1 Reusable Atomic Behaviors

This chapter provides an overview on how to develop Reusable Atomic Behaviors or other assets to be used by a UBIWARE agent. It is assumed that the reader is conceptually familiar with RABs. Furthermore, knowledge of the java programming language is required up to the complexity level needed by the task of the RAB. The first section provides a step by step introduction on how to create a few simple behaviors. The rest of the text provides in depth overview on RAB creation. We assume that for this introduction, the user has the Eclipse programming environment installed. The Eclipse programming environment is not a requirement for developing reusable atomic behaviors, but it is used as a project environment for the example code. The first thing one has to do is to import the Rab programming introduction project provided as sample code to the Eclipse workspace (resource *rab_programming_introduction_project.zip* from the documentation web page).

1.1 Hello Agent World!

To give a first taste on how RABs can be implemented, this section provides a few basic examples on implementation of RABs. Source code for this section can be found in the 'Rab programming introduction' project of the example code.

1.1.1 Hello World

The first RAB we will consider is performing the typical Hello World example. Every RAB is implemented as a java class extending from the class ubiware.core.ReusableatomicBehavior. In this example, we only give an implementation for the doAction method in which we implement the actual action performed by the RAB. The source doAction method code for this RAB looks as follows:

```
protected void doAction() throws Exception {
        System.out.println("Hello world!");
}
```

The S-APL needed in the general context of the ubiware agent to activate this RAB looks as follows:

```
{ sapl:I sapl:do java:fi.jyu.it.iog.ubiware.documentation.HelloWorldBehavior }
sapl:configuredAs sapl:null
```

The code for this example is in the project the 'Rab programming introduction' in the files

- /src/fi/jyu/it/iog/Ubiware/documentation/HelloWorldBehavior.java
- helloWorld.sapl

The example can be run using the 'Hello World.launch' run configuration. If you want to know more about creating your own run configurations, see the section "How to run a Ubiware application" in the "Ubiware application developer guide".

1.1.2 Hello You

In this example, we will show a first example on how to handle parameters handed from the agent to the RAB in the RAB call (The S-APL structure used to activate an RAB). The code looks as follows:

```
{ sapl:I sapl:do java:fi.jyu.it.iog.ubiware.documentation.HelloYouBehavior }
sapl:configuredAs { parameter:name sapl:is "your name" }
```

The parameter which is handed to the RAB has name parameter:name. Notice that this parameter's name is actually a resource. This RAB call can be read as follows: Sapl:I, a shorthand for the actual agent, perform the java action fi.jyu.it.iog.ubiware.documentation.HelloYouBehavior configured as described in the configuration context. The value associated with the resource parameter:name is "yourname" in the context of this configuration.

To handle the parameter, the RAB fetches the value given for the parameter by fetching its value from the object passed to the initializeParameters method. In this case, we want to retrieve the value for the parameter p:name which is a) a string parameter and b) an obligated parameter.

The code for the initilizeParameters method for this RAB looks as follows:

```
void initializeRAB (BehaviorStartParameters parameters) throws
IllegalParameterConfigurationException {
        this.name=parameters.getObligatedStringParameter(nameParameter);
}
```

The code of doAction is then actually performing the action.

```
void doAction() throws Exception {
    System.out.println("Hello " + this.name);
}
```

The code for this example is in the project the 'Rab programming introduction' in the files

- /src/fi/jyu/it/iog/Ubiware/documentation/HelloYouBehavior.java
- helloYou.sapl

The example can be run using the 'Hello You.launch' run configuration delivered with the project.

1.1.3 Print container content

This example will show how the content of a container can be printed. In order to do this, a parameter is passed in a similar way as the previous example. The only difference is that the value specified for the parameter is a container. The RAB call now looks as follows:

```
{ sapl:I sapl:do java:fi.jyu.it.iog.ubiware.documentation.PrintContainerBehavior }
sapl:configuredAs
{
        parameter:container sapl:is {
                ex:S1 ex:P1 ex:O1 .
                ex:S2 ex:P2 {
                        ex:S2a ex:P2a ex:O2a .
                        ex:S2b ex:P2b ex:O2b .

                } .
                ex:S3 ex:P3 ex:O3 .
        }
}
```

The RAB is implemented as follows:

```
void initializeRAB(BehaviorStartParameters parameters) throws
IllegalParameterConfigurationException {
        this.container =
parameters.getObligatedContainerID(containerParameter);
}


void doAction() throws Exception {
        SaplDocument sapl = this.produceN3Document(container);
        System.out.println("Now follows the s-apl code");
        System.out.println(sapl.generateSapl());
}
```

In the initializeParameters method, the ID of the container is taken from the parameters object. This id is then consecutively used to generate the S-APL code which is then in turn printed to the console. The built-in ubiware.shared.PrintBeliefsBehavior works in a similar way.

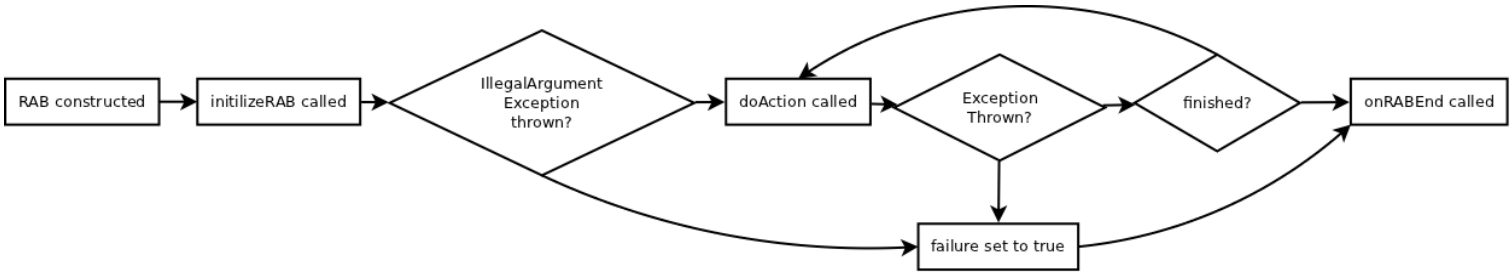
The code for this example is in the project the 'Rab programming introduction' in the files

- /src/fi/jyu/it/iog/Ubiware/documentation/PrintContainerBehavior.java
- printContainer.sapl

The example can be run using the 'Print Container.launch' run configuration delivered with the project.

1.2 The RAB lifecycle

This section provides more information about how the agent executes RABs. The lifecycle of an RAB can be graphically represented as in the following picture:



The RAB instance is created by the UBIWARE agent when it notices that its beliefs structure contains an RAB call in the general context, which this agent is allowed to execute by the policy management (When not set-up, the policy management allows all actions. - more information on policies can be found in Infrastructure guide (section Policy Agent).

A RAB call is a belief of the following form:

{sapl:I sapl:do java:<RAB class name> } sapl:configuredAs <parameters>

Where,

- <RAB class name> is a package qualified name of a class with a default constructor which derives from ReusableAtomicBehavior.
- <parameters> is sapl:null (in case of empty parameter list) or a container containing statements of the form:
  <resource> sapl:is <value>
  Where,
  - <resource> is a valid resource name
  - <value> is any valid S-APL Object. (literal, container or resource)

For example:

```
{sapl:I sapl:do java:ubiware.shared.PrintBehavior } sapl:configuredAs {
        p:print sapl:is “Text”
}
```

After creation of the RAB instance, it is scheduled on the agent thread, meaning that at some point in the future this RAB will get a chance to run in the thread of control allowed to change the agent’s beliefs’ structure.

When the agent scheduling mechanism decides that the time to perform this RAB has come, the method initializeRAB of the RAB will be called. This gives the RAB the chance to read the parameters which were provided as object in the RAB call mentioned earlier. More about parameter handling in 1.3 Parameter Handling

When parameter initialization was not successful; an IllegalParameterConfigurationException was thrown from the initilizeRAB method; the method onRABEnd will be called and this RAB ends in a failure.

If the initialization was successful, the method doAction will be called. In this method, the developer can be sure that he has exclusive access to the agent’s beliefs’ structure. This is the right place to perform the actual action the behavior must perform. Interacting with the agent is possible trough methods defined by the ReusableAtomicBehavior superclass from which all

behaviors inherit. More information on these methods can be found in the java documentation. If the doAction method throws an exception during its operation, the method onRABEnd will be called and this RAB ends in a failure.

If the execution of the doAction ended without any thrown exceptions, there are two possibilities: In the first and most common case, the onRABEnd method will be called. However, if the RAB has during its execution set the finished flag to false, the doAction method will be called again at a later point.

When programming RABs, the programmer must be aware of the fact that the methods of the RAB will be called on the agent's thread which is also responsible for manipulation of the agent beliefs' structure. This implies that the agent is not able to reason while the thread of control is in the method. This means concretely that the implementer of the RAB may not make any assumptions about reasoning on data which RABs add to the agent before the agent's live behavior has ran.

For more advanced RAB development, it is also important that if the RAB takes too much time to complete, the agent might appear irresponsive to other tasks it must perform. That is why there is a possibility to spawn separate threads to perform actions and later on put the results back to the agent's beliefs. One way is doing so using the fact that RABs can be made cyclic. To ease this task, there is a framework for thread-safety provided. This is especially useful if the creation of the threads is not in the hands of the developer (for example when using web servers), more information about threading can be found in 1.4 Multi threading and long running tasks.

1.3 Parameter Handling

Parameters passed to the RAB from the agent are accessible through the parameters object which is passed to the initializeRAB method before RAB execution. This object maintains the mapping between resources and their values specified in the RAB call.

To get a value specified, the most basic method provided is getParameterValue(Resource) which will give a String representation of the value specified by the user or null if the parameter is not specified. If a parameter is specified multiple times and the getParameterValue(Resource) method is called, an IllegalParameterConfigurationException is thrown since the implementation cannot decide which value to return.

Its multi-valued counterpart is getParameterValues(Resource), which will return a (possibly empty) Collection of Strings which are the values of the multi-valued parameter.

Besides these two basic methods, convenience methods for retrieving Strings, Container, Boolean, integer, File, and URL values are provided. Information about specific methods for retrieving this type of values can be found from the Java documentation of ubiware.core.BehaviorStartParameters.

Remark: when the agent specifies a container as a parameter to the RAB, only the ID of the container can be retrieved from the parameters object (for example by a call to getObligatedContainerID). It is possible to retrieve the actual container from the agent directly through the protected myAgent member of the ReusableAtomicBehavior class. This is however hardly ever necessary and deprecated. Instead methods provided by the ReusableAtomicBehavior super class should be used. Examples of these methods include hasBeliefsN3, removeBeliefs, …See Java documentation for ubiware.core.ReusableAtomicBehavior for more details.

1.4 Multi threading and long running tasks

When writing RABs it is important to notice that while the RAB code is running, the RAB has exclusive access to the agent's beliefs' structure. This implies that the agent is unable to continue its work (is also completely irresponsive) while the behavior's code is running or waiting for other tasks to complete. To solve this issue, the programmer must, if feasible, start a separate thread of control to handle the outstanding task and let the agent continue its work. The decision whether separate threads are needed is depending of the function of the agent. One could say that if a behavior needs more than a couple of seconds to complete, it should start a second thread.

To understand the way multi threading is implemented, it is useful to look into the internals of the platform.The agents provided by Jade and thus ubiware have one thread which executes the behaviors of the agent with a round-robin scheduling algorithm. The same thread is used to maintain the beliefs' structure of the agent. If another thread would access the beliefs of the agent concurrent with the agent's thread, the internal representation of the beliefs might get corrupted. Those inconsistencies would in general be noticed by ConcurrentModificationExceptions.

The approach used to implement thread safety uses the command pattern. Using design pattern syntax, we could say that any client (other thread) is able to create a command (ubiware.core.commands.UbiwareAgentCommand) which can be received by the UBIWARE agent. The UBIWARE agent has a behavior which will be the actual executor of the action. This behavior is scheduled in the agent and is thus executed in a thread safe way. An extension of this system is the blocking command which allows another thread to block till the actual action is executed inside the agent's thread.

On top of this framework, a concurrent agent wrapper is created. This wrapper is called RabRunnable and can be used as a normal java.lang.Runnable. The advantage of the RabRunnable is that from within the runnable, calls to the agent can be made in a thread safe manner while preserving the possibility to schedule this Runnable in whatever execution service is desired for the application.

It is important to note that the RAB will, even if a thread is created continue as if the thread does not exist. This means that the agent gets notified about a successful completion of the RAB and in particular that the content of the context containers for adding beliefs on RAB end will be added to the global context. To prevent this from happening, the developer is has to set the finished flag to false and only switch it back to true if the behavior really finished.

Example code using these features can be found in the files:

- /src/fi/jyu/it/iog/Ubiware/documentation/PerformExternalTaskBehavior.java
- /src/fi/jyu/it/iog/Ubiware/documentation/PerformLongTaskBehavior.java
- multiThread.sapl

The example can be run using the 'Multi Thread.launch' run configuration delivered with the project.