SmartResource Platform

and

Semantic Agent Programming Language (S-APL)

Developer's guide

Artem Katasonov

03.04.2007 version 1.0

Industrial Ontologies Group
University of Jyväskylä
email: artem.katasonov@jyu.fi

Contents

1 General Description

This section provides a general description of the SmartResource Platform.

The SmartResource Platform is a development framework for creating multi-agent systems. The SmartResource Platform is built on the top of the Java Agent Development Framework (JADE). JADE is a Java implementation of IEEE FIPA specifications. If you need information on FIPA's general architecture and information on how it is realized in JADE, consult FIPA documents (at http://www.fipa.org/) and JADE documentation (at http://jade.tilab.com/).

The central to the SmartResource Platform is the architecture of a SmartResource agent depicted in Figure 1. It can be seen as consisting of three layers: *Reusable Atomic Behaviors (RABs)*, *Behavior Models* corresponding to different roles the agent plays, and the *Behavior Engine*. An additional element is the storage for beliefs and goals of the agent.

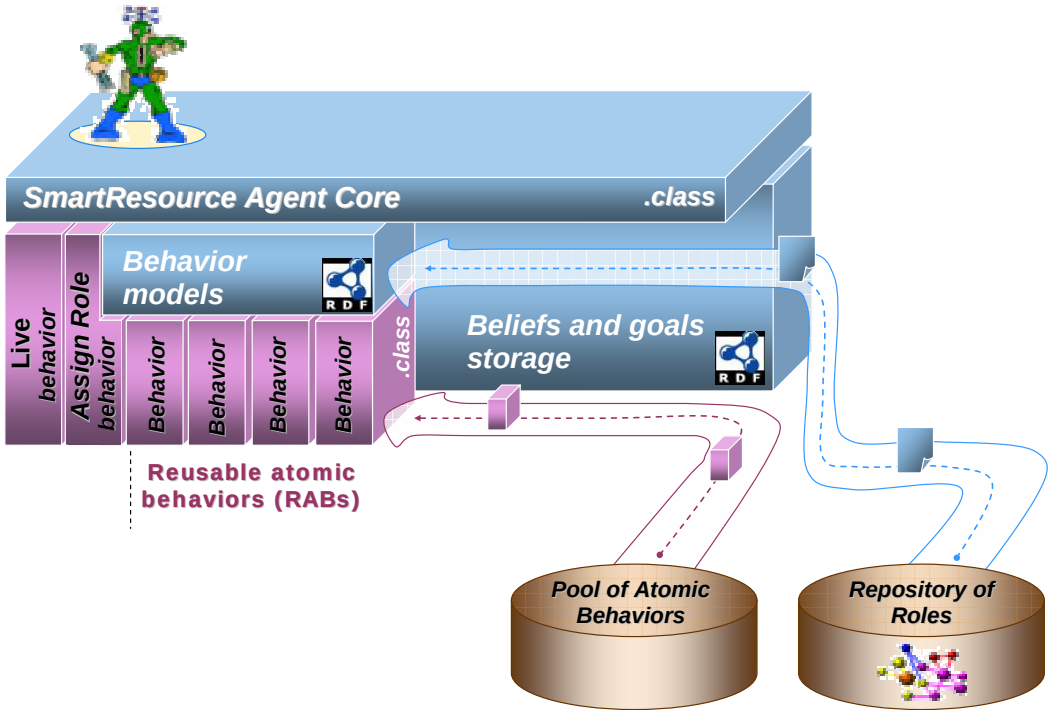

Figure 1. SmartResource agent architecture

A *belief* is a statement about the agent itself, the environment or the present situation, which is believed to be true. A *goal* is a statement about the agent itself, the environment or the situation, which is not believed to be true, but is desired to become true. Both beliefs and goals in the SmartResource Platform are based the RDF data model, i.e. any belief or goal is a subject-predicate-object triple, e.g. "John Loves Mary".

A *reusable atomic behavior (RAB)* is a piece of Java code implementing a reasonably atomic function. Therefore, RABs can be seen as the agent's *perceptors* and *actuators*. As the name implies, RABs are assumed to be reusable across different applications, different agents, different roles and different interaction scenarios. Obviously, RABs need to be parameterizable.

A description of RABs provided with the SmartResource Platform is given in Chapter 3. A description of application programming interfaces needed for developing new RABs is given in Chapter 4.

In the SmartResource Platform, the behavior of an agent is defined by the roles it plays in one or several organizations. Some examples of the possible roles: operator's agent, feeder agent, agent of the feeder N3056, fault localization service agent, ABB fault localization service agent, etc. Obviously, a general role can be played by several agents. On the other hand, one agent can (and usually does) play several roles.

A *behavior model* is document that is supposed to specify a certain organizational role, and, therefore, there is one-to-one relation between roles and behavior models. In the SmartResource Platform, behavior models are presented in a high-level rule-based language, to which we refer to as *Semantic Agent Programming Language*, with shorter forms *semantic APL* or just *S-APL*. S-APL is based on the RDF data model, i.e. the whole document can be seen as a set of subject-predicate-object triples. A behavior model consists of a set of beliefs representing the knowledge needed for playing the role and a set of behavior rules. Roughly speaking, a behavior rule specifies conditions of (and parameters for) execution of various RABs. A description of S-APL is given in Chapter 2.

The *behavior engine* is the same for all the SmartResource agents (this of course means that each agent has a copy of it). The behavior engine consists of the agent core and the two core behaviors that we named "Assign Role" and "Live".

The *AssignRole* behavior processes an S-APL document, loads specified initial beliefs and goals into the beliefs and goals storage, and parses the behavior rules (see Section 3.4.2). In addition, it registers the new role with the system Directory Facilitator agent. Note that it is recommended that if a behavior model is to specify the need of interaction with another agent, that agent should be specified by its role, not the name or another unique identifier of a particular agent. Then, DFLookup atomic behavior (see Section 3.4.1) can find with DirectoryFacilitator names of agents playing a particular role. If several agents play the role needed, the behavior model is supposed to include some rules specifying a mechanism of resolving such a situation, e.g. random select, auction, etc. Different such mechanisms can of course be assigned to resolving conflicts with respect to different roles. See some examples in Section 3.2.6.

When an agent is created it has to be given at least one behavior model to start working. Therefore, the agent's core needs to directly engage the AssignRole behavior for the model(s) specified. However, all the later invocations of AssignRole, i.e. adding new roles, are to be specified in some behavior models. Therefore, AssignRole has the duality of being a part of the behavior engine and a RAB in the same time.

The *Live* behavior implements the run-time cycle of an agent. Roughly speaking, it iterates through all the behavior rules, checks them against existing beliefs and goals, and executes the appropriate rules. Execution of a rule normally includes execution of a RAB and performing a set of additional mental actions, i.e. adding and removing some beliefs (see Section 2.3). At least at the present stage, if there are several rules that are executable, all of them are executed.

As can be seen from Figure 1, the SmartResource Platform allows agents to access behavior models from an external repository, which is assumed to be managed by the organization which "hires" the agents to enact those roles. It is done either upon startup of an agent, or later on.

As can also be seen from Figure 1, the SmartResource Platform allows agent on-demand access even of RABs. If an agent plays a role, and that role prescribes it to execute an atomic behavior that the agent is missing, the agent can download it from the repository of the organization. In a sense, the organization is able to provide not only instructions what to do, but also the tools

enabling doing that. Physically, a RAB is delivered as either *.class* file or a *.zip* file (in case when the RAB has several .class files).

The present version SmartResource Platform provides the behavior model *OntologyAgent.rdf* along with used in it RAB OntologyLookupBehavior. By starting an agent based on this model, one creates an agent that provides access to both repository of roles and pool of atomic behaviors. In the present version, the repository for such an agent is organized simply as the folder "DB_ontology\" and the pool as the folder "DB_ontology\classes\" (the structure of the packages is to be followed).

The SmartResource Platform provides the behavior model *startup.rdf*, which has to be loaded by an agent at startup in order to enable it to remotely access behavior models from an OntologyAgent. For accessing new roles in run-time, the convention described in Section 3.4.2 is to be used. For roles specified on startup of the agent, the agent's core takes care of it. In addition, startup.rdf includes the rule for engaging DFLookupBehavior. This rule is obviously needed for resolving the OntologyAgent role. However, it is also enough for any future needs of resolving roles, just the convention described in Section 3.4.1 is to be followed.

The SmartResource Platform provides also the behavior model *RABLoader.rdf*, which has to be loaded by an agent in order to enable it to remotely access atomic behaviors from an OntologyAgent. It includes rules for requesting, receiving, and (if needed) unzipping the behavior files.

An agent for the SmartResource Platform can be started in two ways. First, an agent can create another agent using CreateAgentBehavior RAB (see Section 3.3.1). Second, an agent can be started upon startup of the whole platform. In such a case, the parameters are the same as those of CreateAgentBehavior, only need to be structured according to JADE conventions:

```
<name>:smartresource.core.SmartResourceAgent(<scripts>,<roles>)
```

where <name> is the name of the agent, <scripts> is the star(*) separated list of the initial behavior models, <roles> is plus(+) separated list of the roles. Parameter <roles> is not mandatory and needed only if startup.rdf is one of the models given in <scripts>.

Below is an example of normal (minimal) Windows batch file for starting an application build using the SmartResource Platform is:

```
@setlocal

@set classpath=%classpath%;.\lib\jade\JadeLeap.jar;.\lib\sesame\rio.jar;.\lib
\sesame\openrdf-model.jar;.\lib\sesame\openrdf-util.jar

@set agent1=ontology:smartresource.core.SmartResourceAgent(DB/_ontology/Ontol
ogyAgent.rdf)

@set agent2=starter:smartresource.core.SmartResourceAgent(DB/startup.rdf,Plat
formStarter+RABLoader)

@set agents=%agent1%;%agent2%

@java jade.Boot -port 80 %agents%

@endlocal
```

Note that the *classpath* declaration needs to be extended if some special libraries are to be utilized by agents (in RABs).

There is also a third method of starting a new agent, which can be utilized in some specific cases. It is described in Section 4.6.

Technically, the SmatResource agent's core is an extension (subclass) of JADE's Agent class. A common RAB is an extension of JADE's SimpleBehavior class. As it will be explained in Chapter 4, due to technical issues related to organizing external interfacing, in addition to regular RABs, we differentiate three types of special RABs: GUI, HTTP server, and event handler (for HTTP). Conceptually, they all treated as close as possible to common RABs though.

2 Semantic Agent Programming Language (S-APL)

This chapter describes the RDF/XML syntax of the Semantic Agent Programming Language (S-APL) through its three constructs: Belief, Goal and Behavior. At the present stage, the namespace "gb**:**" is defined as "xmlns:gb="http://www.smartresource.com/rgbdf#"", and "x**:**" is defined as "xmlns:x="http://www.smartresource.com/atomic_behaviors#""

Any S-APL document should have the following basic structure:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
             xmlns:gb="http://www.smartresource.com/rgbdf#"
             xmlns:x="http://www.smartresource.com/atomic_behaviors#">
...

0..N Belief or Goal or Behavior

...
</rdf:RDF>
```

If a resource is specified without the namespace (mainly URI of beliefs, goals and behaviors), e.g. <gb:Belief rdf:about="belief1">, the default dynamically-assigned namespace is used "http://www.smartresource.com/<agent_name>/<role>/", where <role> is defined by the name of the S-APL document. An example is "http://www.smartresource.com/ontology/ OntologyAgent/belief1".

In the objects of all triples, it is possible to use two constants, which we be substituted with their values upon parsing (i.e. already in AssignRole behavior): %AgentName% - the name of the agent, %Today% - the present date in the format "dd.mm.yyyy".

As can be noticed, in defining/referring to beliefs and goals, S-APL does not use the RDF syntax, but has them as literals of the form "<subject> <predicate> <object>". The main reason for this is inability of RDF to restrict the scope of statement. In RDF, every statement is treated as a global truth. But for describing behavior rules, the statements are specifications of IF and THEN conditions, not facts. Additional reason is a wish to keep S-APL documents concise and human-readable/editable.

In S-APL, all beliefs/goals must be triples. However, at least at present stage, we do not enforce the RDF rule that only the object can be a literal while the subject and the predicate must be URIs. In other words, in S-APL beliefs/goals, the subject and the predicate can be literals as well. When using URIs, a convenient way is to utilize XML's ENTITY construct (to simulate the namespaces mechanism). For example:

```
<?xml version="1.0"?>

<!DOCTYPE rdf:RDF [
<!ENTITY  our "http://www.metso.com/ontology#">
<!ENTITY  ph "http://www.physics.org/ontology#">
]>

<rdf:RDF
...
<gb:trueIf>&our;Environment &ph;TemperatureCelcius 23</gb:trueIf>
...
```

2.1 Initial Beliefs

Element: <gb:Belief>

Specifies a belief, with which the agent (in a role) will be initialized. A belief is a statement about the agent itself, the environment or the present situation, which is believed to be true.

Fields:

Name	Meaning	Mandatory
gb:statement	The semantic statement of the belief, in the format (whitespace-separated) "<subject> <predicate> <object>". Leading and trailing whitespaces in all three elements will be trimmed off. If only one whitespace is found, the object is set to "?" meaning undefined. If no whitespace is found, both predicate and object are set to "?".	Yes

Example of usage:

```
<gb:Belief rdf:about="belief1">
      <gb:statement>MyDevice Is Feeder_ID_4583</gb:statement>
</gb:Belief>
<gb:Belief rdf:about="belief2">
      <gb:statement>I  Start  GUI</gb:statement>
</gb:Belief>
```

2.2 Initial Goals

Element: <gb:Goal>

Specifies a goal, with which the agent (in a role) will be initialized. A goal is a statement about the agent itself, the environment or the situation, which is not believed to be true, but is desired to become true.

Fields:

Name	Meaning	Mandatory
gb:statement	The semantic statement of the goal in the format as described in the previous section.	Yes

Example of usage:

```
<gb:Goal rdf:about="goal1">
      <gb:statement>I Solved Problem</gb:statement>
</gb:Goal>
```

2.3 Behavioral Rules

Element: <gb:Behavior>

Specifies a behavioral rule. The fields *trueIf*, *falseIf*, *trueIfGoalAchieved*, *achievesGoal*, and *event* describe the left side (IF) of the rule, while the rest describe the right side (THEN) of the rule.

Fields: None of the fields is mandatory. All can appear more than once, with exception of *gb:class* and *gb:event* (for those, the last occurrence is used).

Name	Meaning
gb:trueIf	Specifies a precondition, i.e. a belief that must be found in the set of the agent's beliefs to make the rule applicable. If several *trueIf* is given, they all must be found, i.e. they can be seen as connected with AND.
gb:falseIf	Specifies a negative condition, i.e. such a belief that, when found in the set of the agent's beliefs, makes the rule not applicable. If several *falseIf* is given, any of them is enough, i.e. they can be seen as connected with OR.
gb:achievesGoal	Specifies a goal that must be found in the set of the agent's goals to make the rule applicable. In a sense, it specifies an expected rational effect of the rule execution and puts a need for it as a precondition. If several *achievesGoal* is given, any of them is enough, i.e. they can be seen as connected with OR.
gb:event	Specifies an interface event (either from GUI or from HTTP server) that must be the current event to make the rule applicable. If a rule has *event* specified, it will never be executed from the normal Live cycle of the agent, but only from the interface event handling routine.
gb:trueIfGoalAchieved	Specifies a sub-goal that must be achieved before an applicable rule can be executed. If according to all the other conditions the rule is applicable and only one or more *trueIfGoalAchieved* is not fulfilled, the agent will add those to the set of its goals. In other words, the agent will try to eventually execute a rule that is applicable. If several *trueIfGoalAchieved* is given, they all need to be present in the beliefs to make the rule executable, so they can be seen as connected with AND. Also, all of them that are not present in the beliefs, will be added to the set of goals at once.
gb:addOn<X>	Specifies a belief that has to be added in the phase <X>. Possible values for <X> are: *Start* – add the belief when the rule is executed, before invoking the actual behavior (e.g. RAB). *End* – add the belief when the behavior has ended the execution.

	Success – add when and if the behavior has ended the execution in success. *Fail* - add when and if the behavior has ended the execution in failure.
gb:removeOn<X>	Specifies a belief that has to be removed in the phase <X>. Possible values for <X> are the same as above. If the specified belief contains "*" and matches several existing beliefs, all matching beliefs will be removed.
gb:class	Specifies the Java class implementing the behavior. This must be one of the following: a subclass of ReusableAtomicBehavior, a subclass of InterfaceEventHandler, a subclass of SmartResourceAgentGUI, or SmartResourceAgentServer or its subclass.
x:<anything>	A parameter that is to be passed to the instance of behavior, and has some meaning in the context of that behavior.

Note that explicit adding or removing of goals is not supported. A goal can only be added if it appears in *trueIfGoalAchieved* of an applicable rule (and not yet in the set of goals), and removed only when either (1) a rule is executed having it among its *achievesGoal* or (2) in the beginning or a Live cycle, it is found in the set of beliefs.

Note also that an additional implicit condition for whether a rule is executable is presence of the specified Java class. If it is not found, the rule is considered as not executable, so neither beliefs are modified nor goals are removed.

In the statements, "*" can be used with the meaning of "anything", and "*<var>*" can be used as variable. If variables are used, the rule is applicable/executable if the beliefs and goals of the agent provide at least one possible binding of the values. If several bindings are possible, the first found is taken. The left part of the rule is processed in the following order: *event*, *trueIf*, *achievesGoal*, *trueIfGoalAchieved*, *falseIf*. This defines the order in which the variables are bind. The possible set of values for a variable is searched when the variable is first encountered. After that, values from this set can only get filtered out but no new ones can be added.

All the fields of a rule are passed as the start parameters to the instance of the behavior, not only *x:<anything>*. Therefore, if needed, the behavior can have access to the context of its execution, e.g. trueIf, addOnStart, etc. Note though that, in all the start parameters, the variables are substituted with their values.

Example of usage: Given that the date is 8 of March, if the agent knows a woman and knows something that she likes, start (a non-existing) GiftingBehavior to gift her that thing. A sub-goal of this is to buy the needed thing (handled in the rule behavior2). FalseIf statements are used to prevent the behavior to be executed twice. The belief "I Gifting <X>" is added as soon as the rule is executed (note, this happens after the sub-goal is achieved), and removed as soon as GiftingBehavior ends (regardless of the result). If the result is success, belief "I Gifted <X>" is added.

```
<gb:Behavior rdf:about="behavior1">
      <gb:class>smartresource.non_existing.GiftingBehavior</gb:class>
      <gb:trueIf>Date Is 08.03</gb:trueIf>
      <gb:trueIf>I Know *X*</gb:trueIf>
      <gb:trueIf>*X* Is Woman</gb:trueIf>
      <gb:trueIf>*X* Likes *thing*</gb:trueIf>
```

```
      <gb:falseIf>I Gifting *X*</gb:falseIf>
      <gb:falseIf>I Gifted *X*</gb:falseIf>
      <gb:trueIfGoalAchieved>I Bought *thing*</gb:trueIfGoalAchieved>
      <gb:addOnStart>I Gifting *X*</gb:addOnStart>
      <x:receiver>*X*</x:receiver>
      <x:object>*thing*</x:object>
      <gb:removeOnEnd>I Gifting *X*</gb:removeOnEnd>
      <gb:addOnSuccess>I Gifted *X*</gb:addOnSuccess>
</gb:Behavior>

<gb:Behavior rdf:about="behavior2">
      <gb:class>smartresource.non_existing.BuyingBehavior</gb:class>
      <gb:achievesGoal>I Bought *thing*</gb:achievesGoal>
      <x:object>*thing*</x:object>
      <gb:addOnSuccess>I Bought *thing*</gb:addOnSuccess>
</gb:Behavior>
```

3 Available Reusable Atomic Behaviors (RABs)

This chapter describes RABs available at the moment. Note that "outputs" specified do not include the print outs of the stack traces of possible exceptions. "Libraries" specify libraries needed in addition to normal Java SE and JADE ones. At the present stage, the namespace "x**:**" is defined as "xmlns:x="http://www.smartresource.com/atomic_behaviors#""

3.1 Embedded actions

There a couple of behaviors, which of course could have been implemented with separate pieces of Java code, but are so simple that we decided to embed them into the Live behavior. Reference to such behavior is done as @<command>.

3.1.1 @Print

Action: Prints to the screen (and to the log) specified text. Text is printed preceded by "[<present time>] <agent's name>: ".

Inputs:

Name	Meaning	Mandatory	Default
x:print	The text to print.	Yes	

Outputs: the text is printed.

Example of usage: Print "Job is done!" when no "I Process <X>" belief is left in the agent's beliefs.

```
<gb:Behavior rdf:about="behavior1">
      <gb:class>@Print</gb:class>
      <gb:trueIf>I Have Data</gb:trueIf>
      <gb:falseIf>I Process *</gb:falseIf>
      <gb:removeOnStart>I Have Data</gb:removeOnStart>
      <x:print>Job is done!</x:print>
</gb:Behavior>
```

3.1.2 @ModifyGUI

Action: Sends a request to the active GUI to perform some actions, mainly related to modifying GUI (enabling/disabling elements, putting text to labels, etc.).

Inputs: depend on specific GUI.

Outputs: none

Example of usage: Given that the GUI is SeveralButtonsGUI (see corresponding section), disable the button that was just clicked.

```
<gb:Behavior rdf:about="gui_event1">
      <gb:class>@ModifyGUI</gb:class>
      <gb:event>User ClickedButton *x*</gb:event>
      <x:modify>Disable</x:modify>
      <x:target>*x*</x:target>
</gb:Behavior>
```

3.2 Local actions

This section describes behaviors that do not involve any other agents, i.e. have no social component.

3.2.1 DelayerBehavior

Full name: smartresource.shared.DelayerBehavior

Action: Waits for a specified period of time, after that ends in success.

Inputs:

Name	Meaning	Mandatory	Default
x:delay	Time to wait in milliseconds.	Yes	

Outputs: none.

Example of usage: Wait 5 seconds, then add belief “I Send Alert”.

```
<gb:Behavior rdf:about="behavior1">
      <gb:class>smartresource.shared.DelayerBehavior</gb:class>
      <gb:trueIf>I Start Delayer</gb:trueIf>
      <gb:removeOnStart>I Start Delayer</gb:removeOnStart>
      <x:delay>5000</x:delay>
      <gb:addOnSuccess>I Send Alert</gb:addOnSuccess>
</gb:Behavior>
```

Libraries needed: none.

3.2.2 ExternalAppStarterBehavior

Full name: smartresource.shared.ExternalAppStarterBehavior

Action: Executes a command of the operating system. Primarily, is intended for starting external software applications. If the command is successfully executed (it does not necessarily implies that the application is started), ends in success. If an exception occurs, ends in failure.

Inputs:

Name	Meaning	Mandatory	Default
x:startExternal	Command to execute.	Yes	

Outputs: none.

Example of usage: Execute the text editor Notepad and open in it file “Dog.txt”.

```
<gb:Belief rdf:about="belief1">
      <gb:statement>I Have Dog</gb:statement>
</gb:Belief>
<gb:Behavior rdf:about="behavior1">
      <gb:class>smartresource.shared.ExternalAppStarterBehavior</gb:class>
      <gb:trueIf>I Have *who*</gb:trueIf>
      <gb:removeOnStart>I Have *who*</gb:removeOnStart>
      <x:startExternal>notepad.exe *who*.txt</x:startExternal>
</gb:Behavior>
```

Libraries needed: none.

3.2.3 UnzipperBehavior

Full name: smartresource.shared.UnzipperBehavior

Action: Unzips a ZIP archive. If parameter “delete” is equal to “true”, deletes the original file. If an I/O exception occurs, ends in failure. Otherwise, ends in success.

Inputs:

Name	Meaning	Mandatory	Default
x:input	Name of ZIP file.	Yes	
x:delete	Whether to delete the ZIP file after unzipping the contents.	No	“false”

Outputs: On success, prints to the screen “<input> is unzipped”.

Example of usage: Unzip ZIP archive whose name is defined through belief “I Unzip *zip*”, then delete the archive.

```
<gb:Behavior rdf:about="behavior1">
      <gb:class>smartresource.shared.UnzipperBehavior</gb:class>
      <gb:trueIf>I Unzip *zip*</gb:trueIf>
      <gb:removeOnStart>I Unzip *zip*</gb:removeOnStart>
      <x:input>*zip*</x:input>
      <x:delete>true</x:delete>
</gb:Behavior>
```

Libraries needed: none.

3.2.4 HttpDataFetcherBehavior

Full name: smartresource.shared.HttpDataFetcherBehavior

Action: Downloads a document (e.g. an HTML page) from an Internet server and saves it to a file. The download process is done in a separate thread, so does not block the agent. If everything is fine, ends in success. If there is a problem with either fetching or saving, ends in failure.

Inputs:

Name	Meaning	Mandatory	Default
x:url	URL of the document.	Yes	
x:saveTo	Name of the file to save the document to.	Yes	

Outputs: Prints to the screen "Failed to retrieve <url>" if download failed.

Example of usage: Download from the site of Finnish Meteorological Institute the page with the present weather information and forecast for Jyväskylä.

```
<gb:Behavior rdf:about="behavior1">
      <gb:class>smartresource.shared.HttpDataFetcherBehavior</gb:class>
      <gb:trueIf>*requestID* request GetData</gb:trueIf>
      <gb:falseIf>I Fetch *requestID*</gb:falseIf>
      <gb:addOnStart>I Fetch *requestID*</gb:addOnStart>
      <x:uri>http://www.fmi.fi/saa/paikalli.html?Keywords=&amp;param=T&amp;ne
ito=1&amp;kunta=Jyv%E4skyl%E4</x:uri>
      <x:saveTo>DB/%AgentName%/received/*requestID*.htm</x:saveTo>
      <gb:addOnSuccess>I Fetched *requestID*</gb:addOnSuccess>
</gb:Behavior>
```

Libraries needed: none.

3.2.5 EmailSenderBehavior

Full name: smartresource.shared.EmailSenderBehavior

Action: Sends an email. The sending process is done in a separate thread, so does not block the agent. If the server responds with error 451 (local error in processing) or 452 (insufficient system storage), and parameter "waitOnFail" is given, waits specified time and tries again. If email is sent, ends in success. Ends in failure if an exception other than error 451 or 452 occurs, or on any exception if "waitOnFail" is not given. Also ends in failure if parameter "to" is not given or empty.

Inputs:

Name	Meaning	Mandatory	Default
x:smtp	SMTP server to use for sending email.	Yes	
x:to	Addresses to which the email to be sent, divided by commas.	Yes	
x:from	Address to put as the outgoing address.	No	"no@addr. ess"
x:cc	Addresses to which a copy to be sent.	No	""

x:bcc	Addresses to which a blind copy is to be sent.	No	“”
x:subject	Subject of the email.	No	“”
x:message	The content of the email.	No	“”
x:attach	Names of the files to attach to the email, divided by comma.	No	“”
x:contentType	The type of the email content, e.g. “text/plain” or “text/html”.	No	“text/plain”
x:waitOnFail	Time to wait in milliseconds before next attempt of sending.	No	

Outputs: On success, prints to the screen “Email to <to> is sent”. On failure, prints “Sending email to <to> is failed!”. In case of error 451 or 452, prints to the screen “Email sending error. Waiting <waitOnFail/1000> sec”.

Example of usage: Send email through smtp.jyu.fi server. If a problem, wait 10 seconds and try again.

```
<gb:Behavior rdf:about="behavior2">
      <gb:class>smartresource.shared.EmailSenderBehavior</gb:class>
      <gb:trueIf>I SendEmail *text*</gb:trueIf>
      <gb:removeOnStart>I SendEmail *text*</gb:removeOnStart>
      <gb:addOnStart>I Send Email</gb:addOnStart>
      <x:smtp>smtp.jyu.fi</x:smtp>
      <x:to>akataso@cc.jyu.fi</x:to>
      <x:from>artem.katasonov@jyu.fi</x:from>
      <x:subject>test</x:subject>
      <x:message>*text*</x:message>
      <x:waitOnFail>10000</x:waitOnFail>
      <gb:removeOnEnd>I Send Email</gb:removeOnEnd>
</gb:Behavior>
```

Libraries needed: JavaMail (mail.jar) and JavaBeans Activation Framework (activation.jar). Both are included into J2EE, but need to be acquired separately for J2SE. Provided with the SmartResource platform.

3.2.6 SimpleSelectBehavior

Full name: smartresource.shared.SimpleSelectBehavior

Action: Performs selection among several alternatives. Ends in failure if no single alternative is found.

Inputs:

Name	Meaning	Mandatory	Default
x:input (>1 is allowed)	Input beliefs, must contain variables.	Yes	

x:output	Output belief, must refer to variables defined in "x:input".	Yes	
x:criterion	Selection criterion: "random", "minimum" or "maximum".	Yes	
x:evaluate	Formula to evaluate in case of "minimum" or "maximum", must refer to variables defined in "x:input".	Yes if the criterion is minimum or maximum	

Outputs: Prints to the screen "var1=value1 var2=value2 … is selected as <criterion> <formula>".

Example of usage: Select an agent randomly among several playing the same role.

```
<gb:Behavior rdf:about="behavior1">
      <gb:class>smartresource.shared.SimpleSelectBehavior</gb:class>
      <gb:trueIf>*role* RoleHasToBe Resolved</gb:trueIf>
      <gb:falseIf>LocateAgent RoleHasToBe Resolved</gb:falseIf>
      <gb:removeOnStart>*role* RoleHasToBe Resolved</gb:removeOnStart>
      <x:input>*role* RoleIsPlayedBy *agent*</x:input>
      <x:criterion>random</x:criterion>
      <x:output>*role* ForRoleIsSelected *agent*</x:output>
      <gb:removeOnSuccess>*role* RoleIsPlayedBy *</gb:removeOnSuccess>
</gb:Behavior>
```

Example of usage: Select a LocateAgent which provides a better value (minimum) according to formula "Price + ResponseTime*200".

```
<gb:Behavior rdf:about="behavior2">
      <gb:class>smartresource.shared.SimpleSelectBehavior</gb:class>
      <gb:trueIf>LocateAgent RoleHasToBe Resolved</gb:trueIf>
      <gb:removeOnStart>LocateAgent RoleHasToBe Resolved</gb:removeOnStart>
      <x:input>*agent* Price *price*</x:input>
      <x:input>*agent* ResponseTime *speed*</x:input>
      <x:criterion>minimum</x:criterion>
      <x:evaluate>*price* *speed* 200 * +</x:evaluate>
      <x:output>LocateAgent ForRoleIsSelected *agent*</x:output>
</gb:Behavior>
```

Libraries needed: none.

3.2.7 ExcelReaderBehavior

Full name: smartresource.shared.ExcelReaderBehavior

Action: Reads data from a worksheet of a Microsoft Excel file. Takes the first worksheet only. Considers only columns that have something in the first row and considers that as a label. For every row from the second on, generates a set of beliefs using those labels as predicates (see below for details). If I/O exception, ends in failure, otherwise ends in success.

Inputs:

Name	Meaning	Mandatory	Default
x:input	Name of Excel file	Yes	

Outputs: For every row from the second on, generates a row ID based on system time when process is started and number of the row in the file. Adds following beliefs:

<input> Row <rowID>

<row ID> <label> <value> - for every column that has something (<label>) in the first row, value is the value of the cell

Example of usage: Read data from file.xls, then do something based on that.

```
<gb:Behavior rdf:about="behavior1">
      <gb:class>smartresource.shared.ExcelReaderBehavior</gb:class>
      <gb:trueIf>I Read Excel</gb:trueIf>
      <gb:removeOnStart>I Read Excel</gb:removeOnStart>
      <x:input>DB/%AgentName%/file.xls</x:input>
</gb:Behavior>

<gb:Behavior rdf:about="behavior2">
...
      <gb:trueIf>DB/%AgentName%/file.xls Row *id*</gb:trueIf>
      <gb:trueIf>*id* Name *name*</gb:trueIf>
      <gb:trueIf>*id* Email *email*</gb:trueIf>
      <gb:trueIf>*id* Saldo *saldo*</gb:trueIf>
...
</gb:Behavior>
```

Libraries needed: Jakarta POI by Apache (poi.jar). Acquired from http://jakarta.apache.org/poi/. Provided with the SmartResource platform.

3.3 Inter-agent actions

This section describes behaviors that involve other agents, such as communicative actions.

3.3.1 CreateAgentBehavior

Full name: smartresource.shared.CreateAgentBehavior

Action: Creates a new agent in the same container.

Inputs:

Name	Meaning	Mandatory	Default
x:name	The name of the agent.	Yes	
x:scripts	Scripts of the agent. Star(*) separated.	Yes	
x:roles	Roles of the agent. Plus(+) separated. Needed if	No	

	startup.rdf model is used.		

Outputs: none.

Example of usage: Start an agent based on beliefs like “operator Is OperatorAgent” and using “startup.rdf” script enabling accessing actual role scripts from an OntologyAgent.

```
<gb:Behavior rdf:about="behavior1">
      <gb:class>smartresource.shared.CreateAgentBehavior</gb:class>
      <gb:trueIf>*name* Is *roles*</gb:trueIf>
      <gb:removeOnStart>*name* Is *roles*</gb:removeOnStart>
      <x:name>*name*</x:name>
      <x:scripts>DB/startup.rdf</x:scripts>
      <x:roles>*roles*</x:roles>
</gb:Behavior>
```

Libraries needed: none.

3.3.2 RequestSenderBehavior

Full name: smartresource.shared.RequestSenderBehavior

Action: Sends a message to another agent on the same platform. Always ends in success.

Inputs:

Name	Meaning	Mandatory	Default
x:receiver	Name of the agent to send message to.	Yes	
x:request	Content of the message.	Yes	
x:performative	ACL performative, only “request” and “inform” are used.	No	“request”
x:conversationID	ID of the converstation. If not given, new is generated based on current system time.	No	
x:parameter	Additional parameter for the request to be included as a user defined parameter "requestParameter".	No	
x:addBeliefs	Whether the agent is to have memory of sending this message.	No	“true”

Outputs: If “addBeliefs” is equal to “true”, a request ID is generated based on current system time and the following beliefs are added:

<request ID> conversationID <conversationID>

<request ID> performative <performative>

<request ID> receiver <receiver>

<request ID> request <request>

<request ID> parameter <parameter>

Example of usage: Request data in KML format from a selected NetworkAgent by sending request “GetData” with parameter “kml”.

```
<gb:Behavior rdf:about="behavior1.2">
      <gb:class>smartresource.shared.RequestSenderBehavior</gb:class>
      <gb:trueIf>I Request NetDataKML</gb:trueIf>
      <gb:trueIfGoalAchieved>NetworkAgent Is *name*</gb:trueIfGoalAchieved>
      <gb:removeOnStart>I Request NetDataKML</gb:removeOnStart>
      <x:receiver>*name*</x:receiver>
      <x:request>GetData</x:request>
      <x:parameter>kml</x:parameter>
      <gb:addOnSuccess>I Receive NetData</gb:addOnSuccess>
</gb:Behavior>
```

Libraries needed: none.

3.3.3 RequestReceiverBehavior

Full name: smartresource.shared.RequestReceiverBehavior

Action: Listens for incoming messages matching defined parameters (or all messages if none given). If neither “waitOnlyFirst” nor “maxWait” is given, does this forever. Upon receiving a matching message, generates beliefs about this (only if “addBeliefs” is equal to “true”) and imitates ending in success (performs addOnSuccess, etc.). If “maxWait” is given, when the specified period expires, ends in failure.

Inputs:

Name	Meaning	Mandatory	Default
x:matchRequest	Match the content of the message.	No	
x:matchConversationID	Match conversation ID.	No	
x:matchPerformative	Match ACL performative, only “request” and “inform” are used.	No	
x:waitOnlyFirst	Whether to end when the first message is received.	No	“false”
x:maxWait	Time in milliseconds after which receiver ends (never ends, if not given).	No	
x:addBeliefs	Whether the agent is to have memory of receiving requests.	No	“true”

Outputs: Prints to the screen “"<request> <parameter>" from <sender> received”. If “addBeliefs” is equal to “true”, a request ID is generated based on current time and the following beliefs are added (objects come from the message):

<request ID> conversationID <conversationID>

<request ID> performative <performative>

<request ID> sender <sender>

<request ID> request <content>

<request ID> parameter <parameter>

Example of usage: Receive “GetData” requests for 1 minute. Upon every received request, do not add detailed beliefs but add belief “I Received Something”. After 1 minute, end and add belief “I Ended Receiving”.

```
<gb:Behavior rdf:about="behavior1">
      <gb:class>smartresource.shared.RequestReceiverBehavior</gb:class>
      <gb:trueIf>I Start RequestReceiver</gb:trueIf>
      <gb:removeOnStart>I Start RequestReceiver</gb:removeOnStart>
      <x:matchRequest>GetData</x:matchRequest>
      <x:matchPerformative>request</x:matchPerformative>
      <x:addBeliefs>false</x:addBeliefs>
      <x:maxWait>60000</x:maxWait>
      <gb:addOnSuccess>I Received Something</gb:addOnSuccess>
      <gb:addOnFail>I Ended Receiving</gb:addOnFail>
</gb:Behavior>
```

Libraries needed: none.

3.3.4 ResponseReceiverBehavior

Full name: smartresource.shared.ResponseReceiverBehavior

Action: Listens for a response to a request sent earlier. When received, adds belief, for which the template is specified in “saveTo”, filling all the undefined (“?”) parts (subject, predicate, object) of the template with the content of the received message, and ends in success. If “maxWait” is given, if response is not received within specified time, ends in failure.

Inputs:

Name	Meaning	Mandatory	Default
x:conversationID	ID of the conversation to match (usually generated by a RequestSenderBehavior).	Yes	
x:saveTo	Template of belief to add.	Yes	
x:maxWait	Time in milliseconds to wait (wait forever, if not given).	No	
x:addBeliefs	Whether the agent is to have memory of receiving the response.	No	“false”

Outputs: Print to the screen “"<message>" from <sender> received”. Adds belief, for which the template is specified in “saveTo”, filling all the undefined parts with the content of the message (see example). ”. If “addBeliefs” is equal to “true”, a request ID is generated based on current time and the following beliefs are added (objects come from the message):

<request ID> conversationID <conversationID>

<request ID> sender <sender>

<request ID> inform<request>

Example of usage: Assuming that a request with parameter "Price" was earlier sent to agent "loc_service", wait maximum 5 seconds for a response. Assuming that the response is "10", add belief "loc_service Price 10".

```
<gb:Behavior rdf:about="behavior1">
      <gb:class>smartresource.shared.ResponseReceiverBehavior</gb:class>
      <gb:trueIf>*name* IsAsked *param*</gb:trueIf>
      <gb:trueIf>*requestID* receiver *name*</gb:trueIf>
      <gb:trueIf>*requestID* parameter *param*</gb:trueIf>
      <gb:trueIf>*requestID* conversationID *convID*</gb:trueIf>
      <gb:falseIf>*name* Sends *param*</gb:falseIf>
      <gb:removeOnStart>*name* IsAsked *param*</gb:removeOnStart>
      <gb:addOnStart>*name* Sends *param*</gb:addOnStart>
      <x:conversationID>*convID*</x:conversationID>
      <x:saveTo>*name* *param* ?</x:saveTo>
      <x:maxWait>5000</x:maxWait>
      <gb:removeOnEnd>*requestID* * *</gb:removeOnEnd>
      <gb:removeOnEnd>*name* Sends *param*</gb:removeOnEnd>
</gb:Behavior>
```

Libraries needed: none.

3.3.5 DataSenderBehavior

Full name: smartresource.shared.DataSenderBehavior

Action: Sends to another agent on the same platform a message containing data from a file. This is usually done in response to a request. Puts the extension of the file into the "ontology" field of ACL message. Able of sending both textual and binary (.class and .zip) files. If there is I/O exception, ends in failure. Otherwise, ends in success.

Inputs:

Name	Meaning	Mandatory	Default
x:receiver	Name of an agent to send message to.	Yes	
x:loadFrom	File from where the data is to be loaded.	Yes	
x:conversationID	ID of the converstation. If not given, new is generated based on current system time.	No	
x:addBeliefs	Whether the agent is to have memory of sending this message.	No	"false"

Outputs: If "addBeliefs" is equal to "true", a request ID is generated based on current system time and the following beliefs are added:

<request ID> conversationID <conversationID>

<request ID> receiver <receiver>

<request ID> inform <loadFrom>

<request ID> ontology <extension of loadFrom>

Example of usage: Assuming there was a request, send data.kml in response.

```
<gb:Behavior rdf:about="behavior2">
      <gb:class>smartresource.shared.DataSenderBehavior</gb:class>
      <gb:trueIf>*requestID* sender *requestor*</gb:trueIf>
      <gb:trueIf>*requestID* conversationID *convID*</gb:trueIf>
      <gb:falseIf>I Respond *requestID*</gb:falseIf>
      <gb:addOnStart>I Respond *requestID*</gb:addOnStart>
      <x:receiver>*requestor*</x:receiver>
      <x:conversationID>*convID*</x:conversationID>
      <x:loadFrom>DB/%AgentName%/data.kml</x:loadFrom>
      <gb:removeOnSuccess>* * *requestID*</gb:removeOnSuccess>
      <gb:removeOnSuccess>*requestID* * *</gb:removeOnSuccess>
</gb:Behavior>
```

Libraries needed: none.

3.3.6 DataReceiverBehavior

Full name: smartresource.shared.DataReceiverBehavior

Action: Listens for a response to a request sent earlier, which is to be data of a known type. When received, saves the content of received message to a file, and ends in success. If "maxWait" is given, if response is not received within specified time, ends in failure.

Inputs:

Name	Meaning	Mandatory	Default
x:conversationID	ID of the conversation to match (usually generated by a RequestSenderBehavior).	Yes	
x:saveTo	File to save data to.	Yes	
x:datatype	List of accepted datatypes, separated by whitespaces.	Yes	
x:maxWait	Time in milliseconds to wait (wait forever, if not given).	No	
x:addBeliefs	Whether the agent is to have memory of receiving the response.	No	"false"

Outputs: Prints to the screen "<datatype> from <sender> received". Writes data to the file: If filename does not have extension, the received one ("ontology" field) is added. If "addBeliefs" is equal to "true", a request ID is generated based on current time and the following beliefs are added (objects except "saveTo" come from the message):

<request ID> conversationID <conversationID>

<request ID> sender <sender>

<request ID> inform <saveTo>

<request ID> ontology <ontology>

Example of usage: Assuming a request for data was earlier sent (as in the example to RequestSenderBehavior), wait for response.

```
<gb:Behavior rdf:about="behavior1.3">
      <gb:class>smartresource.shared.DataReceiverBehavior</gb:class>
      <gb:trueIf>I Receive NetData</gb:trueIf>
      <gb:trueIf>NetworkAgent ForRoleIsSelected *name*</gb:trueIf>
      <gb:trueIf>*requestID* receiver *name*</gb:trueIf>
      <gb:trueIf>*requestID* conversationID *convID*</gb:trueIf>
      <gb:trueIf>*requestID* parameter *datatype*</gb:trueIf>
      <gb:removeOnStart>I Receive NetData</gb:removeOnStart>
      <x:conversationID>*convID*</x:conversationID>
      <x:saveTo>DB/%AgentName%/received/network.*datatype*</x:saveTo>
      <x:datatype>.*datatype*</x:datatype>
      <gb:removeOnSuccess>*requestID* * *</gb:removeOnSuccess>
      <gb:addOnSuccess>I Have NetData</gb:addOnSuccess>
</gb:Behavior>
```

Libraries needed: none.

3.3.7 SecurityCheckBehavior

Full name: smartresource.shared.SecurityCheckBehavior

Action: Checks with Directory Facilitator whether the agent in question plays a role that would authorize it for something. Ends in success if authorization is granted, ends in failure otherwise. If no authorized roles are given, nobody passes.

Inputs:

Name	Meaning	Mandatory	Default
x:agent	Name of the agent.	Yes	
x:authorized (>1 is allowed)	Authorized roles.	No	

Outputs: Prints to the screen "<agent> is authorized as an <role>" or "<agent> has not been authorized".

Example of usage: Assuming the agent received a request to start a new role, perform check whether the requestor is an OperatorAgent, and, if so, add belief "I MissingScript *role*" to start the procedure of requesting and loading the needed script.

```
<gb:Behavior rdf:about="behavior2.3">
      <gb:class>smartresource.shared.SecurityCheckBehavior</gb:class>
      <gb:trueIf>*requestID* request LoadRole</gb:trueIf>
      <gb:trueIf>*requestID* parameter *role*</gb:trueIf>
      <gb:trueIf>*requestID* sender *sender*</gb:trueIf>
```

```
      <gb:falseIf>I Handle *requestID*</gb:falseIf>
      <gb:addOnStart>I Handle *requestID*</gb:addOnStart>
      <x:agent>*sender*</x:agent>
      <x:authorized>OperatorAgent</x:authorized>
      <gb:addOnSuccess>I MissingScript *role*</gb:addOnSuccess>
      <gb:removeOnEnd>* * *requestID*</gb:removeOnEnd>
      <gb:removeOnEnd>*requestID* * *</gb:removeOnEnd>
</gb:Behavior>
```

Libraries needed: none.

3.4 Core actions

This section describes behaviors which do not normally need to be used directly, because rules that trigger them are provided in the *startup.rdf* script. Instead of using directly, a certain beliefs or goals are to be used to initiate needed actions.

3.4.1 DFLookupBehavior

Full name: smartresource.core.behaviors.DFLookupBehavior

Action: Finds with the Directory Facilitator names of agents playing a particular role. Ends in failure only if an exception occurs, otherwise ends in success regardless of the search result.

Inputs:

Name	Meaning	Mandatory	Default
x:search	Role to search for.	Yes	
x:addIfNone	Belief to add if no agent is found.	No	
x:addIfOne	Belief to add if exactly one is found. The name of the agent is put into the object of the statement.	Yes	
x:storeIfSeveral	Beliefs to add if several are found (one per agent). The names of the agents are put into the objects of the statements.	Yes	
x:addIfSeveral (>1 is allowed)	Additional beliefs to add if several are found (to trigger resolving actions).	No	

Outputs: In case when no agent is found, prints to the screen “<search> is not found in DF”.

Example of usage: As in startup.rdf.

```
<gb:Behavior rdf:about="behaviorDF">
      <gb:class>smartresource.core.behaviors.DFLookupBehavior</gb:class>
      <gb:achievesGoal>*role* ForRoleIsSelected *</gb:achievesGoal>
      <x:search>*role*</x:search>
      <x:addIfNone>*role* RoleIsPlayedBy Noone</x:addIfNone>
      <x:addIfOne>*role* ForRoleIsSelected ?</x:addIfOne>
      <x:addIfSeveral>*role* RoleHasToBe Resolved</x:addIfSeveral>
      <x:storeIfSeveral>*role* RoleIsPlayedBy ?</x:storeIfSeveral>
```

```
</gb:Behavior>
```

Example of usage: To trigger this behavior, given that startup.rdf is loaded.

```
<gb:Behavior rdf:about="behavior1">
...
      <gb:trueIfGoalAchieved>OperatorAgent ForRoleIsSelected *name*</gb:trueI
fGoalAchieved>
...
</gb:Behavior>
```

Libraries needed: none.

3.4.2 AssignRoleBehavior

Full name: smartresource.core.AssignRoleBehavior

Action: Parses a script and registers the new role with Directory Facilitator. Ends in failure if an exception occurs.

Inputs:

Name	Meaning	Mandatory	Default
x:input	Name of script file.	Yes	
x:startup	Whether this is startup, not really a role (not to register with DF).	No	“false”
x:emptyBeliefs	Whether to remove all the old beliefs.	No	“false”
x:emptyGoals	Whether to remove all the old goals.	No	“false”
x:emptyRules	Whether to remove all the old behavioral rules.	No	“false”

Outputs: Prints to the screen “<input> role is assigned”.

Example of usage: As in startup.rdf.

```
<gb:Behavior rdf:about="behavior1">
      <gb:class>smartresource.core.AssignRoleBehavior</gb:class>
      <gb:trueIf>I MissingScript *role_script*</gb:trueIf>
      <gb:trueIf>*requestID* parameter *role_script*</gb:trueIf>
      <gb:trueIf>I Received *role_script*</gb:trueIf>
      <gb:falseIf>I Parse *role_script*</gb:falseIf>
      <gb:addOnStart>I Parse *role_script*</gb:addOnStart>
      <x:input>DB/%AgentName%/received/*role_script*.rdf</x:input>
      <x:emptyBeliefs>false</x:emptyBeliefs>
      <x:emptyGoals>false</x:emptyGoals>
      <x:emptyRules>false</x:emptyRules>
      <gb:removeOnSuccess>I * *role_script*</gb:removeOnSuccess>
      <gb:removeOnSuccess>*requestID* * *</gb:removeOnSuccess>
      <gb:addOnSuccess>I PlayRole *role_script*</gb:addOnSuccess>
</gb:Behavior>
```

Example of usage: To trigger the process of requesting the script and assigning the role, given that startup.rdf is loaded.

```
<gb:Behavior rdf:about="behavior1">
...
      <gb:addOnSuccess>I MissingScript *role*</gb:addOnSuccess>
...
</gb:Behavior>
```

Libraries needed: none.

3.5 Graphical User Interfaces (GUI)

This section describes simple reusable GUIs.

3.5.1 SeveralButtonsGUI

Full name: smartresource.shared.gui.SeveralButtonsGUI

Action: Gives a very simple interface with N buttons and possibility to enable and disable those buttons. When the user clicks on a button, a predefined event "User ClickedButton <n>" is generated, where <n> is the consecutive number of the button {1..N}.

Inputs for construct:

Name	Meaning	Mandatory	Default
x:button (>1 is allowed)	Create a button with given text.	No	
x:title	The title of the window.	No	""
x:state	The state to start in: "normal" or "minimized".	No	"normal"

The inputs *x:title* and *x:state* are actually handled in the super-class SmartResourceAgentGUI, and therefore apply to any GUI.

Inputs for modify:

Name	Meaning	Mandatory	Default
x:modify	Action: "Disable" or "Enable"	Yes	
x:target	The consecutive number {1..N}of a button to disable or enable.	Yes	

Outputs: Generates "User ClickedButton <n>" events.

Example of usage: Create GUI with 3 buttons. Disable the button that was clicked.

```
<gb:Behavior rdf:about="start_gui">
      <gb:class>smartresource.shared.gui.SeveralButtonsGUI</gb:class>
      <gb:trueIf>I Start GUI</gb:trueIf>
      <gb:removeOnStart>I Start GUI</gb:removeOnStart>
      <x:title>Example</x:title>
      <x:button>Click me</x:button>
```

```
      <x:button>Click me too</x:button>
      <x:button>Click also me</x:button>
      <x:state>minimized</x:state>
</gb:Behavior>

<gb:Behavior rdf:about="gui_event1">
      <gb:class>@ModifyGUI</gb:class>
      <gb:event>User ClickedButton *x*</gb:event>
      <x:modify>Disable</x:modify>
      <x:target>*x*</x:target>
</gb:Behavior>
```

Libraries needed: none.

4 Application Programming Interfaces

This chapter provides details of Java classes needed when developing Reusable Atomic Behaviors or other assets to be used by a SmartResource Agent.

4.1 Reusable Atomic Behavior (RAB)

Full name: smartresource.core.ReusableAtomicBehavior

Extends: jade.core.behaviours.SimpleBehaviour

In order to create a new RAB, the programmer needs to make a class that extends *ReusableAtomicBehavior*, and to implement its *action()* method. No constructor is needed; if required, initializing actions are done in *onStart()*.

JADE methods to be implemented in subclasses:

	Name	Meaning	Mandatory
public void	*action()*	Body of the behavior. Called at least once when the behavior is executed, and after that repeatedly while *finished = false*.	Yes
public void	*onStart()*	Method for initializing behavior. Called only once: when the behavior is executed, before calling *action()*.	No

JADE methods already implemented:

	Name	Meaning
public void	*done()*	Used by JADE to check after every execution of *action()* whether the behavior has ended. Just returns the value of *finished*.
public int	*onEnd()*	Called once when the behavior is ended, before removing it. Performs operations addOnSuccess and removeOnSuccess (if *success = true*), addOnFail and removeOnFail (if *success = false*), addOnEnd and removeOnEnd (in any case), and also calls *wakeAgent()*. If needs to be overridden in subclasses, we recommend calling *super.onEnd()* so these actions are still performed.

Public Fields:

	Name	Meaning	Default
public	*finished*	Whether the behavior has ended. This field is checked	true

boolean		after every execution of *action()*. Default value is true. So, by default, a RAB is a one-shot-behavior. Needs to be set to *false* in order to create cyclic, waiting, etc. behaviors.	
public boolean	*success*	Whether the behavior has ended in success or failure. Used inside *onEnd()* to lead the actions.	true

Public API methods provided:

	Name	Meaning
public String	*getParameterValue* (String name, String ns)	Get the value of the first found start parameter with specified name and namespace.
public String	*getParameterValue* (String name)	The same as above, only namespace is not taken into account.
public Vector	*getParameterValues* (String name, String ns)	Get the values (Vector of Strings) for all found start parameters with specified name and namespace.
public Vector	*getParameterValues* (String name)	The same as above, only namespace is not taken into account.
public void	*print* (String text)	Prints to the screen (and to the log) specified text. Text is printed preceded by "[<current time>] <agent's name>: "
public void	*wakeAgent*()	Informs the agent that it must wake up its *Live* behavior to check whether new rules became applicable.
public void	*addBelief* (SemanticStatement belief)	Adds specified belief to agent's beliefs. Must not contain "*".
public void	*removeBelief* (SemanticStatement belief)	Removes specified belief from agent's beliefs. If the specified belief contains "*" and matches several existing beliefs, all matching beliefs will be removed.
public Boolean	*hasBelief* (SemanticStatement belief)	Checks whether agent's beliefs include the specified belief, which can have "*".
Public Vector	*getBeliefs* (SemanticStatement template)	Returns all the beliefs (Vector of SemanticStatements) matching the specified template.

Relevant API methods and fields inherited from JADE's SimpleBehaviour (consult JADE documentation for more details)*:*

	Name	Meaning
public void	*block*()	Block behavior (do not call) until a new message arrives.
public void	*block*(long millis)	Block behavior until a new message arrives or until specified time (in milliseconds) elapses.

public boolean	*isRunnable*()	Returns false if behavior is blocked and true if not.
protected Agent	*myAgent*	Reference to the agent owning this behavior. Enables behaviors to call public methods of jade.core.Agent such as send(), receive(), doMove() and other. After the type casting ((SmartResourceAgent)myAgent), enables access to public methods of SmartResourceAgent.

Example of usage: A trimmed down version of ResponseReceiverBehavior

```
package smartresource.shared;
import smartresource.core.*;

import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;

public class ResponseReceiverBehavior extends ReusableAtomicBehavior{

      String convID="";
      String saveTo="";

      public void onStart(){
            convID=getParameterValue("conversationID");
            saveTo=getParameterValue("saveTo");
      }

      public void action() {
         finished=false;
         ACLMessage msg = myAgent.receive(MessageTemplate.
                                          MatchConversationId(convID));

         if (msg!=null){
            String message=msg.getContent();
            print("\""+message+"\" from "+msg.getSender().
                                                getLocalName()+" received");
            SemanticStatement ss_output=new SemanticStatement(saveTo);
            if(ss_output.isSubjectUndefined()) ss_output.subject=message;
            if(ss_output.isPredicateUndefined()) ss_output.predicate=message;
            if(ss_output.isObjectUndefined()) ss_output.object=message;
            addBelief(ss_output);
            finished=true;
         }
         else block();
      }
}
```

4.2 Semantic Statement

Full name: smartresource.core.SemanticStatement

Extends: nothing.

Implementing RABs, the programmer will often need to work with instances of this class. It is because an agent's beliefs set is basically a Vector of Semantic Statements and therefore ReusableAtomicBehavior's methods hasBelief(), getBeliefs(), addBelief() and removeBelief() operate on them.

Public Fields:

	Name	Meaning	Default
public String	*subject*	The subject of the statement.	"?"
public String	*predicate*	The predicate of the statement.	"?"
public String	*object*	The object of the statement.	"?"

For all three components, "?" has the meaning of "not defined", "*" has the meaning of "anything", and "*<var>*" refers to a variable.

Constructors:

Name	Meaning
SemanticStatement (String s, String p, String o)	Semantic Statement is created and initialized with given subject, predicate and object. Leading and trailing whitespaces in all three elements are trimmed off. If any of the elements is given *null*, it is set to "?".
SemanticStatement (String str)	Semantic Statement is created and initialized from a String of the format (whitespace-separated) "<subject> <predicate> <object>". If only one whitespace is found, the object is set to "?". If no whitespace is found, both predicate and object are set to "?".
SemanticStatement (Object toCopy)	Semantic Statement is created and initialized as a copy of the Semantic Statement passed as the parameter.

Public API methods provided:

	Name	Meaning
public boolean	*isFullyDefined*()	Returns false if either subject, predicate or object is undefined (i.e. is "?"), and returns true otherwise.
public boolean	*isSubjectUndefined*()	Returns true is the subject is undefined.
public boolean	*isPredicateUndefined*()	Returns true is the predicate is undefined.
public	*isObjectUndefined*()	Returns true is the object is undefined.

boolean		
public boolean	*isSubjectAny*()	Returns true if the subject is “anything” (i.e. starts with “*”)
public boolean	*isPredicateAny*()	Returns true if the predicate is “anything” (i.e. starts with “*”)
public boolean	*isObjectAny*()	Returns true if the object is “anything” (i.e. starts with “*”)
public boolean	*hasVariables*()	Returns false if either subject, predicate or object is a variable, i.e. of the form *x*.
public String	*getSubjectVariableName*()	Returns the name of the variable in the subject (“x” if it is *x*) or null if it is not a variable.
public String	*getPredicateVariableName*()	Returns the name of the variable in the predicate (“x” if it is *x*) or null if it is not a variable.
public String	*getObjectVariableName*()	Returns the name of the variable in the object (“x” if it is *x*) or null if it is not a variable.
public boolean	*equals*(Object obj)	Returns true if this statement is equal to the one given as the parameter, taking “*” into account.
public boolean	*bindVars*(HashMap vars)	Substitutes variables with their values given in the HashMap <String var, String value>.
public String	*toString*()	Creates string representation of the statement “<subject> <predicate> <object>”.

Example of usage: See the example in the section on Reusable Atomic Behavior. The input parameter saveTo can, for instance, be “Agent1 HasPrice ?”.

4.3 Variables Binding Manager

Full name: smartresource.core.VariablesBindingManager

Extends: nothing.

This class was created in order to handle variables in the behavior rules. However, it can also be utilized by the programmer when implementing a RAB that needs some kind of rule-based logic itself. An example is SimpleSelectBehavior (see corresponding section).

Public API methods provided:

	Name	Meaning
public	*VariablesBindingManager*()	Constructor

public boolean	*evaluate*(SemanticStatement stat, Vector vect, int mode)	Evaluates a Semantic Statement against a Vector of Statements (e.g. agent's beliefs) and a current set of matching bindings of variables (stored locally). Also prepares an updated set of bindings (but does not adopt it yet). If the parameter "mode" is *VariablesBindingManager.TRUE_IF*, the updated set of bindings contains the bindings according to which *stat* belongs to *vect*, and the method returns true if there is at least one such binding. If the parameter "mode" is *VariablesBindingManager.FALSE_IF*, the updated set of bindings contains the bindings according to which *stat* does not belong to *vect*, and the method returns true if this set is empty.
public void	*emptyNewBindings*()	Empties the updated set of bindings created in the course of previous *evaluate*() operation(s).
public void	*adoptNewBindings*()	Adopts the updated set of bindings as the current set and performs *emptyNewBindings.*
public int	*getNumberOfBindings*()	Returns the number of bindings in the current set.
public HashMap	*getBinding*(int index)	Returns HashMap <String var, String value> representing the binding at position *index* in the current set of bindings. If *index* is larger that the number of bindings, returns an empty HashMap.

Example of usage: Some relevant pieces of code from SimpleSelectBehavior

```
import java.util.*;
...

Vector inputs=getParameterValues("input");
String output=getParameterValue("output");

VariablesBindingManager manager=new VariablesBindingManager();
for(int i=0;i<inputs.size();i++){
     SemanticStatementst=new SemanticStatement((String)inputs.elementAt(i));
     boolean  found=manager.evaluate(st,  ((SmartResourceAgent)myAgent).beliefs,
                                          VariablesBindingManager.TRUE_IF);
     if(found){
        manager.adoptNewBindings();
     }
     else{
        success=false;
        return;
     }
}

...
/*selection process*/
...

vars=manager.getBinding(best);
Iterator itvars=vars.entrySet().iterator();
while(itvars.hasNext()){
      Map.Entry ent=(Map.Entry)itvars.next();
```

```
      String regex="\\*"+(String)ent.getKey()+"\\*";
      output=output.replaceAll(regex,(String)ent.getValue());
}
SemanticStatement ss_output=new SemanticStatement(output);
addBelief(ss_output);
```

4.4 Interface Event Handler

Full name: smartresource.core.InterfaceEventHandler

Extends: smartresource.core.ReusableAtomicBehavior

In some cases, an agent has to be able to react to some events in synchronous way, i.e. to provide some response immediately to the same channel, through which the request arrived. A common case is when an agent starts an instance of smartresource.core.SmartResourceAgentServer to be able to communicate with external applications which can issue HTTP requests, e.g. GoogleEarth.

In such a case, the programmer needs to make a class that extends *InterfaceEventHandler*, and to implement its *handle()* method. *onStart()* can be implemented although it is hardly sensible, *action()* is not used. *onEnd*() is called as well, so using the *success* field is possible to lead the selection between *add(remove)OnSuccess* and *add(remove)OnFail*.

Methods to be implemented in subclasses:

	Name	Meaning	Mandatory
public String	*handle*(String event)	Body of the handler. The return String of the method will be sent to the requestor without modifications.	Yes

Public API methods are as in ReusableAtomicBehavior (see corresponding section).

Example of usage:

```
public class ExpertAgentHandler extends InterfaceEventHandler
{
      public String handle(String event){
          String message="";
          if(event.startsWith("Get Network")){
            message="some xml";
          }
          else ...

          String response="HTTP/1.0 200 OK \r\nContent-Type:
      text/xml\r\n\r\n";
          response=response+message;
          return response;
      }
}
```

4.5 SmartResource Agent GUI

Full name: smartresource.core.SmartResourceAgentGUI

Extends: javax.swing.JFrame

If there is a need to give agent a new window-based GUI, the programmer needs to make a class that extends *SmartResourceAgentGUI*, to implement its *construct()* method, an action listener, and *modify()* method (if GUI needs to react agent's command).

Methods to be implemented in subclasses:

	Name	Meaning	Mandatory
public void	*construct*()	To initialize the GUI: create the elements, set size, create and assign the action listener.	Yes
public void	*modify*()	To perform some actions upon agent' request, mainly related to modifying GUI (enabling/disabling elements, putting text to labels, etc.). From the script, called when "@ModifyGUI" is given as the behavior class.	No

Before calling *construct*() or *modify*(), the agent will set the start parameters provided in the corresponding behavior rule. The access to those parameters is possible using the same methods as those in ReusableAtomicBehavior, i.e. *getParameterValue* (String name, String ns), *getParameterValue* (String name), *getParameterValues* (String name, String ns), *getParameterValues* (String name).

Note that there are two inputs for *construct*() that are handled in the implementation of SmartResourceAgentGUI, and therefore automatically apply to any subclass. This is done inside method public void *start*(), which is called after *construct*(), processes these two inputs and makes the window visible. If there is a need to overrule this functionality, the subclass needs to override *start*().

Name	Meaning	Mandatory	Default
x:title	The title of the window.	No	""
x:state	The state to start in: "normal" or "minimized".	No	"normal"

The action listener must create a *GuiEvent*, add a single parameter defining the event, and post it through *myAgent.postGuiEvent* (see example).

Example of usage: Implementation of SeveralButtonsGUI (see corresponding section)

```
package smartresource.shared.gui;
import smartresource.core.*;
import javax.swing.*;
import java.awt.*;
import java.util.Vector;
import java.util.Iterator;
```

```
import jade.gui.GuiEvent;

public class SeveralButtonsGUI extends SmartResourceAgentGUI
{
	Vector buttons;

	public void construct(){
		Container pane=getContentPane();
		setLayout(new BoxLayout(pane, BoxLayout.Y_AXIS));
		pane.add(Box.createRigidArea(new Dimension(0,10)));
		ActList aActList = new ActList();

		buttons=new Vector();
		Vector b=getParameterValues("button");
		Iterator it=b.iterator();
		while(it.hasNext()){
			String text=(String)it.next();
			JButton button=new JButton(text);
			buttons.add(button);
			pane.add(button);
			pane.add(Box.createRigidArea(new Dimension(0,10)));
			button.addActionListener(aActList);
		}
		pack();
		setSize((int)getSize().getWidth()+20,(int)getSize().getHeight());
	}

	public void modify(){
		String event=getParameterValue("modify");
		String target=getParameterValue("target");
		int button=new Integer(target).intValue();

		if(event.equals("Enable")){
			((JButton)buttons.elementAt(button-1)).setEnabled(true);
		}
		else if(event.equals("Disable")){
			((JButton)buttons.elementAt(button-1)).setEnabled(false);
		}
	}

	class ActList implements java.awt.event.ActionListener
	{
		public void actionPerformed(java.awt.event.ActionEvent event)
		{
			Object object = event.getSource();
			int button=buttons.indexOf(object)+1;
			GuiEvent ev = new GuiEvent(this, 0);
			ev.addParameter("User ClickedButton "+button);
			myAgent.postGuiEvent(ev);
		}
	}

}
```

4.6 Working with HTTP

The SmartResource Platform provides the class smartresource.core.SmartResourceAgentServer. An agent can start an instance of this class to be able to communicate with external applications which can issue HTTP requests, e.g. GoogleEarth. Upon receiving an HTTP request,

SmartResource Agent Server creates a *ServerEvent* (encapsulating a request String and a Socket through which the response should be sent) and posts it to the agent using *myAgent.postServerEvent*:

```
myAgent.postServerEvent(new ServerEvent(request,socket))
```

The method *postServerEvent* will put the event to the queue, so that the agent can pick it up and process in its own thread, rather than in the thread of the server. If, for some reason, there will be a need to extend the implementation of SmartResourceAgentServer, the same procedure should be followed.

In some cases, it may be needed to do it another way around, i.e. to start an agent under an HTTP server rather than start an HTTP server under an agent. Such a need can appear when attempting to integrate the agent system with an existing enterprise system. Starting a new container which will connect to the platform running on localhost:80, and an agent in this new container, can be done from any Java application using a code like follows:

```
jade.core.Runtime r=jade.core.Runtime.instance();
jade.core.ProfileImpl pi=new jade.core.ProfileImpl(false);
pi.setParameter(jade.core.ProfileImpl.MAIN_HOST,"localhost");
pi.setParameter(jade.core.ProfileImpl.MAIN_PORT,"80");
jade.wrapper.AgentContainer ac=r.createAgentContainer(pi);
SmartResourceAgent myAgent=new SmartResourceAgent();
Object[] params=new Object[2];
params[0]="DB/startup.rdf";
params[1]=role;
myAgent.setArguments(params);
jade.wrapper.AgentController actl=ac.acceptNewAgent(name, myAgent);
actl.start();
```

In such a case, using the normal procedure (myAgent.postServerEvent) is still preferable if possible. However, more likely, processing the request in the server queue will be unavoidable. For such cases, the following method is provided by SmartResourceAgent:

public String processServerMessage(String message)

To avoid harmful interactions (processing happen in parallel with the normal cycle of the agent), the method suspends the agent (the normal thread) for the time of processing the event and resumes afterwards. Otherwise, the processing routine is exactly the same.