GAIPS/INESC-ID

MyPleo System Description

(version 1.00) - August 8, 2011

Paulo F. Gomes

Contents

Introduction

This document describes the artificial pet prototype used in the *Pleo* research scenario of the European project *LIREC* (LIving with Robots and IntEractive Companions). I will refer to the artificial pet as "prototype" or as "MyPleo". *MyPleo* has two embodiments: a robotic one, consisting of a modified *Pleo*[1] robot; and a virtual one, consisting of an *Android*[2] application for an *HTC Desire*[3][4]. These two embodiments will be referred to as *PhyPleo* and *ViPleo* respectively (see Figure 1). *MyPleo* can switch between embodiments (*PhyPleo and ViPleo*), only being active in one at a time.

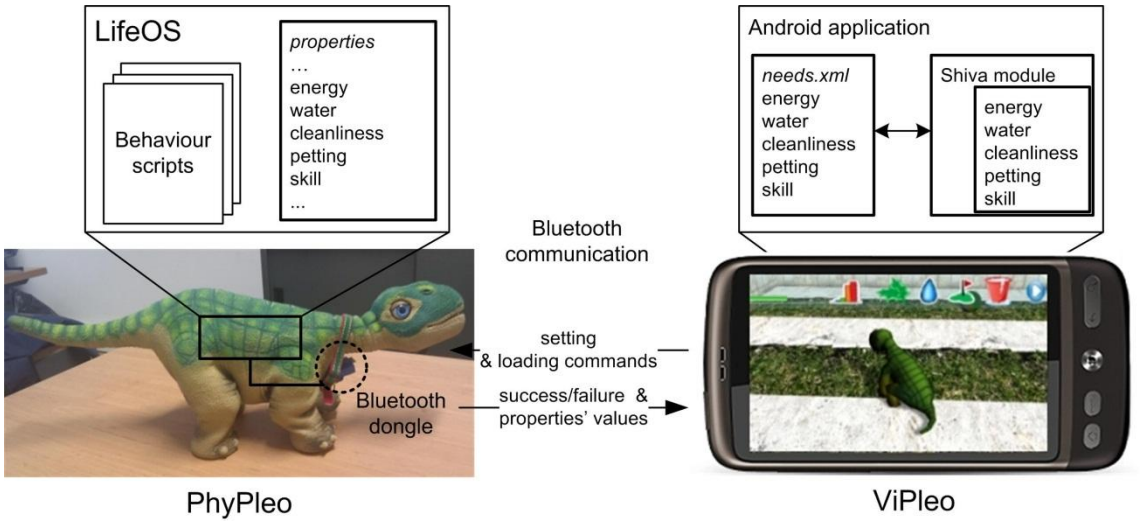

Figure 1: MyPleo architecture

The behaviour of both *PhyPleo* and *ViPleo* is driven by the following needs:

- *Preserving needs*: energy, water (thirst) and cleanliness;
- *Affiliation need*: petting (need for affection);
- *Competence need*: skill;

Each need has a corresponding numeric value that ranges from 0% (need completely unfulfilled) to 100% (need completely fulfilled). For instance, an energy value close to 0% would represent that Pleo was starving. In both embodiments the energy, water, cleanliness and petting values decay with time. The decay rate is the same in *PhyPleo* and *ViPleo*. Moreover, the initial values for all needs are also the same in both embodiments.

PhyPleo's behaviour was defined in Pawn script using the SDK for Pleo 1.1.1. *ViPleo*, on the other hand, is an Android application written in Java that has a module using the ShiVa3d graphical engine[5]. This module is the interactive part of the application and is internally scripted in *Lua*. The android application is responsible for communicating with *PhyPleo*, loading the needs values from an xml file in which they are stored persistently, and starting up the Shiva module. The remaining document will describe in detail the three mentioned

[1] Copyright Innvo Labs Corporation.
[2] Copyright Google Inc.
[3] Copyright HTC Corporation.
[4] The device used will also be referred to simply as "Smartphone".
[5] Copyright 2010 Stonetrip.

elements: the *ShiVa Module*, the *Android Application* and *PhyPleo*. Besides these three sections, we will end by presenting a summary of the migration process in the section *Migration*. This last section will also include the description of the automatic migration feature: *MyPleo* can automatically migrate from one embodiment to another after a defined time interval.

This manual was originally created to enable *LIREC* developers/researchers. However, it may also serve other individuals that want to get better acquainted with the prototype and how it works. Although, only *LIREC* developers/researchers will have access to the complete *MyPleo* bundle with all the prototype's resources, most of the resources can be found in http://trac.lirec.org/browser/scenarios/MyFriend/MyPleo/. Furthermore, all the code is there and has been licensed under *GNU GENERAL PUBLIC LICENSE - Version 3*, 29 June 2007 (http://www.gnu.org/copyleft/gpl.html).

Throughout the text there will be references to other documents, namely concerning used tools. For a prototype such as *MyPleo*, it would extend the document considerably to include the documentation for all the used software. Readers not familiar with the frameworks and tools should consider taking a look at such sources, especially if they find themselves troubled with the terminology.

Still concerning outside references, the mentioned folders assume that you are already in the bundle folder (https://svn.lirec.eu/scenarios/MyFriend/MyPleo/miniBundle/), unless they start with *workspace*, in which case they refer to this folder (https://svn.lirec.eu/scenarios/MyFriend/MyPleo/workspace/).

ShiVa Module

Is a virtual pet game in which the player can take care of *MyPleo* (e.g. clean him), play with him (obstacle course) and monitor its need values. It was developed using the *ShiVa 3D Editor*. The reader is assumed to have basic knowledge of the *ShiVa 3D Game Engine*. If that is not the case I suggest going through some tutorials such as:

- http://www.stonetrip.com/developer/162-first-application
- http://www.youtube.com/watch?v=UeiSlPdUSJ4&feature=player_embedded
- http://www.stonetrip.com/developer/609-shiva3d-quick-start-guide

More resources can be found at http://www.stonetrip.com/developer/. I recommend bookmarking the script reference: http://www.stonetrip.com/developer/doc/api/introduction. Note that many of the variable names do not follow the rules suggested in the documentation.

ShiVa games can communicate to the outside environment via html requests and by xml access. In this prototype xml access is used to load need values at start up, store need values when exiting[6], and load the configurations for the automatic migration. Note that the xml files in *workspace**viPleoShivaModuleOutput**files*\ are only used when the project is being debugged. The files that will actually be used when the prototype is running are the ones in *workspace**Pleo**assets*\.

The Shiva module has six critical elements:

- *MainAI* (AI Model);
- *MyPleoAI* (AI Model);
- *MyPleoAI_Game* (AI Model);
- *Menu_Accoes* (HUD Template);
- *Camera* (AI Model): responsible for aligning the camera with *MyPleo*;
- *HUD_Game_Menu* (HUD Template): is only shown when the obstacle course is being taken and generates the events necessary for this mini-game;

We continue by describing the first four in detail.

MainAI

MainAI is responsible for:

- Dealing with all events sent from the interfaces. It redirects most of the events to *MyPleoAI*;
- Storing need values to *files**needs.xml*;
- Loading the automatic migration configuration from *files**configuration.xml*;
- Keeping track of a timer for the automatic migration: At each frame, if the automatic migration is active, it verifies if it is time to migrate;
- Updating the needs and overall need satisfaction value (named attachment estimate) progress bars when they are updated by *MyPleoAI*;

[6] During gameplay need values are typically stored in environment variables to facilitate loading and storing values from/to the xml.

When the *ShiVa module* starts, *MainAI* performs three main actions:

- If the Operative System is Android, rotates the view to match the device's screen orientation;
- The main scene and interface are loaded;
- The timer for the automatic migration is started, or not, according to the *files\configuration.xml*. The file is first loaded to a local xml variable, and then read.

When the *ShiVa module* is about to exit, needs are stored to the *files\needs.xml* according to the following steps:

1. A temporary empty xml is created in the local variables;
2. Need values are loaded from the environment variables to the temporary xml;
3. The temporary xml is written to *files\needs.xml*;

MyPleoAI

MyPleoAI is responsible for defining *MyPleo's* general behaviour on the playground and for updating its needs. The configuration of how actions affect needs, and of how needs decay, can be done through the *init* values of its variables. The variable naming style is the following:

- *nNeed<need-name>Weight*: weight of need *<need-name>* on the overall need satisfaction value. It must be a value between 0 and 1, and the sum of all weights must be 1.
- *nNeed<need-name>Minimum*: if the need *<need-name>* goes bellow this value, the need is activated;
- *nNeed<need-name>Initial*: default initial value for need *<need-name>*. It is only used if for some reason the program was unable to load the needs xml;
- *nNeed<need-name>Decay*: decrease of need *<need-name>* each time *<need-name>* needs are updated;
- *nNeed<need-name>Increase<action-name>*: increase of need *<need-name>* when action *<action-name>* is performed;
- *nNeed<need-name>Decrease<action-name>*: decrease of need *<need-name>* when action *<action-name>* is performed;

The defined values are the same, or equivalent, to those defined for PhyPleo. Need values should be accessed as much as possible through the getter and setter functions. Need values are loaded from xml, and if an error occurs during loading, the *nNeed<need-name>Initial* values are loaded instead.

MyPleoAI updates the overall need satisfaction value and needs values. The time interval between updates is defined by *nNeedsTimerInterval init* value and is the same as the update frequency used in PhyPleo. The update consists of the following set of steps:

1. Decreases need values according to respective decays (*nNeed<need-name>Decay*);
2. Updates the overall need satisfaction value by performing a weighted sum of all needs;
3. Notifies *MainAI* that needs were updated;

4. Verifies if any need is active (its value is below the corresponding *minimum* value). Activates behaviour accordingly. Currently this verification is only done for the *energy* need;
5. Reset the a timer so that update function will be called again;

MyPleo can be in a finite number of states (e.g. Eating – eating a patch of leaves). The current state is defined by direct interface interactions and by needs reaching critical values (currently only energy need). Instead of describing the states explicitly, its overall behaviour in the playground is described according to performed user actions:

- Placing a patch of leaves in the virtual playground will cause *MyPleo* to move towards the patch and then eat it. After eating it (Figure 2.a), its energy value is increased by *nNeedEnergyIncreaseFood*. Additionally, it will poop after a while (Figure 2.b), which will decrease its cleanliness value by *nNeedCleanlinessDecreasePoop*;
- Placing a water bowl in the virtual playground will cause Pleo to move towards the bowl and drink it (Figure 2.d), which will increase its water value by *nNeedWaterIncreaseBowl*;
- Touching the screen in the area in which Pleo is shown causes it to raise its neck (Figure 2.c). Additionally the petting value is increased by *nNeedPettingIncreasePet*.
- Selecting the washing option causes Pleo to also raise its neck, removes any poop that might be in the playground, and increases the cleanliness value by *nNeedCleanlinessIncreaseCleanPoop*;
- If its energy value is lower than *nNeedEnergyMinimum*, it sits down and cries (Figure 2.f);
- Pleo walks around the playground when the user does not interact with it;

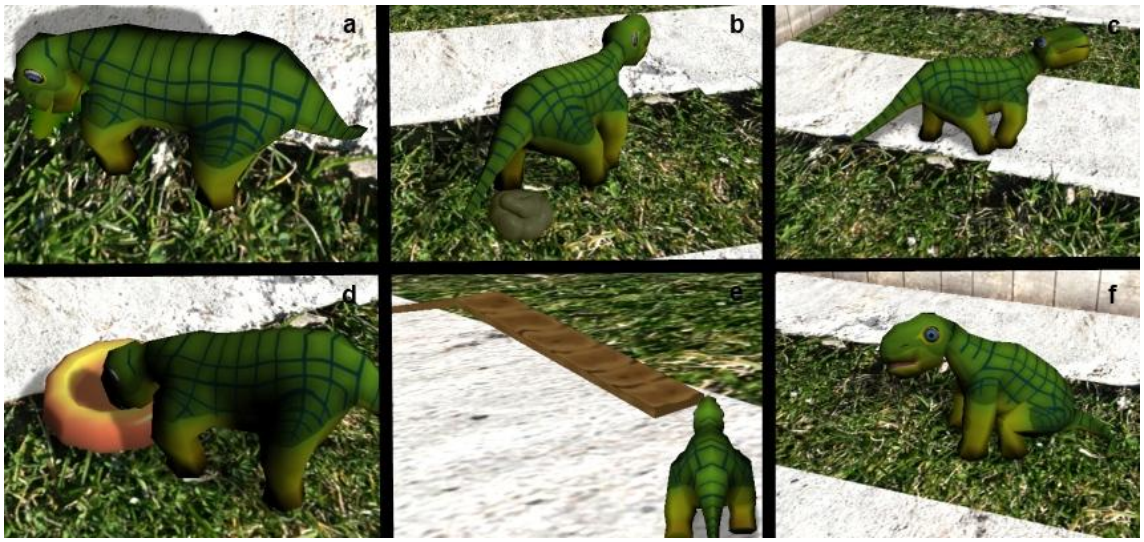

Figure 2: ViPleo's Behaviours - eating (a), pooping (b), being petted (c), drinking water (d), going through an obstacle course (e), sitting and crying (f).

MyPleoAI Game

MyPleoAI Game is responsible for controlling *MyPleo's* behaviour in the obstacle course. The objective of this mini-game is to take MyPleo through the obstacle course (Figure 2.e). The obstacle course has a sequence of obstacles. To cross an obstacle, the player must hit a button when it is red (*HUD Game Menu*). The button oscillates (according to a timer defined in *HUD*

Game Menu) between 5 different positions, but only becomes red when it is at the centre. *MyPleo's* skill value is increased each time an obstacle course is completed[7].

Figure 3: Obstacle course interface. The button oscillates between inactive positions (see left) and the active position at the centre (see right).

Menu Accoes

Menu Accoes has 3 main elements:

- User actions and need stats (Figure 4);
- An invisible button at the screen centre used to detect petting;
- A *Play There* button used to manually migrate;

Figure 4: User actions and need stats.

There are many buttons in the *Menu Accoes* that are not being shown. Some of which have equal or similar functionality to the buttons being shown. The current set of available actions was defined for the *first evaluation* with the *prototype*.

Android Application

Android Application is responsible for:

- The communication between the cell-phone and *PhyPleo*;
- Updating the *needs.xml* according to values received from *PhyPleo*;
- Editing the *configuration.xml* when automatic migration from *ViPleo* to *PhyPleo* is needed;
- Handling the interface;

It was developed for *Android 2.2*. Analogously to the *ShiVa module*, this section assumes that the reader has a certain degree of knowledge of the *Android SDK*. If that is not the case, I suggest going through some links such as:

- http://developer.android.com/resources/tutorials/hello-world.html

[7] If the user migrates *MyPleo* before pressing the *Sair* button the skill value is not increased.

- http://developer.android.com/guide/topics/fundamentals/activities.html
- http://developer.android.com/guide/topics/wireless/bluetooth.html
- http://developer.android.com/guide/topics/fundamentals/services.html

More resources can be found at http://developer.android.com/guide/index.html.

Android Application has the following classes:

- *PleoMainActivity*: Android activity responsible for launching the connection service with *PhyPleo* and displaying the migration options available;
- *PleoConnectionService*: Android service responsible for the communication with the *PhyPleo*. When it receives a request, and is able to connect to *PhyPleo*, it launches a thread responsible for either setting properties or loading them;
- *PleoMonitorRunnable*: Runnable used to set and load properties from the robot's monitor interface. It loads to, and from, the need's xml. It verifies if commands have been correctly executed, and if not, tries to overcome the detected problem;
- *MyPleo*: Android activity responsible for launching the *ShiVa module* and for showing the splash screen. The splash screen is shown for *SPLASH_DURATION_MILI_SECONDS* milliseconds (hardcoded value);
- *MyPleoNeeds and MyPleoNeedsVerifiable*: containers for needs;
- *MyPleoNeedsXMLUpdater*, *MyPleoNeedsParserHandler*, *MyPleoNeedsParserException*, *ConfigurationXMLUpdater*, *ConfigurationXMLParserHandler* and *ConfigurationXMLParserException*: are used for different types of xml parsing/editing;
- *S3DEngine* and *S3DSurfaceView*: were initially generated by the ShiVa Authoring tool and wrap the ShiVa module. The *S3SSurfaceView* was modified so that when the ShiVa module pauses, there is an attempt to migrate to *PhyPleo*;

The *Android Application* eclipse project is placed in the *workspace**Pleo*. In the *Pleo* project, the non-code files worth keeping in mind are:

- *AndroidManifest.xml*: besides being critical for the Android SDK, this file would need to be changed (the intent filters) in order for *PleoConnectionService* to deal with new types of requests;
- *configuration.xml* and *needs.xml* in *assets*: are the files that will actually be used for configuration and initial need values when the application is deployed, not the ones in the ShiVa project (only used for testing purposes);
- *S3DMain.stk* in *assets*: contains all the functionality of the *ShiVa module*;
- *connection.xml* in *res/layout*: layout for the migration interface;
- *strings.xml:* text for the interface buttons and some warnings. Some of these warnings were created so that a Guide would understand what they meant, but a user/tester would not;

In order for the *Android Application* to work correctly, the device in which it is deployed must not be mounted as a disk drive and must have a SD Card. The description of the Android Application will continue with further details concerning four main classes (the first four in the previous class list).

PleoMainActivity

PleoMainActivity is the main activity of the *Application* and it is the first to be launched. At launch, it performs the following steps:

1. Creates the interface buttons;
2. Extracts the *needs.xml* and *configuration.xml* to a local folder. Note that deployment is only done through an *apk* file. In order for the *ShiVa module* to access them as well, they need to be extracted from the *apk*;
3. Setup a broadcast receiver that will deal with messages sent from the *PleoConnectionService*. These messages inform the activity of the services' progress or failure;
4. If the Bluetooth is not turned on, ask the user to do so;
5. Start the *PleoConnectionService*.

In the remaining text, an embodiment is said to be active if *MyPleo's* behaviour is being displayed in that embodiment. *PleoMainActivity* can be in one of four states:

- *BOTH_INACTIVE*: it is the initial state in which both embodiments are inactive;
- *VIPLEO*: the *ViPleo* embodiment is active, or being activated (*ShiVa module* is being started);
- *VIPLEOTOPHYPLEO*: *ViPleo* is inactive and *PhyPleo* is being activated (Scripts are being loaded);
- *PHYPLEO*: *PhyPleo* is active and *ViPleo* is inactive;

There is an additional fifth state that currently is not being used: *PHYPLEOTOVIPLEO*. This state could be used to represent situations in which *PhyPleo* has not yet stopped moving, but the ShiVa module has already been launched. In theory, both *ViPleo* and *PhyPleo* could be active at the same time. In practice, as the *ShiVa module* takes some time to start, it rarely happens. This time could be reduced if the splash screen would be shown for less time, or removed entirely (see *MyPleo*).

The user can perform two main actions through the activity: activating ViPleo or activating *PhyPleo*. Activating *PhyPleo* is only enabled if the Activity has received a message from the *PleoConnectionService* saying that it was able to connect to *PhyPleo*, and if *PhyPleo* is inactive. Activating *PhyPleo* is permanently disabled if the Activity receives a reading error from *PleoConnectionService*. On the other hand, activating *ViPleo* is only available when *ViPleo* is inactive. *PleoMainActivity* keeps track of the embodiment's activity status through the state mentioned above.

PleoConnectionService

When this service tries to connect to *PhyPleo*, it can do so in two ways:

- *strong connect*: if it fails to connect because the system was unable to create a communication socket (most likely because *PhyPleo* is turned off), it waits *CONNECTION_CREATION_INTERVAL* milliseconds and tries again. It will keep on trying to connect if not successful, always waiting *CONNECTION_CREATION_INTERVAL* milliseconds between tries. As *PleoConnectionService* runs on the main thread, a

strong connect causes the whole application to wait until the connection is established.

- *weak connect*: if it fails to connect it immediately gives up;

In both cases it will inform *PleoMainActivity* if it is successful through a broadcast.

When the service is launched by *PleoMainActivity* it performs the following two steps:

1. Loads the *configuration.xml* extracting *PhyPleo*'s dongle mac address. Note that, if one is to use a different *PhyPleo* robot this address must be changed in the xml. The two possible addresses, corresponding to the two different *PhyPleo*'s, are written in a comment of *configuration.xml*;
2. Performs a *weak connect* to *PhyPleo*. Only if the *weak connect* is successful, will *PleoMainActivity* enable the activation of *PhyPleo* (see *PleoMainActivity*). Otherwise that functionality will remain inactive, but the application will continue running (which would not happen if a *strong connect* had been used);

The Service treats the following requests:

- *Unload Pleo Behavior*: activates *PhyPleo's* empty behaviour and loads the needs from *PhyPleo*;
- *Load Pleo Behavior*: announces *MainActivity* that the companion is migrating from *ViPleo* to *PhyPleo* though a broadcast, sets *PhyPleo's* needs to the values in the *needs.xml*, and deactivates *PhyPleo*'s empty behaviour. Performs a *strong connect* if a new connection is needed. Currently, this request is not being used;
- *Load Pleo Behavior try*: equal to the previous request, but uses a *weak connect* if a new connection is needed;
- *Load Pleo Behavior simple*: deactivates *PhyPleo*'s empty behaviour. Used when the first embodiment chosen is *PhyPleo*. In this case the initial need values will be the ones defined in *PhyPleo*;

For all requests, the service first tries to connect to *PhyPleo* if not yet connected. It launches a thread per request. Each thread corresponds to an instance of a *PleoMonitorRunnable* and accomplishes its task by setting PhyPleo' properties. Currently, the numbers that identify the different properties are hardcoded. If the needs are removed or added on *PhyPleo*, these numbers might need to change, as they are automatically generated during the *PhyPleo* deployment.

PleoMonitorRunnable

Each Runnable have a set of commands (text lines) that were previously defined by the *PleoConnectionService*, and must be sent via Bluetooth to *PhyPleo*.

Commands have three types:

- *property setting*: used to update *PhyPleo's* needs and activate/deactivate PhyPleo's empty behaviour;
- *property loading*: used to load *PhyPleo's* needs to the *needs.xml*. The result of this type of command is property values being outputted as a series of lines;
- *clear*;

The clear command deals with a Bluetooth communication problem. Commands sent to the robot's monitor interface (see sub-section *Monitor* in *PhyPleo* section) have a tendency to be received with noise. This phenomenon is probably caused by the serial connection being prone to electro-static discharge, as mentioned in the monitor's documentation. Besides causing commands not to be correctly executed, the received noise has a second effect: if the monitor receives too much noise, it outputs a warning message and shuts down, effectively stopping all communication via Bluetooth. Moreover, the monitor can only be reused after the robot is turned off and on. Although not documented, the robot appears to have a counter for unrecognized characters, that when reaching the value 13, triggers the monitor to shut down. Also not documented as such, the command "clear" appears to reset this counter.

When *PleoMonitorRunnable* is run, it performs the following main steps:

1. Create an empty need container;
2. Create input and output sockets for the Bluetooth connection;
3. Try to execute commands one by one;
4. Update the needs.xml according to the need container if it was filled completely with all the necessary values (*successful loading*);

Commands are grouped in categories. A command is assigned to a category if it matches a category pattern. Category patterns are defined as regular expressions. Patterns are also defined for expected output from the monitor connection. These are named as *expected monitor line patterns*. Each command category has a corresponding *expected monitor line pattern*. If after a command from a category sent, the corresponding *expected monitor line pattern* is detected in the output, then the command is considered to have been successfully executed. *Expected monitor line patterns* are used to detect successful execution.

Additionally, there are patterns to detect individual property values (*property loading patterns*) when a *property loading* command is used. Currently, the numbers that identify the different needs are hardcoded in the *MyPleoNeeds* class. As mentioned before, if the needs are removed or added in *PhyPleo*, these numbers might need to change, as they are automatically generated during the *PhyPleo* deployment.

Finally, there are patterns for failure detection:

- *Monitor Off Pattern* : detects if the monitor has shut down due to noise;
- *Shut Down Pattern*: detects if *PhyPleo's* behaviour has been shut down due to low battery;
- *Property Noise Pattern*: detects if property setting or loading appeared garbled in the output. Effectively, what it does is detect output lines that have to do with properties, but do not match any of the expected monitor line patterns;

Command execution can be defined as the following algorithm:

1. Write command to the output socket;
2. Read line *l* from the input socket;
3. If *l* matches a *property loading pattern*, set the value of the corresponding need on the need container to the read value;
4. If *l* matches the command's category *expected monitor line pattern*, return *SUCCESS*;
5. If *l* matches the *Monitor Off Pattern* or *Shut Down Pattern*, return *FAILURE*;
6. If *l* matches the *Property Noise Pattern*, wait RESEND_INTERVAL_MILLISECONDS milliseconds and jump to step 1;
7. Jump to step 2;

PhyPleo

PhyPleo is a modified *Pleo* robot whose behaviour was redefined and in which a Bluetooth dongle was installed. It was not possible to extent the robot's original behaviour with additional functionality because we did not have the original code, nor the possibility of linking a compiled version of it to our code. The behaviour was based on one of the example behaviours supplied together with the robot's SDK. The Bluetooth dongle was connected to the robot's serial port (UART) thus enabling wireless communication with the robot's monitor interface. In this section we describe how *Needs* are maintained and updated, how *Behaviour* is affected by the needs and sensor data, and some notes concerning the robot's *Monitor* interface.

Before proceeding, I strongly recommend reading *phypleo**documentation**Pleo Programmers Guide.pdf*.

Needs

Behaviour is driven by needs with initial values equal to the ones in *ViPleo* (hardcoded in *main.p*). These needs are stored as properties in a reserved portion of memory. They are updated by leak integrators, user actions and Bluetooth communication. Leaky integrators decay property values over time at the same rate as in *ViPleo* (hardcoded in *main.p*).

User actions are detected through the robot's sensors. *PhyPleo* responds to the following actions: being petted, having a leaf in front of him and having a leaf in his mouth (see *sensors.p*). Being petted immediately increases the petting need value by the same value that in *ViPleo* (*nNeedPettingIncreasePet*).

Behaviour

The described behaviour is defined in the folder *workspace**phypleo**needs_behavior*. It was an adaptation of the example behaviour of the SDK (placed in *phypleo**examples**drive_example* for completeness*)*.

Needs determine behaviour except if the *mode property* is set to 1. In this case the *empty behaviour* is selected. When selected, *PhyPleo* appears to go to sleep and the leaky integrators are deactivated. When deselected (*mode property* is set to 0), the leaky integrators are reactivated and *PhyPleo* appears to wake up (this last step is automatic as no specific call to a wake up animation is performed). The *mode property* is used to activate and deactivate *PhyPleo* via Bluetooth. Note that this property is initially set to 1, hence *PhyPleo* always starts inactive, and performs a *going to sleep* animation when turned on.

PhyPleo has two main drives that take in account the need values: social and hunger. The social drive is active by default, but if the energy value drops below a critical value (hardcoded in *hunger.p* and the same as *ViPleo's nNeedEnergyMinimum*), the hunger drive becomes active. Only one drive is active at a time.

When the social drive is active, *PhyPleo* will perform two different behaviours depending on the petting need value: if the need value is bellow a critical value (hardcoded in *social.p*) *PhyPleo* will whine; if not, it will bark and wag its tail.

On the other hand, when the hunger drive is active only one behaviour is performed: eat. The eat behaviour consists of the following steps:

1. Search for food: *PhyPleo* slowly walks forward with is head bent downwards, sniffing the ground. Stops when leaf is detected;
2. Bite: first PhyPleo raises its neck and opens its mouth. Afterwards it bites and a crunching sound is heard. Stops a while after detecting a leaf in its mouth;

After the behaviour has finished (leaf was detected) it increases the energy need value by the same value that in *ViPleo* (*nNeedEnergyIncreaseFood*).

Monitor

The monitor interface, described in detail in *phypleo**documentation**Pleo Monitor.pdf*, is used to set properties and load properties via Bluetooth. Some of the most important commands for this project are:

- *joint neutral*: set all joints to neutral positions;
- *log disable all*: disable all logging;
- *stats power*: check current power status;
- *clear*: clear screen and apparently noise buffer (see *PleoMonitorRunnable*);
- *motion play <motion-ID>*: play motion with ID <motion-ID>;
- *motion show*: show all available motions;
- *property show*: show all properties and property values;
- *property set <property-ID> <value>*: set property with ID <property-ID> to <value>;

The Monitor can accessed through the serial port (used to connect the Bluetooth dongle) and through the USB port. To do so, you just need to turn on PhyPleo, and connect to the Bluetooth dongle using putty. For detailed instructions consult *MyPleo Installation and Deployment*.

If for some reason you are unable to use the monitor via Bluetooth, you might want to use the USB port. Details for doing so can be consulted in *phypleo**documentation**Pleo Monitor.pdf*. Note however that the drivers available are for Windows XP 32-bit. If you are not using such an operative system, you might be forced to install a virtual machine to use it.

Migration

The migration between embodiments is conceptually performed by sending need values via Bluetooth. Typically, the migration process is triggered in the hand-held device. For it to take place, the robot must be turned on and the Android application running.

We present the steps of the *PhyPleo* to *ViPleo* migration:

- Need values in *PhyPleo* properties are requested via Bluetooth by the *Android applicatio*n;
- These values are sent to the *Android application* via Bluetooth;
- In *Android application* the values in *needs.xml* are overwritten by the received values;
- The *Android application* sends a command to *PhyPleo* so that the "empty behaviour" is loaded;
- *PhyPleo* performs an animation of going to sleep;
- The *Shiva module* is started;
- Need values are loaded from the xml to the *Shiva module*;

The migration from *ViPleo* to *PhyPleo* has the following steps:

- Local need values in the Shiva module are stored in the *needs.xml* file;
- The *Shiva module* exits;
- The *Android application* loads the need values from the xml;
- Setting commands for the need properties are sent via Bluetooth to *PhyPleo*;
- The *Android application* sends a command to *PhyPleo* so that the "empty behaviour" is unloaded and the normal behaviour of the pet is performed;
- *PhyPleo* performs an animation of waking up;

Alternatively MyPleo can automatically migrate from one embodiment to another after a defined time interval. Automatic migration was used in the *first evaluation*. Due to the context of this evaluation, after automatic migration has been set, all of the *Android Application*'s buttons are hidden. Therefore most of the control over migration is lost. This functionality is currently not accessible through the interface (buttons have been hidden). Furthermore, as the prototype has since evolved to be easier to demonstrate outside the context of the evaluation, little effort has been put in maintaining this automatic migration functional.

Automatic migration is defined in two different places, depending on its direction:

- *ViPleo* to *PhyPleo*: automatic migration is configured through the *configuration.xml*. There, one can activate/deactivate it, and also set the time *ViPleo* should remain active before migration occurs. Note that some of the functionality currently not shown on the interface would override the activation/deactivation value.
- *PhyPleo* to *ViPleo*: the time interval value is hardcoded, and activation is done by launching a thread.

Acknowledgements

This work is partially supported by the European Community (EC) and was funded by the EU FP7 ICT-215554 project LIREC (LIving with Robots and IntEractive Companions), and FCT (INESC-ID multiannual funding) through the PIDDAC Program funds. The authors are solely responsible for the content of this publication. It does not represent the opinion of the EC, and the EC is not responsible for any use that might be made of data appearing therein.