JaCa-Android By Example

version: 2.0

Main author: asanti

Creation date: 20110331
Last Changes date: 20110411

DEIS, Università di Bologna, Italy

Contents

Chapter 1

Introduction

In the following we describe some main features of JaCa-Android making a sequence of simple tests, example and performance tests focussing on the different functionalities provided by the middleware. This document is meant to be read by people already familiar with the JaCa programming model—the programming model upon which JaCa-Android is based. If you are an inexperienced reader please consider to acquire a good background knowledge on both CArtAgO and *Jason* and in the JaCa programming model (raw speaking the synergistic integration of these two agent-based technologies). The quickest way for getting started with the JaCa programming model is the CArtAgO by example tutorial that can be found on the CArtAgO website[1]. For more exhaustive informations the following references can be considered:

- The *Jason* book[2], the reference *Jason* paper [1] and the *Jason* website[2].
- Some papers about CArtAgO[4, 3, 5] and the CArtAgO website[3].

All the examples that we describe in this document are packed together in a simple Android application (the JaCa-Android sample suite) that can be used for testing the examples in your device. The `.apk` containing the examples (named JaCa-Android-Samples.apk) can be found inside the `Apk` folder placed in the root of the standard JaCa-Android distribution. The folder `JaCa-Android-Samples` contains instead the test suite sources. This

[1]`http://cartago.sourceforge.net/?page_id=47`
[2]`http://jason.sourceforge.net/`
[3]`http://www.cartago.sourceforge.net`

folder is also a JaCa-Android Eclipse project that can be directly imported into your workspace. For running the samples in your device is required that you first install and configure the JaCa-Android middleware, instructions for doing this can be found in the getting started guide[4].

[4]`http://jaca-android.sourceforge.net/?page_id=257`

Chapter 2

Tests

In this chapter we describe a set of examples useful to highlight the basic use of the main functionalities provided by the JaCa-Android middleware. You can find the source code of this samples inside the `JaCa-Andorid-Samples/src/tests` folder. For running the samples just install the JaCa-Android-Samples.apk, launch the application, choose JaCa-Android Tests and the select the sample you want to launch.

All the samples contained in the `tests` folder have been realised using the same structure for the GUI, showed in Figure 2.1, where you can see: *(i)* a tab containing the specific GUI (Figure 2.1 (a)) of the application (if present), *(ii)* a tab containing the source code of the agent used in that example (Figure 2.1 (b)), *(iii)* a tab containing the mas2j configuration file (Figure 2.1 (c)), and on the bottom *(iv)* a output console in which is possible to see output and error messages. In most of the samples the agents used have an initialisation plan (the `init` plan), not reported in this document for simplicity, exploited for joining the MiddlewareServices shared workspace. For fully understand how to join remote workspaces of other JaCa-Android applications please refer to the example reported in Section 2.6.

2.1 Example 01 - Working with GUI

Before going in the details of this first example we provide a basic description of the GUIArtifact, the basic artifact used for managing an Android GUI in JaCa-Android.

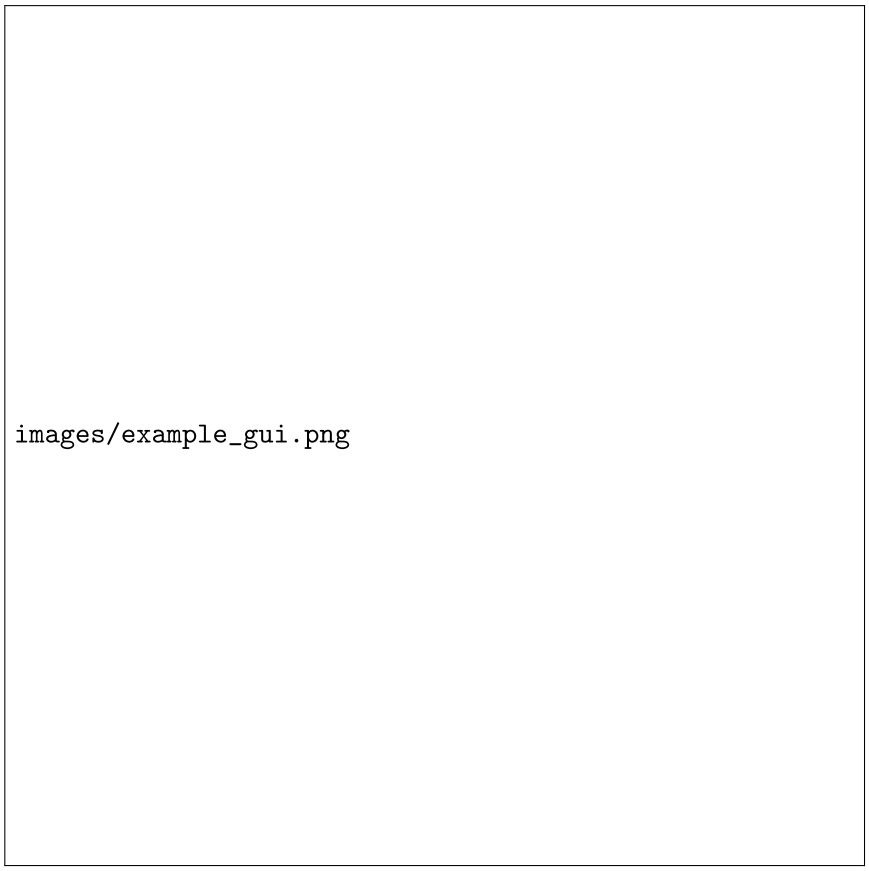

Figure 2.1: The GUI structure of the examples considered in this chapter.

2.1.1 The GUIArtifact

The GUIArtifact is the basic artifact used for managing an Android GUI in JaCa-Android. This artifact wraps a real Android GUI using a IJaCaActivity – which is an interface part of our middleware representing a JaCa-Android extension of an Android GUI base class – for providing developers a connection between the concrete GUI and the artifact that manages it. Thanks to the IJaCaActivity it is hence possible to manipulate the Android GUI directly from the artifact—e.g. see what we have done in the getting started example[1] where inside the `addSMSToList` operation of the ViewerArtifact we added a new SMS to a list displayed in the smart-phone screen.

[1]`http://jaca-android.sourceforge.net/?page_id=257`

This artifact has only one observable property (`state`) representing the current state of the GUI, and provides a set of built-in mechanisms (i.e. protected methods `linkOnStartEventToOp`, `linkOnTouchEventToOp`, etc.) for linking each kind of GUI-related Android event to the artifact operation responsible of the managing of such event. In the artifact initialisation – once the linking between events and operations has been made – is executed an internal operation (`execInternalOp("fetchGUIEvents");`) that manages the notification of Android GUI events. This operation is meant to *cyclically*: *(i)* wait for new events and *(ii)* call the appropriate artifact operation (using the primitive `execInternalOp`).

The GUIArtifact cannot be directly used for realising a GUI in JaCa-Android. For concretely realise an artifact-based GUI we need to specialise the GUIArtifact (like in the case of the ViewerArtifact in the getting started): *(i)* explicitly defining the links between events and operations we are interested in, *(ii)* wrapping the specific IJaCaActivity – i.e. the GUI – we want to manage.

2.1.2 The sample

In this first example we will show how to work with GUI in JaCa-Android through the classic hello world example. You can find the source code of this example in the folder `/jaca/android/tests/gui` placed inside the main `tests` folder. The GUI used in this example is composed of three different tabs (wrapped in three different artifacts), each one of them contains a `print` button that once clicked will produce in standard output the following message: `Hello World from X`, where `X` is the identifier of the Artifact used to discriminate the tab the user was in when he pressed the `print` button.

In this example we will use three instance of the HelloWorldGUIArtifact, each one representing one tab of our sample, extending the base GUIArtifact class.

In Table 2.1 is reported a source code snippet of the HelloWorldGUIArtifact. HelloWorldGUIArtifact highlights:

- In the artifact initialisation the protected methods `linkOnXXXEventToOp` are used for linking the execution of artifact's internal operations: *(i)* to the occurrence of Android events, and *(ii)* to the corresponding actions that the user can do on the GUI (`linkOnClickEventToOp(btnInc,"onClick")` for linking the execution of the `onClick` internal operation to clicks on the print buttons).

```
public class HelloWorldGUIArtifact extends GUIArtifact{

  protected void init(JaCaActivity activity, Bundle savedInstanceState) {
    super.init(activity, savedInstanceState);
    linkOnStartEventToOp("onStart");
    linkOnStopEventToOp("onStop");
    Button btnInc = (Button) activity.findViewById(R.id.btnPrint);
    linkOnClickEventToOp(btnInc, "onClick");
  }

  @OPERATION void print() {
    System.out.println("Hello World from: "+getId().getName());
  }

  @INTERNAL_OPERATION void onClick(View view) {
    signal(getId().getName());
  }

  @INTERNAL_OPERATION void onStart() {
    signal("onStart");
  }

  @INTERNAL_OPERATION void onStop() {
    signal("onStop");
  }
  ...
}
```

Table 2.1: Source code snippet of the HelloWorldGUIArtifact.

- In the `onClick` internal operation the HelloWorldGUIArtifact emits a signal (perceivable by agents that are observing the artifact) containing the name of the artifact identifying the tab. The name of the artifact is retrieved by the `getId()` method inherited from the `Artifact` base class.

- The other internal operations generate signals indicating the occurrence of particular Android events (`onStart`, `onStop`, etc.).

In Table 2.2 is reported the source code of the agent used in this first example. Agent highlights:

- The agent has a `init` plan where it lookups and focuses the workspace artifact. In this way, each time a new artifact is created in the workspace the agent is able to receive the `+artifact(Name, Template, Id)` event.

- The `+artifact(Name, Template, Id)` event is handled by a set of reactive plans that the agent uses for starting focusing each HelloWorldGUIArtifact as soon as it is created.

```
!init.

+!init
  <-  lookupArtifact("workspace", Id);
  focus(Id);
  println("init done").

+state(State) [artifact_name(_,X)]
  <-  println("state  ", X, " :", State).

+onStart [artifact_name(_,X)]
  <-  println("onStart ", X).

+onStop [artifact_name(_,X)]
  <-  println("onStop ", X).

  ...

+artifact("first_activity", _, Id)
  <-  println("first activity created");
      focus(Id).

+artifact("second_activity", _, Id)
  <-  println("second activity created");
      focus(Id).
  ...

+first_activity
  <-  print [artifact_name("first_activity")].

+second_activity
  <-  print [artifact_name("second_activity")].

+third_activity
  <-  print [artifact_name("third_activity")].
```

Table 2.2: Source code snippet of the *Jason* agent used in this test.

- When a click on the GUI buttons is performed by the user the corresponding HelloWorldGUIArtifact generates a signal that is handled by a proper reactive plan of the agent (`+first_activity` in case of signal coming from the first HelloWorldGUIArtifact, `+second_activity` in case of signal coming from the second HelloWorldGUIArtifact, etc.) where the `print` operation of the HelloWorldGUIArtifact is used for printing in the `console` the Hello World message.

We know that at first glance the dynamic just described could seems a bit tricky. Actually it is, but we realised this example in order to show you: *(i)* the linking of artifacts' operations to both GUI and Android events and *(ii)* the possibility to notify to agents the occurrence of events performed by the user on the GUI. In a real world application, you can imagine to

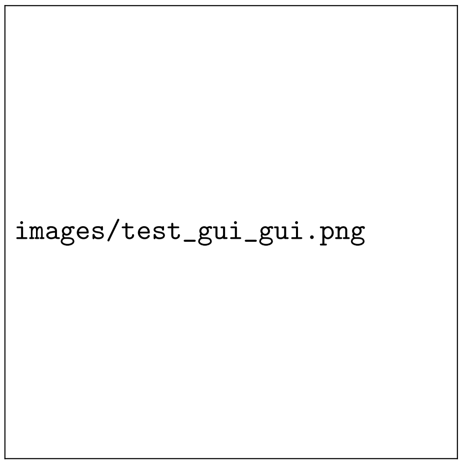

Figure 2.2: A screenshot of the described sample.

update the GUI directly inside the artifact internal operation (the `onClick` in the example) that handles a particular event without generating a signal for delegating the update task to the agent (unless you do not want to do this on purpose, e.g. for notifying to the agent the click, as in the case of this sample).

2.2 Example 01 - Working with Sensors

In this second example we will show how to work with the device's Sensors in JaCa-Android. You can find the source code of this sample in the folder `/jaca/android/tests/allsensors` placed inside the main `tests` folder. The agent used in this sample joins the MiddlewareServices workspace for working with the AllSensorManager artifact, an artifact provided by the middleware that encapsulates the access to three of the most used Android sensors (the accelerometer sensor, the geomagnetic sensor, and the orientation sensor) and then dynamically updates the current sensors values in the application GUI (represented by a proper extension of the GUIArtifact, not reported here for simplicity).

It is worth nothing that, from a programming point of view, the AllSensorManager is just a simple specialisation of the SensorManagerArtifact which is the basic artifact used for having access to sensors information—it is exactly the same relation described for the GUIArtifact and its specialisations.

The artifact stores the information related to the sensors values inside a proper set of observable properties. These observable properties are then dynamically updated by the artifact, by linking events that come from the device's sensors to proper artifact's internal operations with a linking mechanism analogous to the one described for the GUIArtifact.

Besides the artifact used in this sample JaCa-Android provides a set of other artifacts for manipulating the device's sensors, for further details see the source code or the api of the middleware.

```
public class AllSensorsManager extends SensorManagerArtifact{

  @OPERATION public void init(Integer delay) throws Exception{
    super.init(delay, Sensor.TYPE_ACCELEROMETER,
      Sensor.TYPE_MAGNETIC_FIELD, Sensor.TYPE_ORIENTATION);
    linkOnSensorChangedEventToOp(Sensor.TYPE_ACCELEROMETER, "onSensorChanged");
    linkOnSensorChangedEventToOp(Sensor.TYPE_MAGNETIC_FIELD, "onSensorChanged");
    linkOnSensorChangedEventToOp(Sensor.TYPE_ORIENTATION, "onSensorChanged");
    defineObsProperty(SensorManagerArtifact.SENSOR_ACCELEROMETER, 0,0,0);
    defineObsProperty(SensorManagerArtifact.SENSOR_GEOMAG, 0,0,0);
    defineObsProperty(SensorManagerArtifact.SENSOR_ORIENTATION, 0,0,0);
  }

  @Override
  @INTERNAL_OPERATION public void onSensorChanged(SensorEvent event) {
    switch(event.sensor.getType()){
      case Sensor.TYPE_ACCELEROMETER:
        getObsProperty(SensorManagerArtifact.SENSOR_ACCELEROMETER)
          .updateValues(event.values[0], event.values[1], event.values[2]);
        sensorReady = true;
      break;

      case Sensor.TYPE_MAGNETIC_FIELD:
        getObsProperty(SensorManagerArtifact.SENSOR_GEOMAG)
          .updateValues(event.values[0], event.values[1], event.values[2]);
        sensorReady = true;
        ...
      break;
    }

    //Orientation management
    if (... && sensorReady) {
      ...
      getObsProperty(SensorManagerArtifact.SENSOR_ORIENTATION).
        updateValues(azimuth, pitch, roll);
    }
  }
}
```

Table 2.3: Source code snippet of the AllSensorsManager.

In Table 2.3 is reported a source code snippet of the AllSensorsManager. AllSensorManager highlights:

- In the initialisation phase we: *(i)* call the `init` of the SensorManagerArtifact (the superclass) for making our AllSensorManager artifact able to retrieve data from the desired sensors, *(ii)* we register the `onSensorChanged` internal operation (thanks to the `linkOnSensorChangedEventToOp`) as the operation to be called when new sensors value are available, and *(iii)* we define the set of observable properties that will expose to agents the current sensors value.

- Inside the onSensorChanged internal operation we update the observable properties containing the current sensors value (for simplicity some part of the source code was omitted).

```
!start.

+!start
  <-  !init;
      !do_job.

/*Initialisation plan for joining the MiddlewareServices workspace*/
+!init
  <-  ...

/************************* Main agent plan ************************/
+!do_job : art_id(console,LocalConsoleID) & wsp_id(jaca_services,WspID)
  <-  focusWhenAvailable("all-sensors-manager");
      startMonitoring[wsp_id(WspID)];
      println("Ready.")[artifact_id(LocalConsoleID)].

/********* Plans that handle reactions to sensors' events *********/
+sensor_accelerometer(X,Y,Z) : art_id(gui,GUIID)
  <-  updateAccSensorsInfo(X, Y, Z)[artifact_id(GUIID)].

+sensor_orientation(X,Y,Z) : art_id(gui,GUIID)
  <-  updateOrientationSensorsInfo(X, Y, Z)[artifact_id(GUIID)].

+sensor_geomag(X,Y,Z) : art_id(gui,GUIID)
  <-  updateGeomagSensorsInfo(X, Y, Z)[artifact_id(GUIID)].
```

Table 2.4: Source code snippet of the *Jason* agent used in this test.

In Table 2.4 is reported the source code of the agent using the AllSensorManager artifact. Agent highlights:

- The agent use the `init` plan for joining the MiddlewareServices workspace for having access to the AllSensorManager artifact.

- Once joined the MiddlewareServices workspace the agents starts its job (plan do_job) waiting for the AllSensorManager to be available (`focusWhenAvailable("all-sensors-manager")`).

images/test_sensors_gui.png

Figure 2.3: A screenshot of the described sample.

- Once focused the AllSensorManager artifact the behaviour of the agent is entirely managed by a set of reactive plans related to updates of the artifact's observable properties. As soon as a new sensors values are available the AllSensorManager retrieves this values and then updates its observable properties accordingly (through the `onSensorChanged` internal operation). The agent reacts to these observable properties changes updating the GUI with the new values (`updateAccSensorsInfo`, `updateGeomagSensorsInfo` and `updateOrientationSensorsInfo` operations provided by the artifact representing the application GUI).

2.3 Example 02 - Working with the GPS

In this example we will show how to work with the GPS in JaCa-Android. You can find the source code of this sample in the folder `/jaca/android/tests/gps` placed inside the main `tests` folder.

The agent used in this sample joins the MiddlewareServices workspace for working with the GPSManager artifact, an artifact provided by the middleware that encapsulates the access to the device's GPS. The GPSManager artifact is an artifact extending the LocationManagerArtifact, a JaCa-Android artifact that provides basic functionalities for managing the device's geographical location. The artifact has a set of observable properties (`latitude`, `longitude`, `altitude`, `bearing`, etc.) that make directly accessible to agents the information coming from the GPS. These observable properties are dynamically updated by the artifact, by means of the usual linking mechanism between Android events – GPS-related events in this case – and artifact operations as described previously.

```
public class GPSManager extends LocationManagerArtifact {

  protected void init(int minTime, int minDistance) {
    super.init(minTime, minDistance);
    linkOnLocationChangedEventToOp(LocationManager.GPS_PROVIDER,
      "onLocationChange");
    Location location = getLocationManager()
                          .getLastKnownLocation(LocationManager.GPS_PROVIDER);
    if (location!=null) {
      defineObsProperty(LATITUDE, location.getLatitude());
      defineObsProperty(LONGITUDE, location.getLongitude());
      defineObsProperty(ALTITUDE, location.getAltitude());
      ...
    } else {
      defineObsProperty(LATITUDE, 0);
      defineObsProperty(LONGITUDE, 0);
      ...
    }
  }

  protected void dispose() {
    super.dispose();
    removeLocationUpdates(LocationManager.GPS_PROVIDER);
  }

  @INTERNAL_OPERATION void onLocationChange(Location arg0) {
    signal(ON_LOCATION_CHANGE, arg0.getProvider());
    getObsProperty(LATITUDE).updateValue(arg0.getLatitude());
    getObsProperty(LONGITUDE).updateValue(arg0.getLongitude());
    ...
  }
}
```

Table 2.5: Source code snippet of the GPSManager.

In Table 2.5 is reported a source code snippet of the GPSManager artifact.GPSManager highlights:

- In the initialisation phase: *(i)* the definition of the artifact observable properties (for the full list see the artifact's source code), *(ii)* the linking between the gps-related events and the artifact operations responsible of the handling of such events.

- The observable properties are updated as soon as new gps information are retrieved from the device's GPS. This is done inside the `onLocationChange` internal operation that, thanks to the link done in the `init`, is called each time new sensors values are available.
- The generation of appropriate signals each time the artifact starts/ends receiving gps-related events. The start/end of the monitoring of such events is handled by the `startMonitoring`/`stopMonitoring` operations inherited from the `LocationManagerArtifact`.

```
!start.

+!start
  <-  !init;
      !do_job.

/*Initialisation plan for joining the MiddlewareServices workspace*/
+!init
  <-  ...

/************************* Main agent plan ************************/
+!do_job : wsp_id(jaca_services, WspID) & art_id(console,LocalConsoleID)
  <-  focusWhenAvailable("gps-manager")[wsp_id(WspID)];
      startMonitoring[wsp_id(WspID)];
      println("Ready.")[artifact_id(LocalConsoleID)].

+latitude(Latitude) : art_id(console,LocalConsoleID)
  <-  println("Latitude: ", Latitude)[artifact_id(LocalConsoleID)].

+longitude(Longitude) : art_id(console,LocalConsoleID)
  <-  println("Longitude: ", Longitude)[artifact_id(LocalConsoleID)].

+altitude(Altitude) : art_id(console,LocalConsoleID)
  <-  println("altitude: ", Altitude)[artifact_id(LocalConsoleID)].

...
```

Table 2.6: Source code snippet of the *Jason* agent used in this test.

In Table 2.6 is reported the source code of the agent using the GPSManager artifact. Agent highlights:

- For first the agent joins the MiddlewareServices workspace using the usual `init` plan.

- Then the agent focuses the `GPSManager`, contained in the MiddlewareServices workspace, and then invokes the `startMonitoring` operation on such artifact for starting receiving the information coming from the device's GPS.

- Updates to the `GPSManager` observable properties are managed by a set of reactive plans in which the agent simply prints in the console the new information coming from the GPSManager.

images/test_gps_gui.png

Figure 2.4: A screenshot of the GPS sample in action.

2.4 Example 03 - Reading the battery status

In this example we will show how to retrieve information related to the device battery in JaCa-Android. You can find the source code of this sample in the folder `/jaca/android/tests/battery` placed inside the main `tests` folder. The agent used in this sample joins the MiddlewareServices workspace for working with the BatteryArtifact artifact, an artifact provided by the middleware that allows to retrieve information about the device's battery.

The BatteryArtifact is an artifact extending the BroadcastReceiverArtifact, a JaCa-Android artifact that provides basic functionalities for registering an artifact as the target of Android broadcasts, in this case broadcasts concerning battery information updates. Roughly speaking the BroadcastReceiverArtifact can be considered as the JaCa-Android version of the classical Android BroadcastReceiver.

In Table 2.7 is reported a source code snippet of the BatteryArtifact artifact. BatteryArtifact highlights:

- In the artifact initialisation we create the observable properties related to the battery level, status and health (namely the `battery_level`, the

```
public class BatteryArtifact extends BroadcastReceiverArtifact {

  ...

  protected void init(boolean abortBroadcast) {
    super.init();
    defineObsProperty(HEALT, HEALT_GOOD);
    defineObsProperty(STATUS, "ok");
    defineObsProperty(LEVEL, 100);
    IntentFilter filter = new IntentFilter();
    filter.addAction(BATTERY_CHANGED);
    IntentFilter filter2 = new IntentFilter();
    filter.addAction(BATTERY_LOW);
    IntentFilter filter3 = new IntentFilter();
    filter3.addAction(BATTERY_OK);
    linkBroadcastReceiverToOp(filter, "batteryInfoReceived", abortBroadcast);
    linkBroadcastReceiverToOp(filter2, "batteryLowReceived", abortBroadcast);
    linkBroadcastReceiverToOp(filter3, "batteryOkReceived", abortBroadcast);
  }

  @INTERNAL_OPERATION public void batteryOkReceived(BroadcastReceiver broadcastReceiver,
     Context context, Intent intent) {
    getObsProperty(STATUS).updateValue(STATUS_OK);
  }

  @INTERNAL_OPERATION public void batteryLowReceived(BroadcastReceiver broadcastReceiver,
      Context context, Intent intent) {
    getObsProperty(STATUS).updateValue(STATUS_LOW);
  }

  @INTERNAL_OPERATION public void batteryInfoReceived(BroadcastReceiver broadcastReceiver,
    Context context, Intent intent) {
    if (intent.getAction().equals(BATTERY_CHANGED)) {
      int level = intent.getIntExtra(BatteryManager.EXTRA_LEVEL, 100);
      getObsProperty(LEVEL).updateValue(level);
      int healt = intent.getIntExtra(BatteryManager.EXTRA_HEALTH,
         BatteryManager.BATTERY_HEALTH_GOOD);
      String obsHealt = getObsProperty(HEALT).stringValue();

      if(healt!=BatteryManager.BATTERY_HEALTH_GOOD && obsHealt.equals(HEALT_GOOD))
        getObsProperty(HEALT).updateValue(HEALT_DEAD);
      else if(healt!=BatteryManager.BATTERY_HEALTH_DEAD && obsHealt.equals(HEALT_DEAD))
        getObsProperty(HEALT).updateValue(HEALT_GOOD);
    }
  }

  protected void dispose(){
    super.dispose();
    unlinkBroadcastReceiverToOp(OP_BATTERY_INFO_RECEIVED);
    ...
  }
}
```

Table 2.7: Source code snippet of the BatteryArtifact.

battery_status and the battery_health) and then we register the artifact to be notified by the broadcasts of interest for continuously retrieving updates related to the battery status.

- In the artifact disposal we call the unlinkBroadcastReceiverToOp method (inherited by the BroadcastReceiverArtifact) for unregister the artifact, during its disposal, as one of the target to be notified by battery broadcasts.

```
!start.

+!start
  <-  !init;
      !do_job.

/*Initialisation plan for joining the MiddlewareServices workspace*/
+!init
  <-  ...

/************************ Main agent plan ************************/
+!do_job : wsp_id(jaca_services,WspID) & art_id(console,LocalConsoleID)
  <-  focusWhenAvailable("battery-artifact")[wsp_id(WspID)];
      println("Ready.")[artifact_id(LocalConsoleID)].

/********** Plans that handle reactions to battery events **********/
+battery_level(Value) : art_id(console,LocalConsoleID)
  <-  println("Battery level is ", Value)[artifact_id(LocalConsoleID)].

+battery_status(Value) : art_id(console,LocalConsoleID)
  <-  println("Battery status is ", Value)[artifact_id(LocalConsoleID)].

+battery_healt(Value) : art_id(console,LocalConsoleID)
  <-  println("Battery healt is ", Value)[artifact_id(LocalConsoleID)].
```

Table 2.8: Source code snippet of the *Jason* agent used in this test.

In Table 2.8 is reported the source code of the agent using the BatteryArtifact artifact. Agent highlights:

- For first the agent joins the MiddlewareServices workspace using the usual init plan.

- Then the agent focus the BatteryArtifact, contained in the MiddlewareServices, for receiving the information related to the battery.

- As soon as new battery information are available the BatteryArtifact generates proper signals that are handled by the agent with a set of reactive plans that prints in the console the new battery information.

`images/test_battery_gui.png`

Figure 2.5: A screenshot of the described sample.

2.5 Example 04 - Working with the device connectivity

In this example we will show how to manage the different kind of connectivity available on the device in JaCa-Android. You can find the source code of this example in the folder `/jaca/android/tests/connectivity` placed inside the main `tests` folder. The agent used in this sample joins the MiddlewareServices workspace for working with the ConnectivityManager artifact, an artifact provided by the middleware that allows to manage the different kind of connectivity available on the device. Even if this artifact is meant to be able to manage all the different kind of connectivity available, currently, besides the management of the device airplane mode, only the WiFi is supported. This artifact, like the BatteryArtifact, extends the BroadcastReceiverArtifact for receiving the Android's broadcast (i.e. changes in the connectivity status in this case).

In Table 2.9 is reported a source code snippet of the ConnectivityManager artifact. ConnectivityManager highlights:

- In the artifact initialisation we create the observable properties related to the device's connectivity status and then we register the artifact to be notified by the broadcasts of interest for continuously retrieving updates related to such status.
- Thanks to the usual linking mechanism between android events and artifacts' internal operations each time a new Android broadcast is sent the `onReceive` operation is invoked. Inside this operation the artifact's observable properties are updated with the last broadcasted information.
- The artifact provides also a set of operations for: *(i)* enabling/disabling the WiFi connectivity, and *(ii)* enabling/disabling the airplane-mode[2] on the device.

In Table 2.10 is reported the source code of the agent used in this example. Agent highlights:

- For first the agent joins the MiddlewareServices workspace using the usual `init` plan.

[2]This is the Android term used for indicating a device with at least cellular network turned off.

```
public class ConnectivityManager extends BroadcastReceiverArtifact {

  protected void init() {
    super.init();
    boolean isAirplaneEnabled = ...

    wifiManager = (WifiManager) getApplicationContext().getSystemService(Context.WIFI_SERVICE);

    if(isAirplaneEnabled)
      defineObsProperty(AIRPLANE_MODE_STATUS, ON_VALUE);
    else
      defineObsProperty(AIRPLANE_MODE_STATUS, OFF_VALUE);

    if(wifiManager.isWifiEnabled())
      defineObsProperty(WIFI_STATUS, ON_VALUE);
    else
      defineObsProperty(WIFI_STATUS, OFF_VALUE);

    IntentFilter filter = new IntentFilter();
    filter.addAction("android.intent.action.SERVICE_STATE");
    filter.addAction(WifiManager.WIFI_STATE_CHANGED_ACTION);
    linkBroadcastReceiverToOp(filter, "onReceive");
}

@INTERNAL_OPERATION void onReceive(BroadcastReceiver broadcastReceiver,
    Context context, Intent intent) {
    boolean isAirplaneEnabled = ...
    boolean obsIsAirplaneOn = getObsProperty(AIRPLANE_MODE_STATUS).equals(ON_VALUE);
    boolean obsIsWifiOn = getObsProperty(WIFI_STATUS).equals(ON_VALUE);

    if(obsIsAirplaneOn!=isAirplaneEnabled){
      if(isAirplaneEnabled)
        getObsProperty(AIRPLANE_MODE_STATUS).updateValue(ON_VALUE);
      else
        getObsProperty(AIRPLANE_MODE_STATUS).updateValue(OFF_VALUE);
    }
    if(obsIsWifiOn!=wifiManager.isWifiEnabled()){
      if(obsIsWifiOn)
        getObsProperty(WIFI_STATUS).updateValue(ON_VALUE);
      else
        getObsProperty(WIFI_STATUS).updateValue(OFF_VALUE);
    }
  }

  @OPERATION public void enableWiFi(){...}

  @OPERATION public void disableWiFi(){...}

  @OPERATION void enableAirplaneMode(){...}

  @OPERATION void disableAirplaneMode(){...}
  ...
}
```

Table 2.9: Source code snippet of the ConnectivityManager.

- Then the agent focuses the `ConnectivityManager`, contained in the MiddlewareServices, for receiving the information related to the device connectivity status.

- As soon as new information are available the `ConnectivityManager` generates proper signals that are handled by the agent with a set of reactive plans that print in the console the new connectivity information.

- Inside the `do_job` plan we test the different functionalities provided by the `ConnectivityManager` artifact enabling and then disabling the WiFi connection and the airplane mode.

images/test_connectivity_gui.png

Figure 2.6: A screenshot of the described sample.

```
!start.

+!start
  <-  !init;
      !do_job.

/*Initialisation plan for joining the MiddlewareServices workspace*/
+!init
  <-  ...

/******************** Main agent plan ********************/
+!do_job : wsp_id(jaca_services, WspID) & art_id(console,LocalConsoleID)
  <-  focusWhenAvailable("connectivity-manager")[wsp_id(WspID)];
      println("[TESTING WIFI]")[artifact_id(LocalConsoleID)];
      !test_wifi;
      .wait(10000);
      println("[TESTING AIRPLANE]")[artifact_id(LocalConsoleID)];
      !test_airplane;
      .wait(10000);
      println("[DISABLE ALL]")[artifact_id(LocalConsoleID)];
      !disable_all.

+!test_wifi : true
  <-  ?art_id(console,LocalConsoleID);
      println("enabling wifi")[artifact_id(LocalConsoleID)];
      enableWiFi;
      println("wifi enabled")[artifact_id(LocalConsoleID)];
      .wait(15000);
      println("disabling wifi")[artifact_id(LocalConsoleID)];
      disableWiFi;
      println("wifi disabled")[artifact_id(LocalConsoleID)].


+!test_airplane : true
  <-  ?art_id(console,LocalConsoleID);
      println("enabling airplane mode")[artifact_id(LocalConsoleID)];
      enableAirplaneMode;
      println("airplane enabled")[artifact_id(LocalConsoleID)];
      .wait(10000);
      println("disabling airplane mode")[artifact_id(LocalConsoleID)];
      disableAirplaneMode;
      println("airplane disabled")[artifact_id(LocalConsoleID)].

+!disable_all : true
  <-  ?art_id(console,LocalConsoleID);
      println("disabling all")[artifact_id(LocalConsoleID)];
      enableAirplaneSpecific("cell,bluetooth,wifi");
      println("all disabled")[artifact_id(LocalConsoleID)].

+airplane_mode_status(Value) : art_id(console,LocalConsoleID)
  <-  println("[STATUS AIRPLANE:]", Value)[artifact_id(LocalConsoleID)].

+wifi_status(Value) :art_id(console,LocalConsoleID)
  <-  println("[STATUS WIFI:]: ", Value)[artifact_id(LocalConsoleID)].
```

Table 2.10: Source code snippet of the *Jason* agent used in this test.

2.6 Example 05 - Working in remote **JaCa-Android** workspaces

In this example we will show how it is possible to realise a JaCa-Android application that works also in a remote JaCa-Android workspace (so a workspace related to another JaCa-Android application). You can find the source code of this example in the folder `/jaca/android/tests/remotewsp/android` placed inside the main `tests` folder.

Before launching this example is needed to install and run the `[JaCa-Android]RemoteWsp.apk` application contained in the `utils-projects/remote-android` folder on the root of the standard JaCa-Android distribution. This apk contains a simple JaCa-Android application (the application is composed by a Calculator artifact and a test agent that uses the artifact for performing basic computations) configured for being reachable by external JaCa-Android applications.

A JaCa-Android application reachable from others ones can be realised in the following way. For first is necessary to configure the JaCaService in the Android manifest as reported below:

```
<service android:name="jaca.android.JaCaService" android:exported="true" >
  <intent-filter>
    <!--
        The action is the "Android address" that we want to expose as a remote wsp.
        In the mas2j MUST be provided the same address when installing the
        infrastructure: service(android,"remote.myaddress")
    -->
    <action android:name="remote.myaddress" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</service>
```

Then in the `mas2j` configuration file we need to create an environment of type `CartagoEnvironment` as reported below:

```
MAS remote_wsp {

infrastructure: Centralised

    environment:
      c4jason.CartagoEnvironment("infrastructure", service(android,"remote.myaddress"))

    agents: agent agentArchClass c4jason.CAgentArch;

    aslSourcePath: "test/remote";
}
```

In this way, when we launch this application we also install a CArtAgO node (of type `infrastructure`) that, thanks to the service(android,"remote.myaddress") parameter, makes the application reachable by external JaCa-Android ones at the address remote.myaddress. The CArtAgO infrastructure will use the configuration just described for allowing other applications to communicate with the remote one using the infrastructure protocol specified (so in this case means using the CArtAgO infrastructure protocol that relies on Android IPC). For further details please refer to the CArtAgO website[3]. Please note that the address specified in the mas2j **must** be the same as the one specified in the JaCaService inside the Android manifest file.

Our test application is composed by a simple agent that join the default workspace of the [JaCa-Android]RemoteWsp.apk application and then starts to use the Calculator artifact contained in it printing the obtained result in the local console and in the remote one. For making our test application able to reach the remote workspaces its mas2j file must be configured as reported below:

```
MAS test {

infrastructure: Centralised

   environment: c4jason.CartagoEnvironment("standalone", protocol(android))

    agents:
    agent agentArchClass c4jason.CAgentArch;

    aslSourcePath: "jaca/android/tests/remotewsp/android";
}
```

The configuration for the CartagoEnvironment specifies that this application will not be reachable from the outside (standalone parameter) but that it will be able to join remote workspaces through the Android infrastructure protocol (protocol(android) parameter).

In Table 2.11 is reported the source code of the agent used in this test. Agent highlights:

- The test provides some facilitations that allow the user to insert the address (remote.myaddress in our case) to join from a GUI (thanks to the AddressActivity) and to memorise the address provided (thanks to the Config artifact). For sake of simplicity we are not reporting here details about the implementation of these facilitations.

[3] http://cartago.sourceforge.net/?page_id=139

- Once the address has been setted the agent try to join the remote workspace thanks to the `joinRemoteWorkspace` action (`joinRemoteWorkspace("default", Address, RemoteWspId)`). In case of error during the connection a proper *Jason* recovery plan is instantiated (`-!join_remote_wsp`) and the `AddressActivity` is displayed again for letting the user insert a new address for the remote join.

- Once the `joinRemoteWorkspace` succeeds the `work_in_remote_wsp` plan is instantiated. Within this plan the agent works with the remote `Calculator` artifact and prints the result of its computation in both the local and the remote console.

images/test_remote1_gui.png

Figure 2.7: A screenshot of the described sample with in evidence the prints produced in both the local and remote application.

```
!init.

+!init
  <-  lookupArtifact("console",LocalConsoleID);
      focusWhenAvailable("remote-gui");
      lookupArtifact("remote-gui",GUIArtID);
      makeArtifact("config", "jaca.android.tests.remotewsp.common.Config",
        ["android.config"], ConfID);
      linkArtifacts(GUIArtID, "out-1", ConfID);
      +art_id(console, LocalConsoleID);
      !join_remote_wsp.

+!join_remote_wsp : art_id(console, LocalConsoleID)
  <-  getAddress(Address);
    //Address not setted yet, we start the activity for obtaining the address
    if (Address == "") {
      startExplicitActivityForResult("jaca.android.tests.remotewsp.common.AddressActivity", 0);
    //Address setted, we try to join the remote wsp
    } else {
      joinRemoteWorkspace("default", Address, RemoteWspId);
      +remote_wsp(RemoteWspId);
      println("Join remote done!")[artifact_id(LocalConsoleID)];
      !work_in_remote_wsp;
    }.

//Failure in joining the remote wsp: we ask again for the address
-!join_remote_wsp : true
  <-  startExplicitActivityForResult("jaca.android.tests.remotewsp.common.AddressActivity", 0).

/*Signal received by the RemoteGUI: the user has inserted the remote address,
 we can re-instantiate the join_remote_wsp plan */
+address_setted  :  true
  <-  !join_remote_wsp.

+!work_in_remote_wsp : art_id(console, LocalConsoleID)
<-  lookupArtifact("myCalc", CalcId)[wsp_id(RemoteWspId)];
      lookupArtifact("console", RemoteConsoleID)[wsp_id(RemoteWspId)];
      println("Remote calculator artifact found")[artifact_id(LocalConsoleID)];
      println("Using the calculator, the next prints can
         be seen on the remote app as well")[artifact_id(LocalConsoleID)];
      sum(2,3,X);
      println("[remote-agent:]The sum between 2 and 3 is ", X)[artifact_id(RemoteConsoleID)];
      println("The sum between 2 and 3 is ", X)[artifact_id(LocalConsoleID)];
      sub(1,2,Y);
      println("[remote-agent:]The sub between 1 and 2 is ", Y)[artifact_id(RemoteConsoleID)];
      println("The sub between 1 and 2 is ", Y)[artifact_id(LocalConsoleID)];
      division(6,3,Z);
      println("[remote-agent:]The div between 6 and 3 is ", Z)[artifact_id(RemoteConsoleID)];
      println("The div between 6 and 3 is ", Z)[artifact_id(LocalConsoleID)].
```

Table 2.11: Source code snippet of the *Jason* agent used in this test.

2.7 Example 06 - Working with standard JaCa remote workspace

In this example we will show how it is possible to realise a JaCa-Android application that works also in a remote JaCa workspace (so a workspace related to another JaCa application). You can find the source code of this example in the folder `/jaca/android/tests/remotewsp/lipermi` placed inside the main `tests` folder.

Before running this test you need to launch the JaCa application contained inside the `utils-projects/[JaCa]RemoteWsp` folder. Once inside that folder you will find the mas2j for launching the application inside the `src/test/remote` folder. The `[JaCa]RemoteWsp` application is a JaCa version of exactly the same application used in the previous test (so an application composed by a `Calculator` artifact and a simple agent using it).

Note that JaCa-Android applications can communicate with remote JaCa ones thanks to a CArtAgO infrastructure protocol that is `lipermi`-based. `lipermi`[4] is a library for managing remote method invocation in Java on top the TCP/IP protocol. Therefore for being able to run this sample you must run the `[JaCa]RemoteWsp` application from an IP node reachable from the Android device.

A JaCa application reachable from others ones using the `lipermi` infrastructure protocol can be realised simply configuring the mas2j file as follow (for further details see the CArtAgO website[5]) :

```
MAS remote_wsp{

infrastructure: Centralised

    environment: c4jason.CartagoEnvironment("infrastructure", service(lipermi))

    agents: agent agentArchClass c4jason.CAgentArch;

    classpath : "../../../lib/*.jar";
}
```

Table 2.12: mas2j configuration for a JaCa application reachable from remote ones through the `lipermi` infrastructure protocol.

This mas2j configuration differ from the previous one just for the `service` parameter. In this case the infrastructure protocol that we use for joining and working with the remote workspace is the `lipermi` protocol. In the

[4]`http://lipermi.sourceforge.net/`
[5]`http://cartago.sourceforge.net/?page_id=139`

previous case we used the Android protocol but here we can not use it as our infrastructure protocol since the remote application in this case is not Android-based. Moreover we are not specifying an explicit address for the application, indeed in this case the remote address will be defined by the IP address of the machine where we will run the remote application followed by the default port for the CArtAgO `lipermi` infrastructure protocol (20101).

In this test, like the previous one, we use a simple agent that join the default workspace of the `[JaCa]RemoteWsp` application and then start to use the Calculator artifact contained in it printing the obtained results in the local console and in the remote one.
For making our test application able to reach the remote workspaces its mas2j file must be configured as follow:

```
MAS test {

infrastructure: Centralised

    environment: c4jason.CartagoEnvironment("standalone", protocol(lipermi))

    agents:
    agent agentArchClass c4jason.CAgentArch;

    aslSourcePath: "jaca/android/tests/remotewsp/lipermi";
}
```

Table 2.13: mas2j configuration for a JaCa-Android application able to reach remote JaCa ones – even JaCa-Android ones – through the `lipermi` infrastructure protocol.

The configuration for the `CartagoEnvironment` specify that this application will not be reachable from the outside (`standalone` parameter) but that it will be able to join remote workspace through the lipermi infrastructure protocol (`protocol(lipermi)` parameter).
Agent Highlights:

- The source code of the agent is exactly the same as the one used in the previous test (and for this reason we are not reporting it here). Is the underlying CArtAgO infrastructure that manages the interaction with the remote application accordingly with the protocol specified in the mas2j configuration.

- Our JaCa-Android application differ from the one used in the previous test only in the configuration of the mas2j file. Indeed the only thing we need to change is the protocol to use for joining the remote workspace.

- It is quite simple to realise JaCa-Android application that can interact with other JaCa ones distributed in other network nodes. The same procedure can be used for joining remote applications running inside LANs or remote internet nodes.

images/test_remote2_gui.png

Figure 2.8: A screenshot of the described sample with in evidence the prints produced in both the local and remote application.

Chapter 3

Examples

TO BE COMPLETED:

3.1 Context-aware SMS notifier

TO BE COMPLETED:

3.2 JaCa-Locale

TO BE COMPLETED:

Chapter 4

Performance tests

TO BE COMPLETED:

4.1 Reactivity test

TO BE COMPLETED:

4.2 Memory occupation test

TO BE COMPLETED:

4.3 Heavy computation test

Bibliography

[1] R. Bordini and J. Hübner. BDI agent programming in AgentSpeak using Jason. In F. Toni and P. Torroni, editors, *CLIMA VI*, volume 3900 of *LNAI*, pages 143–164. Springer, Mar. 2006.

[2] R. Bordini, J. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason.* John Wiley & Sons, Ltd, 2007.

[3] A. Ricci, M. Piunti, M. Viroli, and A. Omicini. Environment programming in CArtAgO. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah-Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications, Vol. 2*, pages 259–288. Springer, 2009.

[4] A. Ricci, M. Punti, and M. Viroli. Environment programming in multi-agent systems - an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 2010. To appear.

[5] A. Ricci, M. Viroli, and A. Omicini. CArtAgO: A framework for prototyping artifact-based environments in MAS. In D. Weyns, H. V. D. Parunak, and F. Michel, editors, *Environments for MultiAgent Systems III*, volume 4389 of *LNAI*, pages 67–86. Springer, Feb. 2007. 3rd International Workshop (E4MAS 2006), Hakodate, Japan, 8 May 2006. Selected Revised and Invited Papers.