrackingHead, MovingLegs, MovingHead and LostFoundSound (there are also three other special objects on the play, that are described in sections 1.3 and 1.5.5).

The behaviour of every object is described as the transition between its internal states. Each object is based on its present state and the transitions that lead to other states. This means that an object will be always on a state.

[1] Made by the same author of this document and available at http://www.ouroboros.org/~rt71592

[2] All credit for the development of the method goes to the mentioned authors. For more information on their methodology, please check their original work titled *Aibo programming using OPEN-R SDK*, that can be found at http://www.ensta.fr/~baillie/openr_tutorial.html

The design of an object is the design of its required states, transitions and functions to apply when going from one state to another. This design can be really helped by the previous design by using the Serra-Baille charts, in a similar way as UML helps software design.

Summarising, to design an object, it must be specified its internal states. An object can be only in one state at a time. Objects change their state by means of transitions. Transitions are activated by the reception of messages, and can have several paths going to several states. Only the path that satisfies the condition will be taken. Conditions must be exclusive in order to not allow the object be in two different states at the same time.

1.2 Object communication

Objects communicate with each other by message passing. In every inter-object communication the sender of the message is called the **subject** and the receiver is called the **observer**. An object can act as subject in one situation but as observer in another one. It just depends on the situation.

The communication channel between two objects is unidirectional (each channel has a fixed subject and a fixed observer). It means that, if bidirectional communication is required between two objects, then two different channels will be required. Also, an object can have different subjects and be the observer of several objects, but a communication channel will be required for each one. Messages go out of the object and come in through **gates**. There is a gate in the object for each communication channel.

Since objects are single threaded, it means that they can only process one single message at a time. For this purpose, a message queue is implemented in every object, where messages wait their turn to be processed.

The main flow execution of an object is the following:

1. The object is initialised. Then it sends an AR message to all its subjects.
2. The object waits on a determined state the arrival of a message coming from one of its subjects
3. Once the message arrives, the method associated to that message is activated. Within that method, the message is processed
4. Once finished the processing, the object sends an AR message to the subject indicating that is already ready to receive new messages
5. It returns to point 2.

AR messages are special messages defined within OPEN-R. These messages, whose complete name is ASSERT_READY, are part of the synchronisation protocol used to let the observer notify the subject that it is ready to receive new messages. This is usually done after the last received message has been processed.

Messages sent between objects can be of any C++ primary type, or an array, an structure, a class or a pointer. However, their length is limited to the size of OCommandVectorData (more on this on chapter 3). For this reason, it is a common technique

to send pointers to common shared memory as messages, and put on that shared memory the information that wants to be sent.

1.3 System objects

Together with the objects created by the designer of the program, there exist other objects already created by OPEN-R that allow the access to the hardware of the robot. It means that access to sensors or actuators will be done through an OPEN-R object provided by the system. Those objects are:

- OVirtualRobotComm: it interfaces with the robot joints, sensors, LEDs and camera
- OVirtualRobotAudioComm: it interfaces with the robot audio devices for recording of playing sounds

The use of those objects in the programs is the same as the use of programmer defined objects. Some input and output points (the gates) in those objects are already defined to send and receive messages from/to them.

1.4 The graphic formalism

The graphic formalism for the description of the objects will be described here. This formalism will be of great help when designing OPEN-R programs, since all the requirements would be specified by using a king of graphic chart, in the same way as UML can be used for the description of general C++ programs.

First, the object is defined by a big square. That square will contain all the rest of information that defined the insights of the object. States of the object are represented by circles containing their state name. Transitions between states are expressed by using arrow lines with a black square in the middle. Those arrow lines contain several other information:

- The transition event: transitions between states are activated by the reception of messages from other objects.
- The transition condition: the reception of a message (event) can trigger different transitions going to different states. The transition to one state or another will depend on which of the paths satisfies the condition specified on top of the black square. Transition conditions must be exclusive, since one state cannot lead to two different states at the same time.
- The code execution: If there exist a subroutine name specified bellow the black square, then its code must be executed.
- The message to send: If there exists a dotted arrow line going out of the line, then a message must be sent.

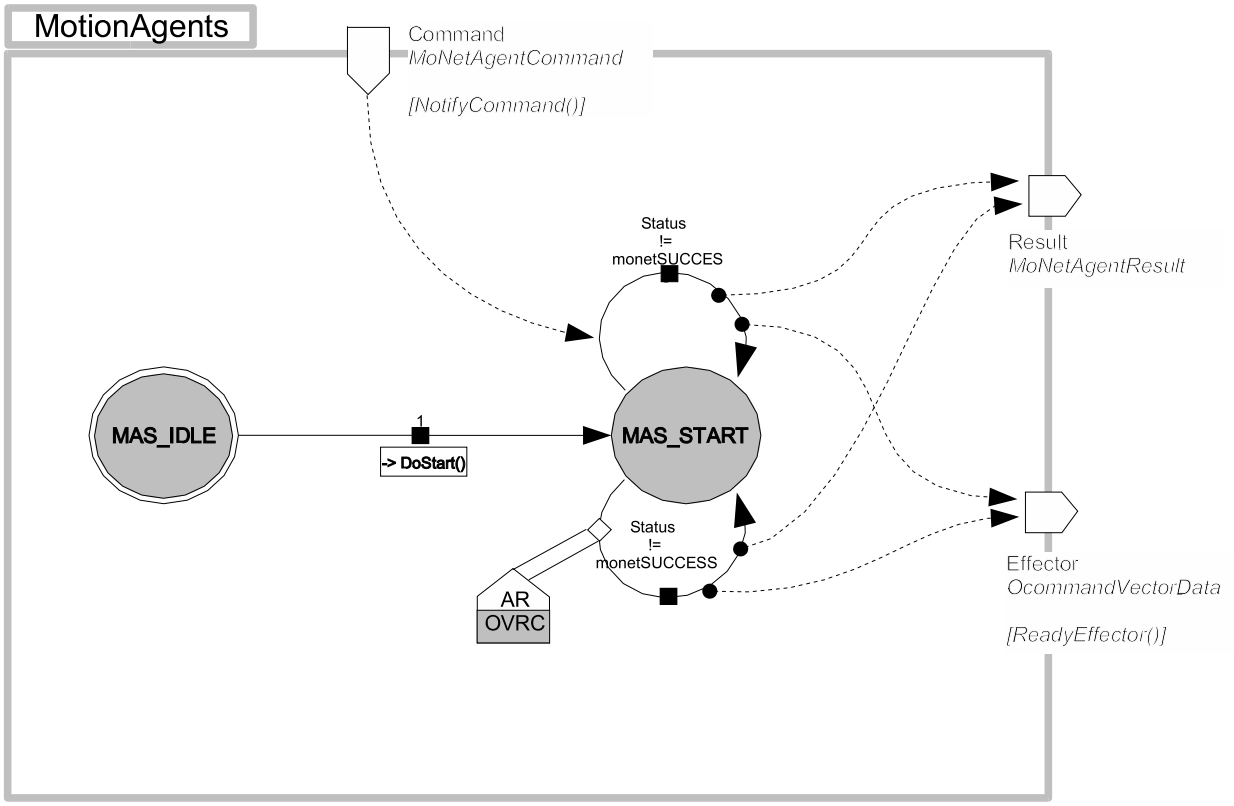

Figure 1: An example of design diagram of the MotionAgents object from the Ball-TrackingHead7 program

Messages are represented by dotted arrow lines. Messages coming from another object, enter the object via gates. Gates are represented by a trapezoid form entering the object or going out of it (the arrow indicates its direction). Gates have several data on its top:

- The first word is the name of the gate
- Second word is the type of message that will go trough that gate
- Third, if it exists, will be the name of the function to be executed when a message is received (in the case of an incoming gate), or when an AR is received (in the case of an outgoing gate)

In order to simplify the scheme, messages received from OVirtualRobotComm and OVirtualRobotAudioComm are represented using a special trapezoid box. Those trapezoid box contain two words: OVRC and AR, when the message comes from the OVirtualRobotComm, or OVRAC and AR, when the message comes from the OVirtualRobotAudioCom. Those boxes represent the reception of a message coming from one of those virtual objects. Instead of drawing a dotted message line coming from the virtual objects to the point in the object where the message is received, the authors of the method have decided to use that trapezoid box for the sake of simplicity, but the meaning is the same a dotted line, it is, the reception of an AR message.

1.5 The ball tracking example

The ball tracking example is one of the program examples provided by Sony. It is called BallTrackingHead7 and consists of four main objects. The program has the following behaviour: first, it starts an initialisation stage where the legs and the head of the robot are brought to an initial position. From that position, the robot stands up. Once it is up, the robot starts searching for the ball moving its head around. If the ball is found, then a sound is played and Aibo focus its attention to the ball. If the ball is lost, then another sound is played and Aibo starts looking again for it.

In this program the BallTrackingHead7 object is the leading object. It orchestrates the actuation of the rest of objects and decides when they must act. It is not necessary to have such a leading object in OPEN-R programs, but it is usually used for convenience.

1.5.1 BallTrackingHead7 object

This object consists of six main states (fig. 2). The object is initialised in BTHS7_IDLE state, but it automatically moves to the BTHS7_WAITING_STAND2STAND_RESULT state. By doing that transition, a message is sent to the MoNet object. That message says to the object that it must perform their movement tasks in order to bring Aibo to the stand up position. The BallTrackingHead7 object waits for a reply message from the MoNet, indicating that the movement task is complete. When received the message, the object will change to the BTHS7_SEARCHING_BALL state, releasing a message to the OVirtualRobotComm object making it to move Aibo's head around its space.

The object will remain on that state until it founds the ball. The detection of the ball is done by looking at the messages received from the *Image* incoming gate. That message comes from the OVirtualRobotComm object containing an image (of type OFbkImageVectorData). Once received that message the object can follow two different paths:

1. If the pink colour is detected and greater than a BALL_THRESHOLD threshold, then it would mean that the ball is present and the image counter for balls *found* is incremented. Then, if the *found* counter exceeds the found threshold FOUND_THRESHOLD the object changes to BTHS7_TRACKING_BALL state and sends two messages to MoNet object: one to play a sound and another to move the head towards the ball.

2. If the pink colour is not detected then the *found* counter is set to zero and the object returns to its BTHS7_SEARCHING_BALL state, waiting for another image.

Once in the BTHS7_TRACKING__BALL state the robot follows a similar behaviour to that of the BTHS7_SEARCHING_BALL state. It will continue on that state as long as the images received from OVirtualRobotComm indicate that the ball is still there. Once the ball is lost, the object will change to the BTHS7_SEARCHING_BALL state and play a sound, following the same sequence presented for the BTHS7_SEARCHING_BALL state. The BallTrackingHead7 object will keep changing its state from SEARCHING BALL to TRACKING BALL and vice-versa during the time the program is run.

There also exists a wild-card (*) state in this object. This state means that in any state, when received the specified message, some actions will be performed that do not change the current state of the object. In this case, when a message is received from OVirtualRobotComm through the Sensor gate an update of the sensor regions will be performed.

1.5.2 SoundAgent object

This objects is responsible for playing sounds when required by the MoNet object. It starts in SAS_IDLE state an after initialisation it switches to SAS_START state. The object will wait on that state until it receives a message from the MoNet object indicating that a sound must be played. At this point, the object changes to SAS_PLAYING state, where it waits for the OVirtualRobotAudioComm to send an AR message indicating that it is ready to receive some WAV data to play the sound. When ready, the object sends to it the data to be played until everything has been sent (played). Once the memory has been released of the WAV data, then the object returns to the SAS_START state waiting for another message from MoNet object to play more sounds.

1.5.3 MoNet object

This object is in charge of doing all the job related to movement and sound playing. For example, it is in charge of bringing Aibo to the standing position. Every time

that a position must be reached, this object receives a message indicating what position needs to be achieved. The object loads then the step movements required and performs it. It also has care of playing sounds. The object starts in MNS_IDLE state and changes to MNS_START automatically. It stays there until a message arrives from the BallTrackingHead7 object via the *ClientCommand* gate. After checking that the command received is valid, the object starts performing the action required. It sends then the required orders to SoundAgent and/or to MotionAgents and changes to state MNS_AGENT_RUNNING. The object will remain in that state until it receives a message from the previous objects indicating that the action has been completed. In that case, the object sends a result message to the BallTrackingHead7 object indicating it has finished, and returns to its stay state (MNS_START).

1.5.4 MotionAgents object

This object is in charge of sending the correct orders to the OVirtualRobotComm object in order to bring Aibo to the standing position. It starts in MAS_IDLE state an changes to MAS_START automatically. There, it wait for a message from MoNet object that requests the implementation of the movement. Then the object communicates with OVirtualRobotComm sending the correct orders to move Aibo to the desired position. Once it finishes the movement, it sends a message to MoNet and waits in MAS_START state for another command from MoNet.

1.5.5 Other objects

Three other objects are required in order to handle with some useful specific task. Those objects are the following:

- OVirtualRobotComm: as has been explained before this object is part of the operating system and must not be created or included. It is in charge of accessing the sensors and actuators of the robot.
- OVirtualAudioComm: this object is in the same situation as the previous one. It is in charge of the audio interaction of the robot.
- PowerMonitor: this object handles all the stuff related to switching off the robot. It is also included as an example from Sony, and automatically included when compiling the ball tracking example. You should include this object explicitly in your programs in order to be able to shutdown Aibo, but for the ball tracking example it is included automatically by the makefile. Its mission is to check for some conditions that may require the shutdown of the robot, like, battery level and temperature, pressing the pause button, connecting to the charging station, etc.

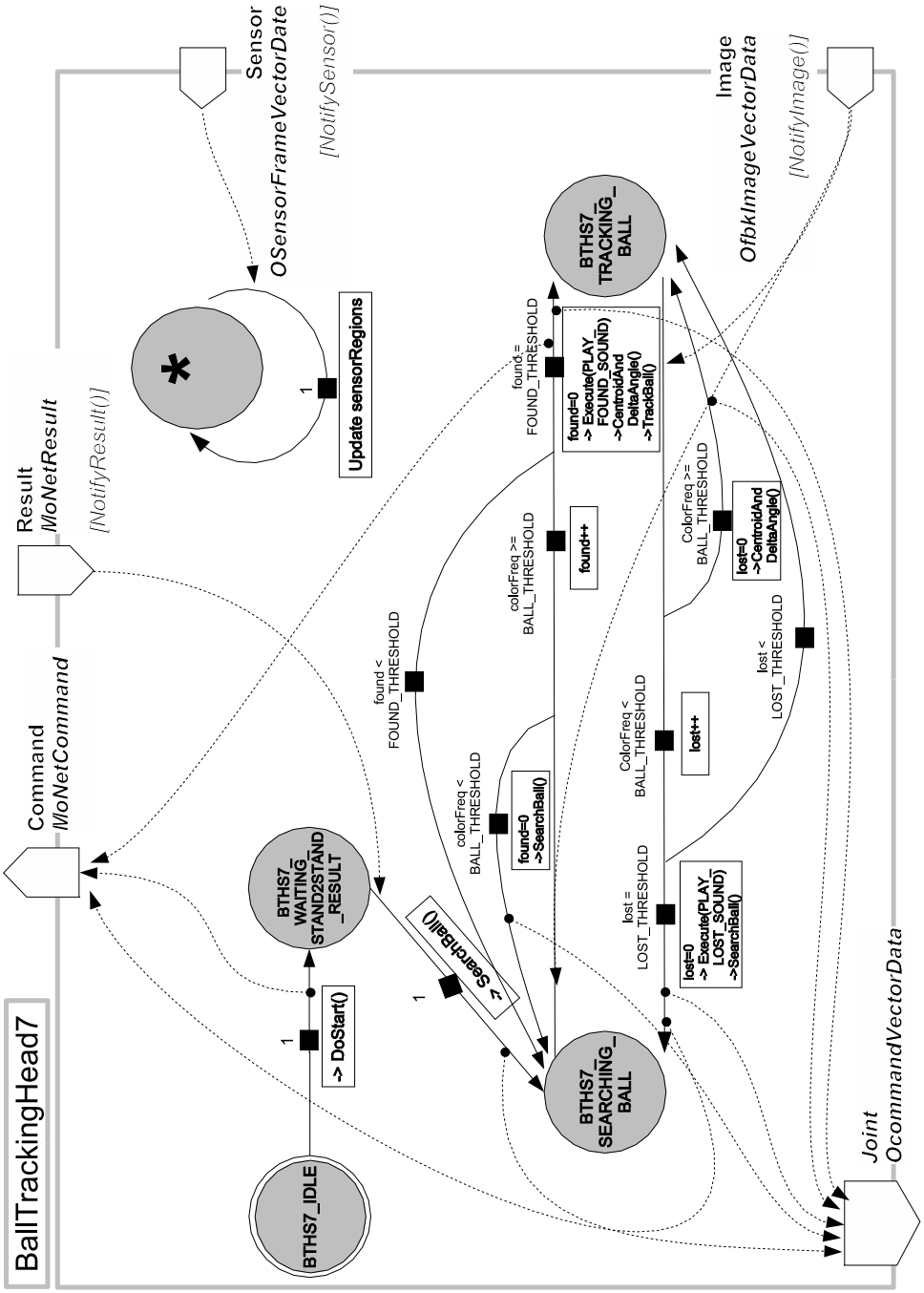

Figure 2: Diagram of the BallTrackingHead7 object

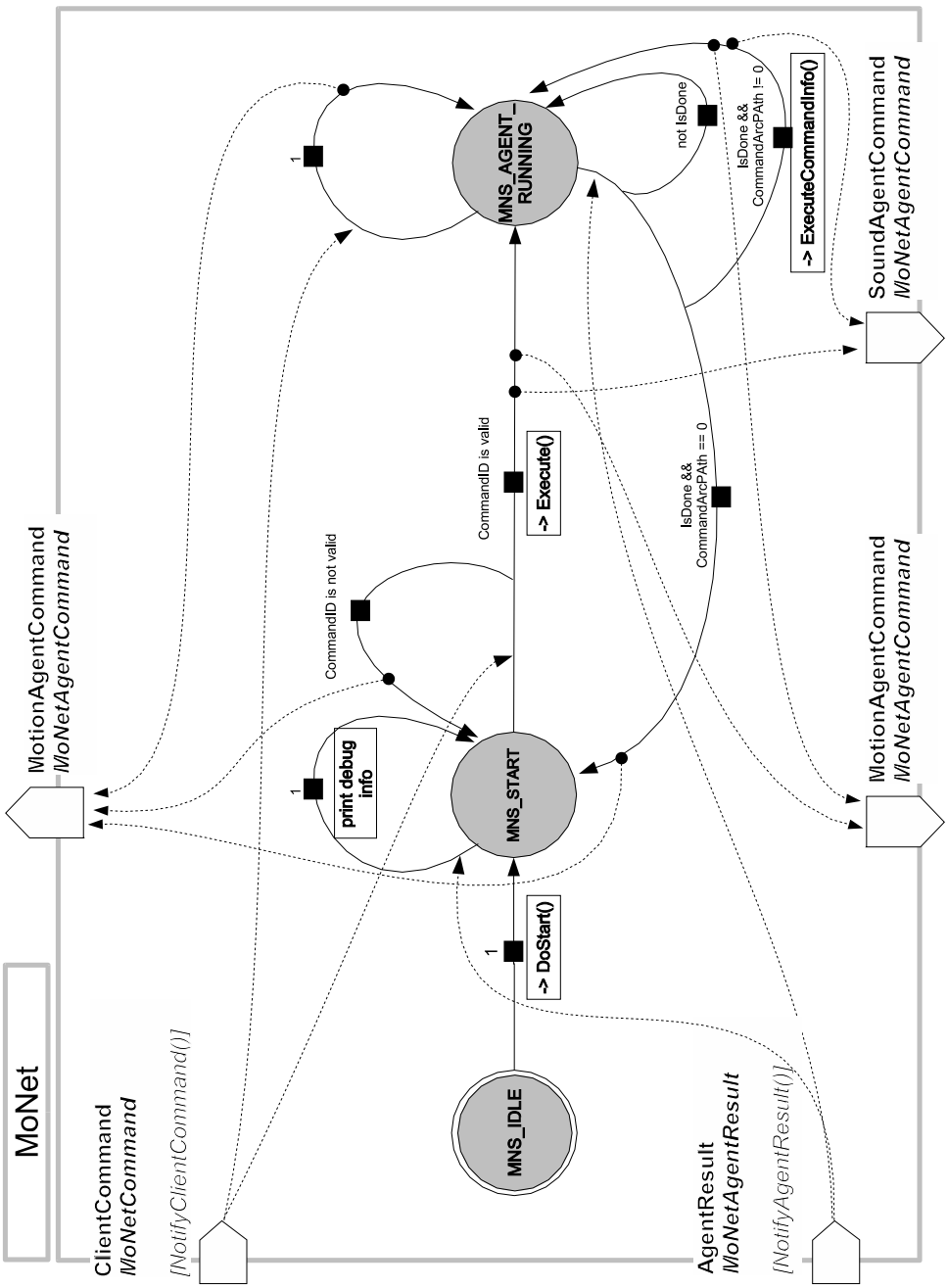

Figure 3: Diagram of the MoNet object

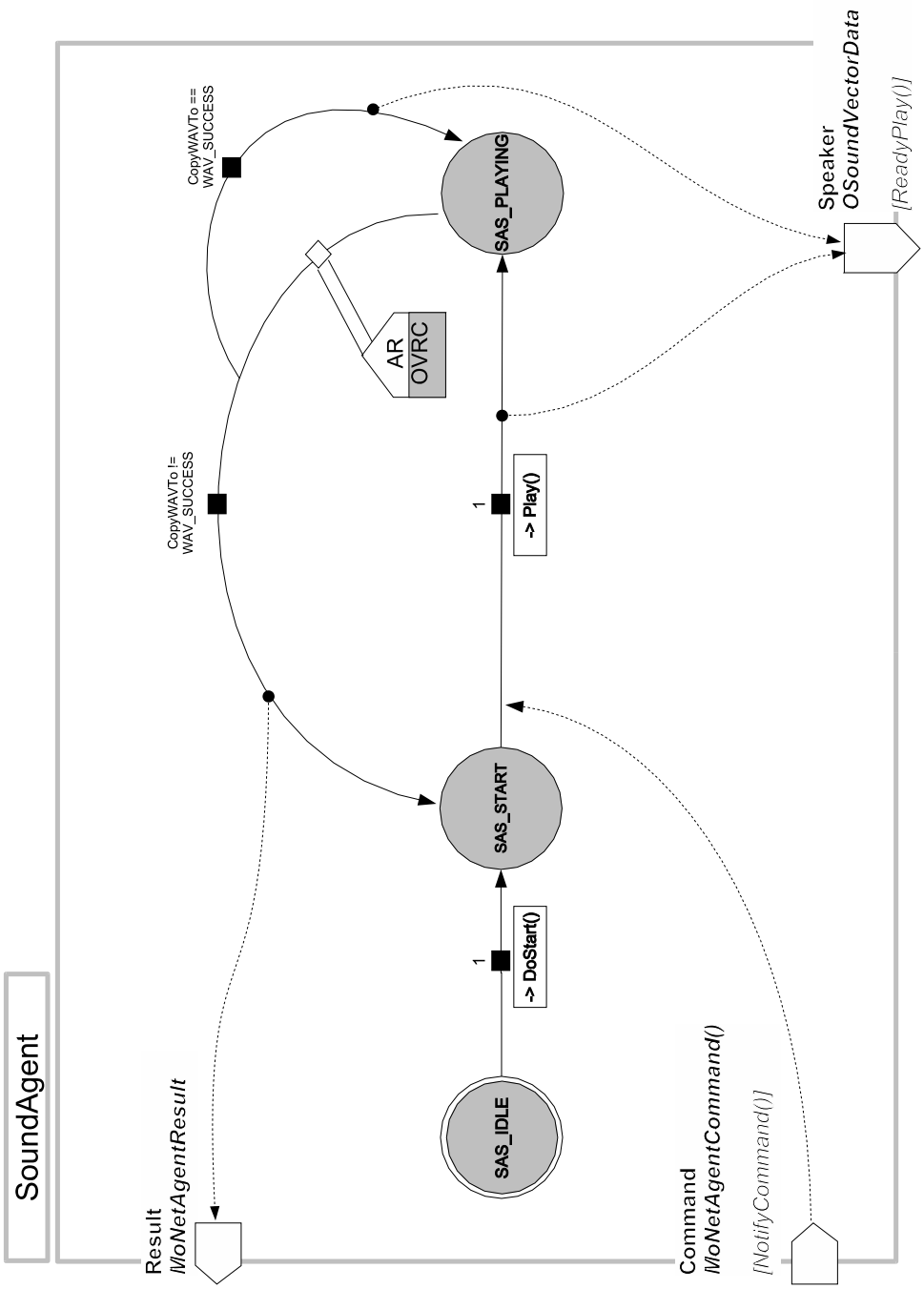

Figure 4: Diagram of the SoundAgent

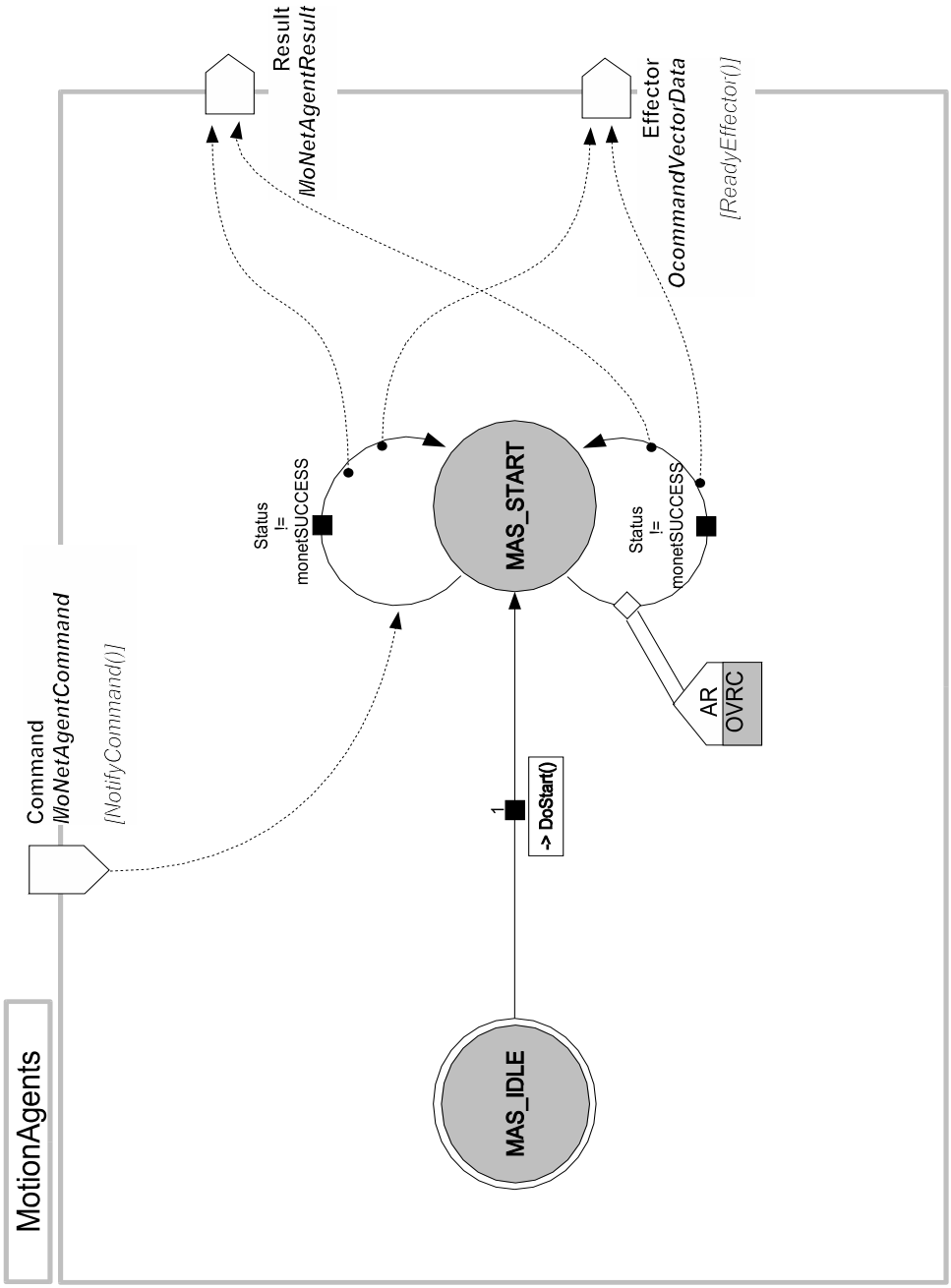

Figure 5: Diagram of the MotionAgents

2 Main structure description and implementation

On chapter one a methodology to graphically describe the behaviour of an OPEN-R program was explained. Now it is time to code all that information into C++ code that will generate the Aibo controller. This chapter describes how to transfer the ideas expressed in the graphical method to computer code. The BallTrackingHead7 program will be used to describe the different steps.

The flow of development for an OPEN-R includes the following steps:

1. **Design of the objects**: this steps includes the design of the required objects for the program to develop, including the flow of data between them and the type of messages interchanged. The graphical method explained on chapter one must be used here.
2. **Description of the gates**: for every object a stub.cfg file must also be created. This contains a description of the object gates and how they connect. More on this on section 2.2.3
3. **Implementation of the object**: creation of the C++ object that will define the OPEN-R object, including some virtual functions (see section 2.1) and the ones specified in the stub.cfg file.
4. **Build the executable**: generate all the compiled objects.
5. **Edit some configuration files**: some additional files will require modification to include how objects connect with each other (CONNECT.CFG file) and which objects must be executed (OBJECT.CFG file).
6. **Execute and debug in Aibo**: after transferring the program to the memory stick, some errors may arise. They can be traced by using the wireless console and inserting some comments in the code.

As has been presented in the previous chapter, an OPEN-R program is a event-oriented one. Every object is basically composed of a set of routines that will act as an answer to several events. That is the idea to have in mind when designing an Aibo program.

2.1 The base class

The base class is the C++ class that will represent an object in OPEN-R. Each object created by the programmer will inherit from the base class and will be represented by only one of those class. The base class name is OObject and contains 4 virtual functions:

- OStatus DoInit (const OSystemEvent& event)
- OStatus DoStart (const OSystemEvent& event)
- OStatus DoStop (const OSystemEvent& event)
- OStatus DoDestroy (const OSystemEvent& event)

Those functions perform some basic functions that will guide the start-up of the object and its shutdown, and they must be implemented in the code of the programmer's object. Most of the code of those functions will be handled by some macros provided by OPEN-R. DoInit and DoStart are initialisation functions. They are called automatically by this order by OPEN-R once the object has been loaded into memory. DoStop and DoDestroy are called when OPEN-R is shutting down Aibo.

When creating an OPEN-R object, it inherits from the base class. The new object created has to redefine the previous virtual functions inside its object construction code. That code has also to define two other things:

1. In order to be able to communicate with other objects, the constructor of the class must define two arrays, one that will contain a list of the subjects (called *subject*) and another for the list of the observers (called *observer*). The programmer has just to care about defining those arrays and use them, but all the construction and filling process is done automatically by the virtual functions described above (see below).

2. The constructor of the object must indicate its starting state (usually, it starts in IDLE state), and how many other states do exist for that object.

2.1.1 DoInit()

This function is called when the object is just loaded in the memory of Aibo. It sets up all the gates of the object and registers all its observers and subjects. In order to do all those tasks easily, OPEN-R provides some macros that perform the job.

The following code is a skeleton for the function

```
OStatus ASampleClass::DoInit(const OSystemEvent& event)
{
    /* OPEN-R macros */
  /* registers observers and subjects */
  NEW_ALL_SUBJECT_AND_OBSERVER;
  /* registers connection to services offered
    by other objects*/
  REGISTER_ALL_ENTRY;

/* registers entry points for receiving messages */
  SET_ALL_READY_AND_NOTIFY_ENTRY;

/* Specific code for the ASampleClass goes here */
  return oSUCCESS;
}
```

The following code is the DoInit function of the BallTrackingHead7 object:

```
OStatus BallTrackingHead7::DoInit(const OSystemEvent& event)
```

```
{
  OSYSDEBUG(("BallTrackingHead7::DoInit()\n"));
  /* OPEN-R macros */
  NEW_ALL_SUBJECT_AND_OBSERVER;
  REGISTER_ALL_ENTRY;
  SET_ALL_READY_AND_NOTIFY_ENTRY;

/* Specific code for the BallTrackingHead7 example */
  OpenPrimitives();
  NewCommandVectorData();
  SetCdtVectorDataOfPinkBall();
  return oSUCCESS;
}
```

2.1.2 DoStart()

This function is called once DoInit is finished in all the existing objects. This function sends an AR message to all its observers, and changes from IDLE state to the next state required. The general code:

```
OStatus ASampleClass::DoStart(const OSystemEvent& event)
{
  /* the programmer code goes here */

state = /* the required state, usually START*/ ;
  /* the OPERN-R macros */
  ENABLE_ALL_SUBJECT;
  ASSERT_READY_TO_ALL_OBSERVER;
  /* code can be also added here */
  return oSUCCESS;
}
```

For the BallTrackingHead7 object:

```
OStatus BallTrackingHead7::DoStart(const OSystemEvent& event)
{
  OSYSDEBUG(("BallTrackingHead7::DoStart()\n"));
  Execute(STAND2STAND_NULL);
  state = BTHS7_WAITING_STAND2STAND_RESULT;
  ENABLE_ALL_SUBJECT;
  ASSERT_READY_TO_ALL_OBSERVER;
  return oSUCCESS;
}
```

2.1.3 DoStop()

This function is called when a shutdown is been performed. The object must return itself to the IDLE state, and to deactivate gates. It also sends a DEASSERT_READY message to the observers, indicating that the object cannot receive more messages.

```
OStatus ASampleClass::DoStop(const OSystemEvent& event)
{
  /* programmer's code goes here */
  state = IDLE;
  /* OPEN-R macros*/
  DISABLE_ALL_SUBJECT;
  DEASSERT_READY_TO_ALL_OBSERVER;
  /* programmer's code goes here */
  return oSUCCESS;
}
```

For the BallTrackingHead7 object:

```
OStatus BallTrackingHead7::DoStop(const OSystemEvent& event)
{
  OSYSDEBUG(("BallTrackingHead7::DoStop()\n"));
  state = BTHS7_IDLE;
  DISABLE_ALL_SUBJECT;
  DEASSERT_READY_TO_ALL_OBSERVER;
  return oSUCCESS;
}
```

2.1.4 DoDestroy()

This function is called after DoStop has been called in all objects. It just deletes all subjects and observers and ends the object. This function usually remains as in the example.

```
OStatus ASampleClass::DoDestroy(const OSystemEvent& event)
{
  DELETE_ALL_SUBJECT_AND_OBSERVER;
  return oSUCCESS;
}
```

2.1.5 Constructor of ASampleClass

Here it is a complete description of our object ASampleClass. You can use the code below as a starting template for your own objects.

- First, the definition of the header file ASampleClass.h

```
#ifndef ASampleClass_h_DEFINED
#define ASampleClass_h_DEFINED
#include <OPENR/OObject.h>
#include <OPENR/OSubject.h>
#include <OPENR/OObserver.h>
#include <OPENR/ODataFormats.h>
/* def.h is defined by the compiler during compilation */
#include “def.h”
/* enumerate the states of the object */
enum ASampleClassStates { IDLE,
/* add here your states */
};
class ASampleClass : public OObject {
public:
  ASampleClass();
  virtual ~ASampleClass() {}

/* the following are the arrays of observers and subjects

numOfSubject and numObserver are defined during
    compilation */
  OSubject* subject[numOfSubject];
  OObserver* observer[numObserver];

virtual OStatus DoInit (const OSystemEvent& event);

virtual OStatus DoStart (const OSystemEvent& event);

virtual OStatus DoStop (const OSystemEvent& event);

virtual OStatus DoDestroy (const OSystemEvent& event);
private:
  ASampleClassStates aSampleClassState;
};
#endif // ASampleClass_h_DEFINED
```

- Second, the code of the class

```
ASampleClass::ASampleClass ()
{
  aSampleClassState = IDLE;
}
OStatus ASampleClass::DoInit(const OSystemEvent& event)
{
    /* OPEN-R macros */
```

```
  /* registers observers and subjects */
  NEW_ALL_SUBJECT_AND_OBSERVER;
  /* registers connection to services offered
    by other objects*/
  REGISTER_ALL_ENTRY;

/* registers entry points for receiving messages */
  SET_ALL_READY_AND_NOTIFY_ENTRY;

/* Specific code for the ASampleClass goes here */
  return oSUCCESS;
}
OStatus ASampleClass::DoStart(const OSystemEvent& event)
{
  /* the programmer code goes here */

state = /* the required state, usually START*/ ;
  /* the OPERN-R macros */
  ENABLE_ALL_SUBJECT;
  ASSERT_READY_TO_ALL_OBSERVER;
  /* code can be also added here */
  return oSUCCESS;
}
OStatus ASampleClass::DoStop(const OSystemEvent& event)
{
  /* programmer's code goes here */
  state = IDLE;
  /* OPEN-R macros*/
  DISABLE_ALL_SUBJECT;
  DEASSERT_READY_TO_ALL_OBSERVER;
  /* programmer's code goes here */
  return oSUCCESS;
}
OStatus ASampleClass::DoDestroy(const OSystemEvent& event)
{
  DELETE_ALL_SUBJECT_AND_OBSERVER;
  return oSUCCESS;
}
```

2.2 Communications between objects

For the communication between different objects, several issues must be taken into account. First, a way of identifying the subject must be provided. For this reason, OPEN-R creates during compile time two different arrays that allow the programmer to access subjects and observers: the *subject[]* array and the *observer[]* array. Second, OPEN-R also defines a set of indexes for easy access to the array. Those indexes are

also generated at compile time based on the information provided in the stub.cfg file. The stub.cfg file contains information about the gates (also called services) of the object (more about this file in section 2.2.3). Since every gate has a name, the array indexes are created in the following way: every index is identified by a name. The index name is formed by the concatenation of the type of the service (*sbj* for subject and *obs* for observer) and the name of the gate. Then, for the BallTrackingHead7 object, there is an outgoing gate called Command that connects the object with MoNet. If it were necessary to access that subject, the sentence would be *subject[sbjCommand]*.

2.2.1 Sending a message

The following steps are required to send a message to an observer:

1. Initialise the message's content
2. Assign the message to the service
3. Notify the observer

Here is how to implement those functions inside a program (in this case, the BallTrackingHead7 object sends a command to the MoNet object):

```
void BallTrackingHead7::Execute(MoNetCommandID cmdID)
{

/* creates the message and initialises its content */
  MoNetCommand cmd(cmdID);
  /* assigns the message to a service */
  /* (in this case, a service called Command */

subject[sbjCommand]->SetData(&cmd, sizeof(cmd));
  /* notifies the observers */
  subject[sbjCommand]->NotifyObservers();
}
```

2.2.2 Receiving a message

The following steps are required to receive a message:

1. Retrieve message's content by casting it
2. Process the message
3. Send an AR message to the subject that sent the message

Here is how to implement those functions

```
void BallTrackingHead7::NotifyResult(const ONotifyEvent& event)
{

/* retrieve a message of type MoNetResult comming
    from the Result gate */

MoNetResult* result = (MoNetResult*)event.Data(0);
...

/* sends an AR message to the subject that sent
    the message. This must be done when all the
    processing of the message has finished */
  observer[event.ObsIndex()]->AssertReady();
}
```

It must be paid attention to two things:

1. The *observer* array is accessed through an index obtained from the message itself, and not using the typical *obsXXX* pattern.
2. The name of the function defined here. This function is activated whenever a message comes from a subject through a gate. To specify which functions must activate to which messages the stub.cfg file must be created.

2.2.3 The stub.cfg config file

The stub.cfg file is the file that describes how the object connects with other objects. It describes the gates, including all the information explained in section 1.4 to completely describe a gate. Every object has its own stub.cfg file, an the information of this file will be used by the compiler when building the binaries. The file must be placed in the same directory where the C++ object program resides.

Here there is an example of stub.cfg file:

```
ObjectName : BallTrackingHead7
NumOfOSubject : 2
NumOfOObserver : 3
Service : "BallTrackingHead7.Command.MoNetCommand.S", null, null
Service : "BallTrackingHead7.Result.MoNetResult.O", null, NotifyResult()
Service : "BallTrackingHead7.Sensor.OSensorFrameVectorData.O", null, Noti
Service : "BallTrackingHead7.Image.OFbkImageVectorData.O", null, NotifyIm
Service : "BallTrackingHead7.Joint.OCommandVectorData.S", null, null
```

The first line describes the name of the object. Second and third line describe the number of subjects and observers the object has. Lines starting by Service are the ones that describe the information included in the label gates of section 1.4. For example, in line

```
Service : "BallTrackingHead7.Result.MoNetResult.O", null, NotifyResult()
```

BallTrackingHead7 is the name of the current object. *Result* is the name of the gate the message will go through. *MoNetResult* is the type of message been interchanged, and *O* means that the gate is incoming (the current object is an observer). If the object was a subject then the last field should say *S*. The null is the name of a function to be executed when a connection result is received. Most of the cases is null. The *NotifyResult* is the name of the function to be executed when an AR or a message is received through this gate.

2.2.4 The connect.cfg config file

The connect.cfg file is a configuration file that specifies how objects interconnect to each other. This is a **unique file per program** and it must be placed in the OPEN-R/MW/CONF/ directory of the memory stick. But before going into the description of the file, it is a good practice to generate a connection graphic containing the information that needs to be coded into the file. Together with the graphic formalism described in chapter 1 perform a powerful mechanism for easy design and comprehension.

In the figure 6 can be see the connections graphic for the BallTrackingHead7 program. In it, each object acting is represented by a circle with its name. Arrows go from subjects to observers an indicate the name of the gates they go through.

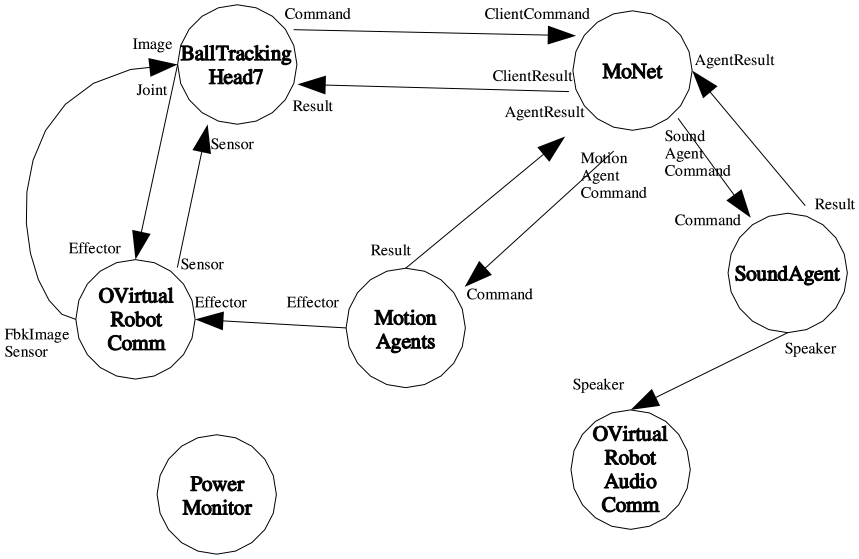

Figure 6: Graphic representation of the connect.cfg file of the BallTrackingHead7 program

The connect.cfg file for the BallTrackingHead7 program is the following:

```
#
# BallTrackingHead7 <--> MoNet
#
BallTrackingHead7.Command.MoNetCommand.S MoNet.ClientCommand.MoNetCommand
MoNet.ClientResult.MoNetResult.S BallTrackingHead7.Result.MoNetResult.O
#
# BallTrackingHead7 <-> OVirtualRobotComm
#
OVirtualRobotComm.FbkImageSensor.OFbkImageVectorData.S BallTrackingHead7.
OVirtualRobotComm.Sensor.OSensorFrameVectorData.S BallTrackingHead7.Senso
BallTrackingHead7.Joint.OCommandVectorData.S OVirtualRobotComm.Effector.O
#
# MoNet <--> MotionAgents
#
MoNet.MotionAgentCommand.MoNetAgentCommand.S MotionAgents.Command.MoNetAg
MotionAgents.Result.MoNetAgentResult.S MoNet.AgentResult.MoNetAgentResult
#
# MoNet <--> SoundAgent
#
MoNet.SoundAgentCommand.MoNetAgentCommand.S SoundAgent.Command.MoNetAgent
SoundAgent.Result.MoNetAgentResult.S MoNet.AgentResult.MoNetAgentResult.O
#
# MotionAgents --> OVirtualRobotComm
#
MotionAgents.Effector.OCommandVectorData.S OVirtualRobotComm.Effector.OCo
#
# SoundAgent --> OVirtualRobotAudioComm
#
SoundAgent.Speaker.OSoundVectorData.S OVirtualRobotAudioComm.Speaker.OSou
```

Lines starting by # are comments, and usually describe the connection between objects being described below. Each no-comment line specifies one connection, starting by the name of the subject and ending by the name of the observer. For example:

```
BallTrackingHead7.Command.MoNetCommand.S MoNet.ClientCommand.MoNetCommand
```

specifies how the BallTrackingHead7 object connects with the MoNet object, acting the first as the subject and the second as the observer. For bi-directional connections, the configuration file must contain two lines, one for each direction of the connection (as happens with the BallTrackingHead7 and MoNet objects connection). The definition of each part of the line follows the same specification as for the stub.cfg file described before. Special attention must be paid to assure that in one line, the messages exchanged between subject and observer have the same type.

3 Interaction with sensors and actuators

This chapter describes how to obtain values sensed by Aibo's sensors and how to send commands to its actuators (joints and LEDs). All this process will be achieved by interacting with the virtual object OVirtualRobotComm provided by OPEN-R. Special treatment is required for the sound (which uses OVirtualRobotSound object).

3.1 Sensors and actuators primitives

In Aibo, every sensor and actuator are also called *primitives*. Each primitive has its own *primitive locator* to access to it. The primitive locator is like the path or the address you must follow to reach the desired sensor or actuator. A list of the available primitive locators is provided by Sony on its *Model Information* documentation. Primitive locators look like this:

PRM:/r1/c1/c2/c3-Joint2:13 <– primitive locator for the HEAD TILT2 motor of ERS-7

The primitive locator cannot be used directly to access the sensor or actuator. Instead of it, the *primitive ID* must be used. The primitive ID is a number that identifies the sensor/actuator within the data structures returned by OVirtualRobotComm (see section 3.2). Because of that, a translation from the primitive locator to the primitive ID must be performed. Even that Sony also provides a conversion table on its documentation, it is recommended not to use it, since it may change for future robot models. Instead of that, an on-line conversion is recommended. Instead it can be used the OPENR::OpenPrimitive[3] static function in order to perform such conversion. Usually such conversion is done once during initialisation phase and the results stored in an array for later use.

Example:

```
void BallTrackingHead7::OpenPrimitives()
{
  OStatus result;
  for (int i = 0; i < NUM_JOINTS; i++)
  {

result = OPENR::OpenPrimitive(JOINT_LOCATOR[i], &jointID[i]);
    if (result != oSUCCESS)
    {

OSYSLOG1((osyslogERROR, "%s : %s %d","BallTrackingHead7::OpenPrimitives()
    }
  }

result = OPENR::OpenPrimitive(FBK_LOCATOR, &fbkID);
  if (result != oSUCCESS)
```

[3]This function also performs some initialisation job, so it must be called once in the program before accessing the sensor/actuator. This is usually done during the initialisation of the object.

```
  {

OSYSLOG1((osyslogERROR, "%s : %s %d","BallTrackingHead7::OpenPrimitives()
  }
}
```

3.2 Frames

Aibo's time is divided in **frames**. A frame is the unit of time and represents 8 ms. Information from sensors is retrieved by blocks of n frames (usually, 4 frames), which are contiguous in time. Commands to effectors are also sent in blocks of frames, it is, when you send a command to an effector, you must provide the commands for the next n frames of time.

3.3 Reading sensor values

In order to read a value from a sensor, it will be necessary to obtain messages sent by the OVirtualRobotComm through its gate named *Sensor*. Through that gate, OVirtualRobotComm sends a message of type OSensorFrameVectorData that contains all information related to the robot sensors. Therefore, the first step to catch that message would be to refer to the subject in the connect.cfg file using the following line:

```
OVirtualRobotComm.Sensor.OSensorFrameVectorData.S your_observer
```

An additional line must be also added to the stub.cfg file of the observer, indicating the name of the routine that will handle the message. The contents of the message is treated in the next section.

3.3.1 Description of the OSensorFrameVectorData message

The OSensorFrameVectorData is a data structure that accommodates all the required information to obtain Aibo's sensors states. It is formed by three main groups of data:

1. vectorInfo: It is another structure of data of type ODataVectorInfo. It contains the numData and maxNumData values. numData contains the number of sensors whose values are included in the structure (have been sensed).

2. OSensorFrameInfo: it is an array of data structures. The structure contains information that identifies the sensor

3. OSensorFrameData: it is another array of data structures of the same length as OSensorFrameInfo. Each OSensorFrameInfo, has its correspondent OSensorFrameData. While OSensorFrameInfo identifies the sensor that is being read, OSensorFrameData contains its values sensed. Values are specified in a structure to allow the allocation of different frames.

Each cell of OSensorFrameInfo has a correspondent cell in OSensorFrameData. It means that the information in frame *n* of OSensorFrameInfo is related to the info of frame *n* in OSensorFrameData. Those two have the information related to a given sensor during the last frames.

```
struct OSensorFrameVectorData
{
ODataVectorInfo vectorInfo;
struct OSensorFrameInfo[];
array OSensorFrameData [];
}
struct OSensorFrameInfo
{
ODataType type;
OPrimitiveID primitiveID;
longword frameNumber;
size_t numFrames;
}
struct OSensorFrameData
{
OSensorValue frame[];
}
```

vectorInfo

vectorInfo contains two values: numData and maxNumData. The two arrays OSensorFrameInfo and OSensorFrameData have an allocated memory size equal to maxNumData, but their actual size is that indicated by numData, it is, only numData sensors will have their values put in the OSensorFrameData array.

OSensorFrameInfo

Data from this array is available by using the GetInfo (int index) function. By using that function, the user obtains a OSensorFrameInfo structure that contains several information: type is a variable of class ODataType and contains the type of sensor been access. primitiveID is a variable of type OPrimitiveID and contains the primitive ID of the sensor been access. frameNumber is a longword containing a tag number that identifies the first frame on its associated OSensorFrameData cell. numFrames indicates the number of frames that are valid in the associated OSensorFrameData cell.

OSensorFrameData

Data from this array is available by using the GetData (int index) function. By using that function, the user obtains a group of *frames* containing OSensorValue structures. These structures are generic data ones for sensor values. This means that every sensor will send their values in a subclass of OSensorValue. Usually a cast to the correct subclass is performed when retrieving a sensor value.

3.3.2 Accessing a sensor value

The main steps to access a sensor value are the following:

1. A OSensorFrameVectorData is received from OVirtualRobotComm
2. Get the primitive ID corresponding to the sensor to access. A complete list of the desired sensor IDs is usually created during initialisation by the programmer (see section 3.1), so a retrieve of the list will do it.
3. Compare the obtained primitive ID with the ones that are in the primitiveID field of the OSensorFrameInfo array. Once matched, the index within that array provides the index within the OSensorFrameData array that contains the sensor value. You can store this index in a user array since it will not change during the execution of the OPEN-R program (see the example below).
4. Use the index obtained to access the sensor value in OSensorFrameData.
5. Do not forget to send an AR message after processing the data, to indicate that you can now receive another sensor message if available.

Example-1: creating the correspondence table between primitive ID and index within OSensorFrameData

```
void BallTrackingHead7::InitSensorIndex(OSensorFrameVectorData* sensorVec
{
  for (int i = 0; i < NUM_JOINTS; i++)
  {

for (int j = 0; j < sensorVec->vectorInfo.numData; j++)
    {

OSensorFrameInfo* info = sensorVec->GetInfo(j);
      if (info->primitiveID == jointID[i])
      {

sensoridx[i] = j; OSYSDEBUG(("[%2d] %s\n", sensoridx[i], JOINT_LOCATOR[i]
      }
    }
  }
}
```

Example-2: accessing a value

```
void BallTrackingHead7::NotifySensor(const ONotifyEvent& event)
{
  RCRegion* rgn = event.RCData(0);
  if (initSensorIndex == false)
  {
```

```
OSensorFrameVectorData* sv = (OSensorFrameVectorData*)rgn->Base();
     InitSensorIndex(sv);
     initSensorIndex = true;
  }

if (sensorRegions.size() == NUM_SENSOR_VECTOR)
  {
     sensorRegions.front()->RemoveReference();
     sensorRegions.pop_front();
  }
  rgn->AddReference();
  sensorRegions.push_back(rgn);
  observer[event.ObsIndex()]->AssertReady();
}
```

3.4 Sending commands to actuators

Sending commands to the Aibo actuators (these are motors and LEDs), requires to send a message of type OCommandVectorData to the OVirtualRobotComm via its incoming gate, named *Effector* (remember that the outgoing gate, from where values sensed come, was called *Sensor*). To be able to connect to the virtual object, the following line must be added to the connect.cfg file:

```
your_subject OVirtualRobotComm.Effector.OCommandVectorData.O
```

3.4.1 Description of the OCommandVectorData message

The OCommandVectorData message is a structure very similar to OSensorFrameVectorData (used to read sensor values). It also contains three members, which are a structure called *vectorInfo*, and two arrays called *OCommandInfo* and *OCommandData*. Each cell of OCommandInfo has a corresponding cell in the OCommandData array.

vectorInfo

It is a structure of type ODataVectorInfo. It contains two elements called numData and maxNumData. The two arrays OCommandInfo and OCommandData have an allocated memory size equal to maxNumData, but their actual size is that indicated by numData, it is, only numData actuators will be available in the OCommandData array.

OCommandInfo

This array is accessible by using the GetInfo (int index) function. Each cell in the OCommandInfo array has three main elements: *type* that describes the type of effector, *primitiveID* describing the effector primitive ID and *numFrames*, describing the number of frames passed in the command (since commands can be grouped to be sent for several frames).

OCommandData

It is a structure composed of an array of OCommandValue. OCommandValue is the general type for commands for effectors. Subclasses exists for each effector, and they must be checked at the Model Information documentation. To access the OCommandData structure use the GetData (int index) function. Each cell of OCommandData contains the commands for the next numFrames for a specific effector.

3.4.2 Sending a command to an effector

The list of steps to send a command to an effector :

1. Initialising the system

   Initialisation is required in order to move Aibo's joints or to send commands to the LEDs. Depending on the effector we want to act, initialisation will require more or less work. Initialisation of the joints can be performed in any OPEN-R object. Only one initialisation is required, usually done at the initialisation step of an object (DoInit procedure).

   - Get primitive IDs

   As happened with sensors, each effector has its own primitive ID. It is required to obtain the primitive ID of an effector in order to identify its index inside the OCommandInfo and OCommandata arrays. The steps to follow are the same as for sensors. First, the primitive locator is found on *Model Information* documentation. Then the OPENR::OpenPrimitive() function is used to make the conversion from locator to ID. This result usually is stored in an array for further use. See the example in section 3.2.2

   - Set joint gains (for joints)

   The joint motor control in Aibo is implemented by using a PID. In order to control those motors, their PID gains and shifts values must be set. The values to set the joints are specified by Sony on its *Model Information* documentation. Other PID values can be set but this is not recommended since they may damage the joints. Note that the PID values are different for each joint motor.

   The first step is to enable the joint gains. To do this, just call the OPENR::EnableJointGain() function. Then you can set the gains by calling the OPENR::SetJointGain() function with the appropriate values.

   Example:

```
void MTNAgent7::SetJointGain()
{
  OSYSDEBUG(("MTNAgent7::SetJointGain()\n"));

OPrimitiveID tilt1ID = moNetAgentManager->PrimitiveID(HEAD_TILT1);

OPrimitiveID panID = moNetAgentManager->PrimitiveID(HEAD_PAN);
```

```
OPrimitiveID tilt2ID = moNetAgentManager->PrimitiveID(HEAD_TILT2);
  OPENR::EnableJointGain(tilt1ID);

OPENR::SetJointGain(tilt1ID, TILT1_PGAIN, TILT1_IGAIN, TILT1_DGAIN, PS

OPENR::SetJointGain(panID, PAN_PGAIN, PAN_IGAIN, PAN_DGAIN, PSHIFT, IS
  OPENR::EnableJointGain(tilt2ID);

OPENR::SetJointGain(tilt2ID, TILT2_PGAIN, TILT2_IGAIN, TILT2_DGAIN, PS
  int base = RFLEG_J1;
  for (int i = 0; i < 4; i++)
  {

OPrimitiveID j1ID = moNetAgentManager->PrimitiveID(base + 3 * i);

OPrimitiveID j2ID = moNetAgentManager->PrimitiveID(base + 3 * i + 1);

OPrimitiveID j3ID = moNetAgentManager->PrimitiveID(base + 3 * i + 2);
    OPENR::EnableJointGain(j1ID);

OPENR::SetJointGain(j1ID, J1_PGAIN, J1_IGAIN, J1_DGAIN, PSHIFT, ISHIF
    OPENR::EnableJointGain(j2ID);

OPENR::SetJointGain(j2ID, J2_PGAIN, J2_IGAIN, J2_DGAIN, PSHIFT, ISHIF
    OPENR::EnableJointGain(j3ID);

OPENR::SetJointGain(j3ID, J3_PGAIN, J3_IGAIN, J3_DGAIN, PSHIFT, ISHIF
  }

OPrimitiveID tailtiltID = moNetAgentManager->PrimitiveID(TAIL_TILT);

OPrimitiveID tailpanID = moNetAgentManager->PrimitiveID(TAIL_PAN);
  OPENR::EnableJointGain(tailtiltID);

OPENR::SetJointGain(tailtiltID, TAIL_PGAIN, TAIL_IGAIN, TAIL_DGAIN, PS
  OPENR::EnableJointGain(tailpanID);

OPENR::SetJointGain(tailpanID, TAIL_PGAIN, TAIL_IGAIN, TAIL_DGAIN, PSH
}
```

- Calibrate the joints (for joints)

It happens sometimes that the position read by the sensors and the real position of the joint differs in some small quantities. For this reason, before moving the joints it is usually performed a calibration step. It consists of reading the actual

value of the joint and then setting the joint to the value sensed. Reading a joint value can be performed with the OPENR::GetJointValue() function. Then, the joint must be set to the read value by using a user defined function (not provided by OPENR).

Example:

```
MoNetStatus NeutralAgent::AdjustDiffJointValue(OVRSyncKey syncKey)
{
  OJointValue current[DRX900_NUM_JOINTS];

RCRegion* rgn = moNetAgentManager->FindFreeCommandRegion();

OCommandVectorData* cmdVecData = (OCommandVectorData*)rgn->Base();
  cmdVecData->vectorInfo.syncKey = syncKey;

for (int i = 0; i < DRX900_NUM_JOINTS; i++)
  {
    OJointValue current;

OPENR::GetJointValue(moNetAgentManager->PrimitiveID(i), &current);

SetJointValue(rgn, i, degrees(current.value/1000000.0), degrees(curren
  }

moNetAgentManager->Effector()->SetData(rgn);

moNetAgentManager->Effector()->NotifyObservers();
  return monetCOMPLETION;
}
```

2. Select a free shared memory region

   Commands for effectors are not directly send. Instead of that, a buffer method is implemented in order to avoid two possible problems: first, messages maximum size may be smaller than the message being actually sent. Then, instead of sending a command structure (of type OCommandVectorData), objects send a pointer to a command structure situated in the shared memory[4]. Second, by using this method, a group of buffers can be set. Buffers would act as a place where commands are stored for retrieval when the OVirtualRobotComm is ready. By using those buffers it is possible to send commands to the OVirtualRobotComm without paying attention if its is ready or not. commands are then just stored in the buffers waiting for the virtual object. This method brings smoothness and

[4]The shared memory is a place of Aibo's memory where all objects can write and read. Thus, it is used to interchange information between objects

higher reactivity to the robot since every command is processed as quick as possible between gaps in the middle. Usually, two buffers are allocated. Buffers are created by the programmer.

To access the shared memory region, OPEN-R provides the *RCRegion* class. To send commands to joints OPEN-R provides a function (*OPENR::NewCommandVectorData()*) to allocate the memory and to hold a reference counter. This counter is used by the system to avoid region overwriting. The OPEN-R command creates a *OCommandVectorData* in shared memory and an ID for that memory. It needs three arguments:

size_t numCommands, that contains the number of cells in the OCommandData array, one for each actuator wanted to command

MemoryRegionID memID*, that will have the ID of the memory allocated

*OCommandVectorData** *baseAddr*, that is the pointer to the memory region

Once called this function, then the RCRegion class must be instantiated. The class constructor is *RCRegion (MemoryRegionID memID, size_r offset, void* baseAddr, size_t size)* where *memID* is the *memRegionID* of the *ODataVectorInfo*, *offset* is the offset of *ODataVectorInfo*, *baseAddr* is the pointer returned by *OPENR::NewCommandVectorData()* and *size* is the total size of *ODataVectorInfo*.

Example:

```
void BallTrackingHead7::NewCommandVectorData()
{
  OStatus result;
  MemoryRegionID cmdVecDataID;
  OCommandVectorData* cmdVecData;
  OCommandInfo* info;

for (int i = 0; i < NUM_COMMAND_VECTOR; i++)
  {

result = OPENR::NewCommandVectorData(NUM_JOINTS, &cmdVecDataID, &cmdVe
    if (result != oSUCCESS)
    {

OSYSLOG1((osyslogERROR, "%s : %s %d", "BallTrackingHead7::NewCommandVe
    }

region[i] = new RCRegion(cmdVecData->vectorInfo.memRegionID, cmdVecDat
    cmdVecData->SetNumData(NUM_JOINTS);
    for (int j = 0; j < NUM_JOINTS; j++)
    {
```

```
info = cmdVecData->GetInfo(j); info->Set(odataJOINT_COMMAND2, jointID
     }
  }
}
```

3. Set the effector value

   Once the memory has been allocated by creating the RCRegion, then joint values can be sent. The process begins by checking that no other object is reading the shared memory before writing to it. To do that, the RCRegion class has a function called NumberOfReference() that returns the number of objects pointing to that memory (this control is established when calling the *OPENR::NewCommandVectorData()* explained above). The function will return the number of objects pointing to that memory including the present object (so a value of 1 is correct to start writing).

   Once it is sure to write to the region, a sequence of frame commands must be created. In order to create a linear movement from the current joint position to the desired new one, a set of mid-steps must be created. This process brings smoothness in the robot movement and prevents damages for too high velocity movements. The way of creating this mid-steps must be decided by the programmer and implemented by it. There is no help function provided for it. This allows the programmer to design his own type of movements (linear, exponential,etc.). All the frames generated will then fill the frames in the *OCommandData*.

   Example: In the BalltrackingHead7 example, the creation of the frame values is performed by implementing the SetJointValue() function.

```
void BallTrackingHead7::SetJointValue(RCRegion* rgn, int idx, double s
{

OCommandVectorData* cmdVecData = (OCommandVectorData*)rgn->Base();

OCommandInfo* info = cmdVecData->GetInfo(idx);

info->Set(odataJOINT_COMMAND2, jointID[idx], NUM_FRAMES);

OCommandData* data = cmdVecData->GetData(idx);

OJointCommandValue2* jval = (OJointCommandValue2*)data->value;
  double delta = end - start;
  for (int i = 0; i < NUM_FRAMES; i++)
  {

double dval = start + (delta * i) / (double)NUM_FRAMES; jval[i].value
  }
}
```