Ubiware application developer guide

Authors:

Artem Katasonov

Michal Nagy

Michael Cochez

Organization:

Industrial Ontologies Group (IOG)

University of Jyväskylä

Version: 1.1

July 23, 2012

Changelog

Date	Who made change	Version	What changed
13.12.2011	-	1.0	-
23.07.2012	minagy	1.1	Formatting, typos, extra clarifications

Table of Contents

Preface

This document is written for programmers that would like to develop applications using Ubiware platform. This document describes the Ubiware platform and Semantic Agent Programming Language (S-APL). In order to understand S-APL, the reader should be familiar with these concepts:

- URIs (Uniform Resource Identifiers) and namespaces
- RDF (Resource Description Framework)
- Notation 3

A good source of reading is the web page of World Wide Web consortium (http://www.w3.org/) and http://www.rdfabout.com/. From this point on, we assume that the reader is familiar with the abovementioned concepts on a basic level.

Theoretical introduction to Agent-based software engineering

Agent

Agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators (= effectors). The schema of a simple agent can be seen in *Figure 1*.

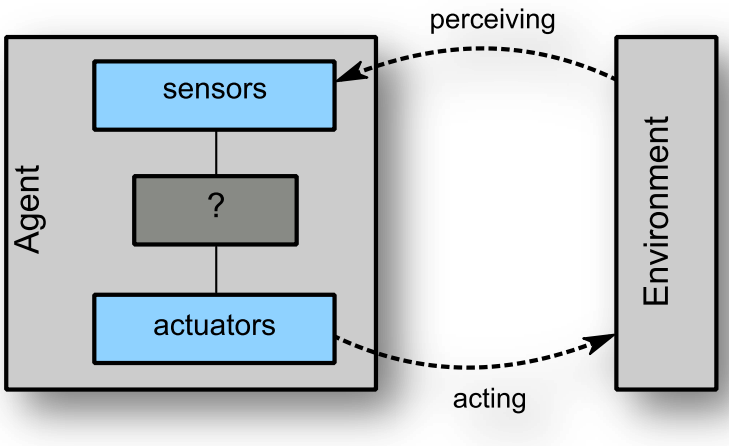

Figure 1: Simple agent

Mandatory properties of an agent are:

- Autonomy
  - Agents have control over their state and actions without the direct intervention of humans or other agents
  - Some people consider learning a necessary condition of autonomy as well
- Reactivity
  - Ability to perceive their environment (through sensors) and respond (through effectors) in a timely fashion to changes that occur in it
- Proactiveness
  - Ability to exhibit goal-directed behavior by taking the initiative
- Social ability (mandatory for MAS)
  - Engage in social activities (cooperation, negotiation, etc.) to achieve their goals

Optional properties are:

- Mobility
  - The ability to change its place
  - In software engineering often means migration from one container to another
- Ability to learn
  - By knowing the suitability of actions done in the past, the agent can learn the right actions for the future
  - The agent has to know the performance measure
- Truthfulness
  - It will not communicate any false information

Agent-oriented software engineering

Software engineering and artificial intelligence have slightly different views on agents. Software engineering sees an agent as a building block of software. From this idea a new paradigm has arisen – AOSE (Agent-oriented software engineering).

From the historical point of view, AOSE is a new step in software engineering. The difference between object-oriented (OO) and agent oriented (AO) programming is that in AO every building block has its own thread of control. This means that every agents "lives its own life". In traditional (single-threaded) OO applications, there was one thread of control that manipulated several objects (called methods, changed attribute values, etc.). In AO there are many threads of control, one per each building block (an agent). Each agent then performs its functions autonomously.

Agent platform

Within AOSE a new term was established – agent platform. An agent platform is a supporting software (usually middleware) that provides basic common functionality to the agents. This functionality includes communication subsystem, agent lifecycle management, migration capability, etc.

Since most agent platforms are distributed in nature, they can spread across several machines (computers). We will concentrate on the JADE[1] agent architecture, because originally Ubiware was based on it. JADE uses the term container. Each platform can have one or more containers. Agents can live only in containers, not "directly" on the platform. Each container can "live" on a different physical machine. Sometimes several containers can live on one machine, but almost never one container on several machines. An example of such scenario can be seen in *Figure 2*.

[1] JADE stands for Java Agent DEvelopment Framework. It is an open source agent platform. JADE agents are programmed using Java language.

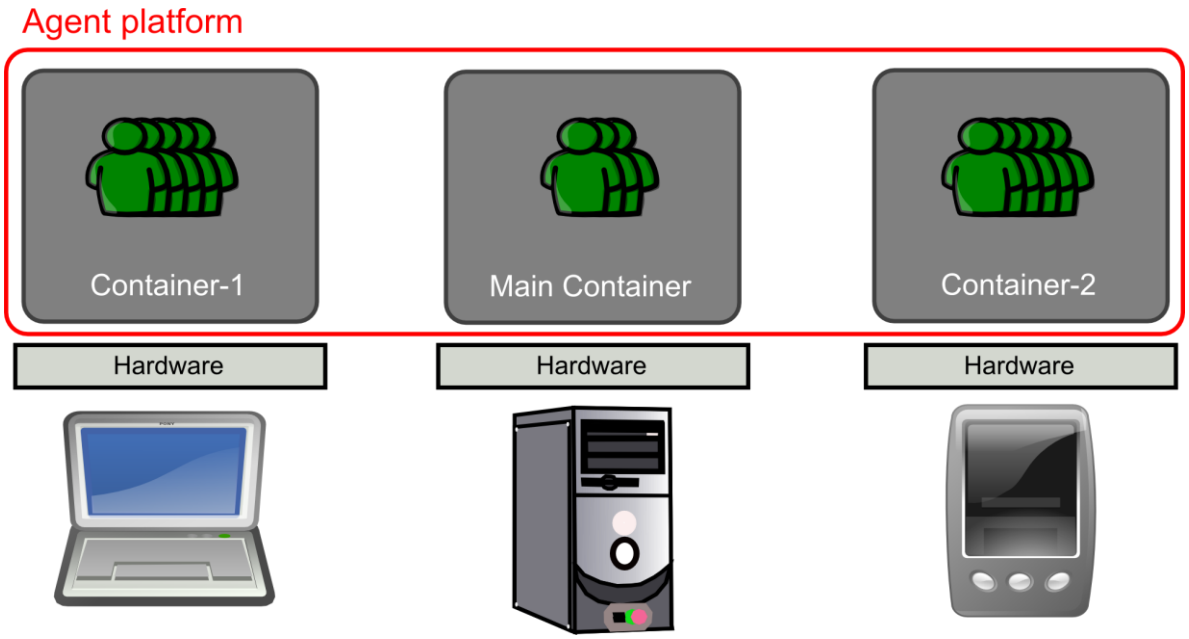

Figure 2: An example of an agent platform consisting of containers

Introduction to Ubiware platform

Typical Ubiware agent

In *Figure 1* we depicted how a typical agent looks like. We mentioned that it consitst of sensors, actuators and some decision component (marked as '?') that reads information from the sensors and influences the environment through actuators. In *Figure 3* you can see how these components correlate with Ubiware architecture. The question mark from *Figure 1* is represented by the engine and belief storage. The effectors and actuators didn't change their representation much. Please try to remember this figure. Keeping this architecture in mind helps understand the operation of the Ubiware platform.

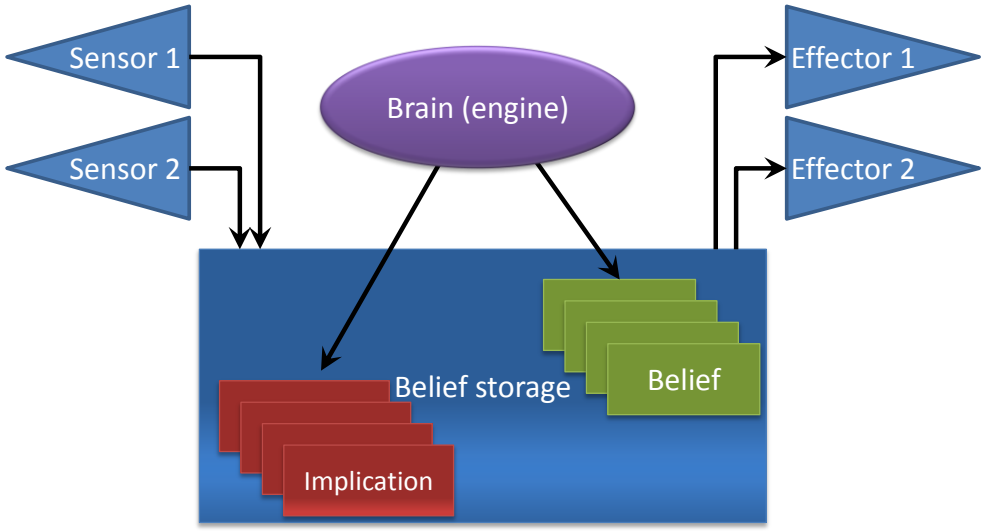

Figure 3: Three-layered model of Ubiware agent

In Ubiware each of the components from *Figure 3* are implemented in a particular way. The behavior engine is implemented as a Java application. Sensors and effectors are implemented as so-called Reusable Atomic Behaviors (RABs)[2]. The belief storage is basically a portion of memory where the beliefs are stored. The initial (first) agent's beliefs are loaded from an agent script written in S-APL (Semantic Agent Programming Language)[3].

Another way of looking at these components is through a layered model depicted in *Figure 4*. Again, you can see these components as three layers – behavior engine, belief storage and Reusable Atomic Behaviors (RABs).

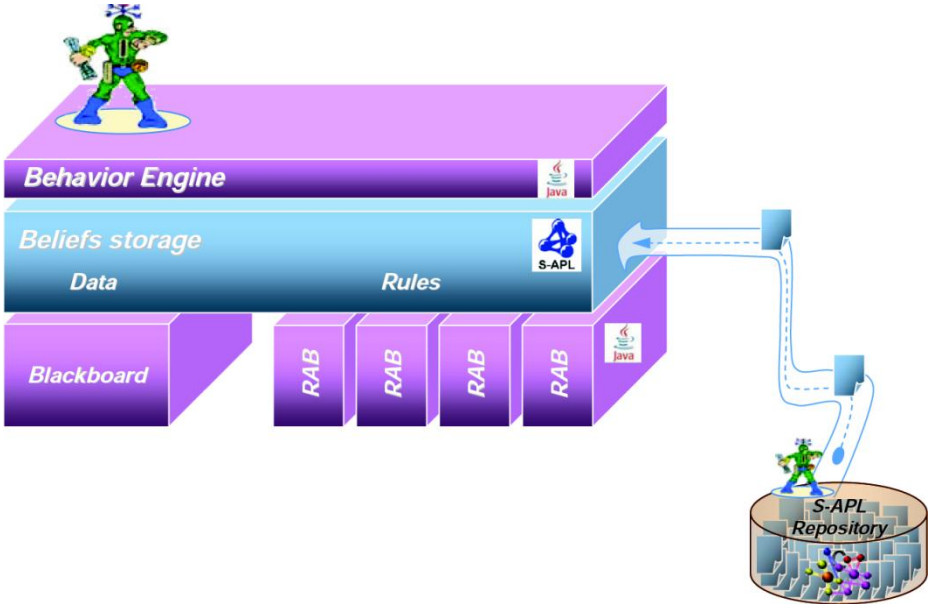

Figure 4: Three-layered hierarchical model of a Ubiware agent

Reusable Atomic Behavior

RAB is a Java class implementing a reasonably atomic function. RABs are called reusable, because they can be reused across scenarios, agents and roles. The agent programmer can call a RAB with some given parameters and it will execute the task it was designed for. Usually they are used as sensors and actuators, but they can fulfill other tasks as well. A few examples of RABs and their function can be found in Table 1.

RAB	Function
MessageSender	Sends message to other agents on the platform
MessageReceiver	Makes agent listen for incoming messages
EmailSender	Sends an email to a particular address
CreateAgent	Creates a new agent
DirectoryLookup	Looks up an agent in "Yellow pages"

Table 1: Several types of RABs

If the policy management is enabled, an agent can execute only RABs that belong to his security group. The group membership and RAB execution is checked by the Policy manager agent. If the policy management is disabled, an agent can run any RAB.

[2] RABs are also pieces of Java code, but they are used in a different way than the engine.

[3] The relationship between S-APL and Ubiware is the following: Ubiware is a multi-agent platform and S-APL is the language that is used to program agents running on Ubiware. Big portion of this guide explains how to program Ubiware agents using S-APL language.

Belief storage

The belief storage is a database that contains all the beliefs of the agent. In *Figure 3* you can see that the belief storage can include either general beliefs or implications. General beliefs are pieces of information that can be expressed in short sentences like "John loves Mary". Implications are rules that say what should happen if the agent has a certain belief. An example of an implication would be "If John loves some person, then John buys flowers for this person". In Ubiware implications are considered just special beliefs[4]. Thanks to this principle you can create new implications or change existing ones.
It is important to realize that the belief storage is dynamic. In the beginning (when the agent is started) the content of the agent script is copied to the empty belief storage. When the agent evolves and rules are executed the beliefs will change and they will not be the same as beliefs from the beginning of the agent's life.

Behavior engine

Every implication consists of left side (query) and the right side (effect). Sometimes implications can be called rules. Each agent has its own copy of the behavior engine, therefore they are independent. The engine is executing so-called Live behavior. This behavior can be described as following[5]:

1. It makes the agent iterate through all the rules (implications) in the belief storage and find out which rules are executable.
    - The rule is executable if and only if the left side of the rule specifies a query that has a solution (the query result is non-empty)
2. It executes executable rules.
    - Rule execution means that all the variables on the right side of the rule will be bound to its values (based on the query) and the whole right side of the rule will be inserted into the belief storage.

Hello World application

In order to start with a simple application, download *Ubiware.zip* from the documentation web page. Unpack the ZIP file. You will see two folders – Ubiware and jEdit. Ubiware folder contains the platform and jEdit contains the editor that you can use to edit agent scripts (S-APL scripts).

Inside the ubiware folder there are the following files:

- run.bat – batch script starting the platform on Windows
- run.sh – bash script starting the platform on Unix-based systems
- startup.rdf – script that describes the startup configuration of the platform
- HelloWorld.sapl – HelloWorld agent script

[4] In Figure 3 they have been marked explicitly, because they are essential to the operation of the Ubiware agent
[5] Actually the truth is more complicated than that, however in order to keep this introduction short and simple, we don't mention the whole process of rule execution.

In order to run this example, run the run.sh or run.bat file. You will see a window with several information statements from the platform and the last message will be “Hello world”. This is the message from the agent. The code of the agent is in the script HelloWorld.sapl.

The next section explains the syntax and semantics of S-APL. If you want to try out some S-APL code, just write it into the HelloWorld.sapl file and run it with run.bat or run.sh.

Semantic Agent Programming Language (S-APL)

Introduction

At the present stage, the namespaces used in this document are defined as follows:

```
@prefix sapl: <http://www.ubiware.jyu.fi/sapl#> .

@prefix java: <http://www.ubiware.jyu.fi/rab#> .

@prefix p: <http://www.ubiware.jyu.fi/rab_parameters#> .

@prefix ex: <http://www.example.com/ubiwareexamples#> .
```

In general you are free to define any prefix/namespace you want. The only thing to remember is that these three abovementioned prefixes/namespaces are reserved for S-APL.

In some examples in this document other prefixes are used. They represent just example namespaces and are not part of S-APL language. Examples of resources with such prefixes are `h:feels`, `h:hungry`, `a:searchFood`, `f:hasBrother`, etc.

S-APL Axioms

- Everything is a *belief*. All other mental attitudes such as desires, goals, commitments, behavioral rules are just complex beliefs.
- Every belief is either a semantic statement (subject-predicate-object triple) or a linked set of such statements.
- Every belief has the *context container* that restricts the scope of validity of that belief. Beliefs have any meaning *only inside* their respective contexts.
- Statements can be made about context, i.e. contexts may appear as subjects or/and objects of triples. Such statements give meaning to contexts. This also leads to a *hierarchy* of contexts (not necessarily a tree structure though).
- There is the *General Context G*, which is the root of the hierarchy. G is the context for the global beliefs of the agent (what it believes to be true here and now). Nevertheless, every local belief, through the hierarchical chain of its contexts, is linked to G.
- Making statements about other statements directly (without mediation of a context container) is not allowed. The only exception is when a statement appears as the object of one of the following predicates: `sapl:add`, `sapl:remove` and `sapl:erase`.

S-APL Notation

The notation that is selected for use in S-APL is a subset of Notation3 (http://www.w3.org/DesignIssues/Notation3.html). This notation was developed Tim Berners-Lee (inventor of WWW, founder of W3C, and the one who coined the Semantic Web concept). Notation3

was proposed by Berners-Lee as an alternative to the dominant notation for RDF, which is RDF/XML. Notation3 is a language which is more compact and probably better readable than RDF/XML, and is also extended to allow greater expressiveness.

One feature of Notation3, which in a sense goes beyond the standard RDF, is the concept of *formula* that allow Notation3 graphs to be quoted within Notation3 graphs using { and }. There is no definition what is the precise semantics of formulae. In S-APL, however, we avoid the issue by fixing that a formula is a *Container that holds a set of reified statements*. Under this convention, S-APL documents remain compliant to the standard RDF data model and can be translated into RDF/XML is a need would arise (this of course would lead to a significant increase in the document length).

The description of S-APL notation follows:

- A statement is a white-space-separated sequence of subject, predicate and object
- Dot (**.**) followed by a white space separates statements of the same level, i.e. *S P O . S P O*
- Semicolon (**;**) followed by a white space allows making several statements about the same subject, i.e. *S P O ; P O*
- Comma (**,**) followed by a white space allows making several statements having common subject and predicate, i.e. *S P O , O*
- **{ }** denotes reification, it may appear as the subject or the object of a statement and has to include inside itself one or more other statements, e.g. *S P { S P O }* or *{ S P O } P { S P O }.* Reification always implies a context; however, the relation is not necessarily 1-to-1. E.g. *{S P O} P O ; P O* implies that the statement in {} is linked to two different contexts defined as given.
- Colon (**:**) is used to specify an URI as a combination of the namespace and the local name, i.e. *ns:localname* There can be default namespace, the colon is used anyway, i.e. *:localname*.
- *@prefix prefix: namespace* links a prefix to a namespace.
- URIs given directly are to be inside **< >**, i.e. *<http://someaddress>.*
- Literals containing whitespaces, **{**, **}**, **<**, **>** , **"**, or **:** are to be inside **" "**, i.e. *"some literal"* .
- Comments are java-style, i.e. **/*** *comment* ***/** as well as **//** *comment <end of line>*
- Character escaping is java-style as well, i.e. e.g. for " symbol, use **\"** while for backslash symbol itself, use ****
- N3 syntax for anonymous nodes with **[ ]**, i.e. *S P [P O]* or *[P O] P O* is also supported.
- N3 syntax for RDF lists of resources with **( )** , i.e. *(R R R ..) P O* or *S P (R R R ..)* is also supported.

Basic descriptive constructs

Note: "sapl" namespace is used for the resources that are defined in the language's ontology. The default namespace is used for all the other resources, which are assumed to be defined somewhere else.

In S-APL, just stating something has the meaning of adding it to the beliefs of the agent. How adding beliefs exactly works, see below in "Unconditional commitment to adding a belief".

Simple belief:

```
:John :Loves :Mary

{:John :Loves :Mary} sapl:is sapl:true
```

These two are equivalent and the S-APL parser just transforms the latter into the former. The latter is introduced for various syntactic purposes which will be explained later in the text.

Belief with a context:

```
{:John :Loves :Mary} :accordingTo :Bill

{:John :Loves :Mary} :since {:Year :Is 2005}
```

Belief about something being falsehood

```
{:John :Loves :Mary} sapl:is sapl:false
```

Please notice that in case of a belief you have to think of three possible states: First, the agent knows that John loves Mary. Second, the agent knows that John does not love Mary. In this case we also can say that the agent knows that "John loves Mary" is false. Third state is that the agent does not know anything about John loving or not loving Mary.

Therefore do not assume that if some belief does not exist in the beliefs of the agent, it automatically means that it is false. It may be true as well, the agent just does not know at the moment.

Embedded beliefs

There is small set of beliefs that are created and updated by the agent's engine.

`sapl:I sapl:haveName ?x` (the local name of the agent)

`sapl:Now sapl:is ?x` (current system time in milliseconds)

`sapl:Time sapl:is ?x` (the present time of the day in the format "hh:mm:ss".)

`sapl:Date sapl:is ?x` (the present date in the format "yyyy-mm-dd".)

Used formats for Time and Date enable using lexicographical comparison, using predicates described below. It is also possible to convert those to number representation (time in milliseconds) using timeMillis() function and to do some calculations needed.

Running a RAB

Running a RAB:

An RAB call has 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.MessageSenderBehavior}

sapl:configuredAs

        {p:receiver sapl:is ex:John .

         p:content sapl:is "bla bla" .

             sapl:Success sapl:add {:John :was :notified}

        }
```

When the agent's engine finds such a belief in G, it executes the specified action. This is done only if the prefix is either `java:`. If the prefix is `java:`, the commitment is moved into the context container "sapl:I sapl:doing {}". When the RAB ends execution, the corresponding statement is removed from "sapl:I sapl:doing {}".

If action has no parameters, it should be written as `{sapl:I sapl:do <action>} sapl:configuredAs sapl:null`. In other words, you must not put an empty container in position of subject. Instead, you put sapl:null.

In the configuration part, one may use special statements to add or remove beliefs. The subject can be `sapl:Start`, `sapl:End`, `sapl:Success`, and `sapl:Fail`. The predicate is either `sapl:add` or `sapl:remove`.

How adding beliefs exactly works, see below in "Unconditional commitment to adding a belief". How beliefs' removal exactly works, see below in "Unconditional commitment to removing a belief".

If you want to find out more about available RABs, read the RAB overview document. It contains the list of available RABs with examples.

Sequential actions / plan:

```
{sapl:I sapl:do <RAB1>} sapl:configuredAs
    { ... sapl:Success sapl:add {
        {sapl:I sapl:do <RAB2>} sapl:configuredAs {...} } }
```

One can make sequential plans of RAB execution. In the example above RAB1 will be executed unconditionally. Upon successful completion of RAB1, also RAB2 will be executed. However, if RAB1 fails, RAB2 will not be executed. By nesting these statements, one can create a chain of RAB executions where the later step depends on the successful or unsuccessful execution of the former step. Please notice also other related subjects that can be used in a similar manner: `sapl:Start`, `sapl:End`, and `sapl:Fail`.

Adding and removing beliefs

Unconditional commitment to adding a belief:

```
sapl:I sapl:add {:John :Loves :Mary}
```

When the agent's engine finds such a belief in G, it adds (note: copies, i.e. creates a new belief based on the given template) the specified belief to G and then removes the commitment itself. The construct as above is not normally needed, because just writing `:John :Loves :Mary` has the same effect. It may be used, however, to allow performing `sapl:I sapl:add ?x`, where ?x is a variable bound to the ID of a context container or a statement. Note again, that a copy is will be added. This is in contrast to performing `?x sapl:is sapl:true`, which will link existing beliefs to G. Some details of the adding procedure are below:

- If a belief exists in some context such as {:John :Loves :Mary} :accordingTo :Bill, and one adds to the same context a new belief such as {:Bob :Likes :Jane} :accordingTo :Bill, the result is {:John :Loves :Mary. :Bob :Likes :Jane} :accordingTo :Bill. In other words, only the leaf-statement is added without duplicating the context structure.
- Exceptions are statements sapl:I sapl:doNotBelieve {}, sapl:I sapl:remove {}, {} sapl:configuredAs sapl:null, {} sapl:forSome ?x, {} sapl:forAll ?x, which are never merged with existing ones. Also, no mergers is performed inside the context sapl:I sapl:want {}.
- The merge is not done for contexts defined through their link to some another context. E.g. if a belief exists such as {:John :Loves :Mary} :since {:Year :Is 2005}, and one adds another belief such as {:Bob :Likes :Jane} :since {:Year :Is 2005}, no mergers will be done at all, i.e. the result is {:John :Loves :Mary} :since {:Year :Is 2005}. {Bob :Likes :Jane} :since {:Year :Is 2005}.
- If the destination context container already has the exact same statement, nothing is added. The commitment is considered fulfilled anyway.

Unconditional commitment to removing a belief:

```
sapl:I sapl:remove {:John :Loves :Mary}
```

When the agent's engine finds such a belief in G, it removes the specified belief and then removes the commitment itself.

The object of such a condition, presents a *pattern* that is used for the removing beliefs through matching it with G. The details of this procedure follow (all applies also to removal through "sapl:Success sapl:remove {...}"):

- If several beliefs are given, e.g. {:John :Loves :Mary. :Mary :Loves :John}, all must match with G for anything to be removed.
- The commitment is considered fulfilled and thus it therefore removed even if no matching belief was found and thus nothing was actually removed.
- If a hierarchical belief structure is given, e.g. {:John :Loves :Mary. :John sapl:want { {:Mary sapl:do a:SendMail} sapl:configuredAs {p:receiver sapl:is John} } } :accordingTo :Bob, the following statements will be removed: :John :Loves :Mary, {} sapl:configuredAs {}, :Mary sapl:do a:SendMail, and p:receiver sapl:is John.. In other words, the statements are removed that match with statements given in "remove" which do not refer to contexts at all or which have contexts as both the subject and the object.
- One can use *, e.g. :John :Loves *. In result, all the matching beliefs like :John :Loves :Mary, :John :Loves :Grandpa and even :John :Loves {…} will be removed.

Basically, the content inside sapl:I sapl:remove {…} is a query similar to those used in the left side of {…} => {} (see below). It can use therefore all the querying constructs (see Section **Error! Reference ource not found.**): variables, sapl:or, sapl:Optional, filtering predicates, statistic predicates, etc.

Note that contexts for rules, goals, goals under work, and falsehoods are protected against removal (both direct and through garbage collection). This means that e.g. `sapl:I sapl:remove {sapl:I sapl:want *}` will have no effect. If one wants to drop all the goals, one has to use `sapl:I sapl:remove {sapl:I sapl:want {* * *}}` instead.

Unconditional commitment to erasing a sub-graph through an ID of a statement or a context container:

```
sapl:I sapl:erase ?x
```

?x is a variable bound to the ID of a statement or a context container. Note the difference: if one uses `sapl:I sapl:remove ?x`, where ?x holds the ID of a context container, the contents of that container will be used as a pattern for removing from G. Using `sapl:erase` will result in removing the given container itself.

Implications

Implications in Ubiware can be of two main types – behavioral rules and commitments. Both types are explained in detail later in this section. *Table 2* and *Table 3* show all types of implications in a compact form. These two tables are very useful as cheat sheets once one knows how these implications work in detail.

Symbol	SAPL predicate	Name	What happens on the implication?	
			If left side true	If left side false
=>	sapl:implies	conditional commitment	Deleted	Stays
->	sapl:impliesNow	conditional action	Deleted	Deleted

Table 2: Implications of type commitment

Symbol	SAPL predicate	Name	Behavior	
			If left side true	If left side false
=>	sapl:implies	behavior rule	Stays	Stays
==>	sapl:infers	inference rule	??????	??????

Table 3: Implications of type rule

There is also another type of commitment represented by SAPL predicate sapl:achievedBy (or >> symbol). You may find it in some older code, however this predicate is deprecated and it is not recommended to use it anymore.

Conditional commitment:

```
{sapl:I h:feel h:hungry} =>
 {sapl:I a:performAction a:foodSearch}
```

=> is the shorthand for `sapl:implies`. When the agent's engine finds such a belief in G and finds out that the conditions (if several, seen as AND-connected) in the subject context are met, it adds to G (note: copies while substituting bound variables with their values) all the beliefs specified in the object context. After that, the conditional commitment is removed. If you want to compare this implication with other implications, look at *Table 2* and *Table 3*.

Note that if the conditions in the subject context are not met, the commitment will stay in the agent beliefs until the conditions will be eventually met, or until the commitment is explicitly removed.

There are no restrictions on the contents of the right side of =>. For example, a conditional commitment can creates another conditional commitment:

```
{...} => { {...} => {...} }
```

Behavior rule:

```
{ {...} => {...} } sapl:is sapl:Rule
```

Unlike conditional commitments (those in G), rules (i.e. statements with => predicate) belonging to the context defined as "{} sapl:is sapl:Rule" will not be removed after an execution. Thus, behavioral rules describe repeated actions. If you want to compare this implication with other implications, look at *Table 2* and *Table 3*.

Nothing prevents, however, use of sapl:existsWhile (see below) to define rules that are removed upon some condition.

Inference rule:

```
{{...} ==> {...}} sapl:is sapl:Rule
```

==> is the shorthand for `sapl:infers.` If a behavior rule to be used for semantic inference (generating new facts from existing ones), one needs to: (1) add to the head of the rule the negation of the tail the rule (see "exclusive condition" below) – to avoid continuous non-stop execution of the rule; (2) use a set of sapl:All wrappings - for all relevant variables (see Section **Error! Reference ource not found.**) – to enforce that the rule infers all possible facts in one iteration. When using ==>, these two things are done automatically – negation of the tail is checked and the rule is executed for every solution found – the rest being exactly the same as for =>. If you want to compare this implication with other implications, look at *Table 2* and *Table 3*.

Meta-rule:

```
{ {...} => {...} } sapl:is sapl:MetaRule
```

Meta-rules behave exactly as normal behavior rules. The difference is only in their position in the agent's run-time cycle. Meta-rules are processed twice: just before starting processing normal rules and conditional commitments, and just after that - just before starting processing commitments to external actions (see Section **Error! Reference source not found.**). Therefore, the actual difference is n intended purpose: meta-rules are supposed to modify/block normal rules or commitments, to implement some organizational, e.g. security, policies. If you want to compare this implication with other implications, look at *Table 2* and *Table 3*.

Conditional action:

```
        {:John :Loves :Mary} ->
    {{sapl:I sapl:do java:SendMail} sapl:configuredAs {...}}
         ; sapl:else {...}
```

-> is the shorthand for `sapl:impliesNow`. The difference from sapl:implies is that such condition is removed if its subject is evaluated as false.

If such a statement is put in G, the result is a condition that is checked only once and removed even after that regardless if it lead to the action or not.

If such a statement is put into "{} sapl:is sapl:Rule" context, the result is a condition that is executed repeatedly as long as the condition is true, and removed as soon as the condition become false.

-> can accompanied by `sapl:else`, which specifies some beliefs and actions that are to added when the condition is evaluated as false. The sapl:else statement is always removed the same time when -> statement is removed. If you want to compare this implication with other implications, look at *Table 2* and *Table 3*.

Exclusive condition:

```
        {:John :Loves :Mary .
    sapl:I sapl:doNotBelieve {:John :hasBeen :notified}
    } => {...}
```

The context defined as "sapl:I sapl:doNotBelieve {}" is a special "virtual" context which is the complement of G, i.e. it includes all the possible beliefs that are not part of G.

`sapl:doNotBelieve` can also appear on inner container levels of the query, e.g. `sapl:I sapl:want {sapl:I sapl:doNotBelieve {:John :Loves :Mary}}`, which is in fact equivalent to `sapl:I sapl:doNotBelieve {sapl:I sapl:want {:John :Loves :Mary}}`.

Note that all the exclusive conditions are checked only after all the normal conditions are checked. This is important when variables are used (see Section **Error! Reference source not found.**).

Alternative conditions:

```
{{:John :Loves :Mary} sapl:or {:John :Loves :Jane}} => {...}
```

Using variables (see Section **Error! Reference source not found.**) is possible as well. E.g. `{?x Loves :Mary} sapl:or {?x :Loves :Jane}` will result in the set of values for ?x which is the set of those loving either Mary or Jane.

`sapl:or` can also appear on inner container levels of the query, e.g. `sapl:I sapl:want {{:John :Loves :Mary} sapl:or {:John :Loves :Jane}}.`

Querying constructs

The left side of an implication contains specification of beliefs to be matched against G.

Direct matching:

```
{:John :Loves :Mary} :accordingTo :Bill
```

Matching with variables:

```
{:John :Loves ?x} :accordingTo ?y
```

Such a query will be evaluated as true if there can be found some values of ?x and ?y so that the belief is present in G. In this case, the variables will be bound to the *first found* matching values, and, if the right side of => refers to these variables, these values will be used.

```
{
  {:John :Loves ?x} :accordingTo ?y. ?x sapl:is :Girl
} => {
  {sapl:I sapl:do java:SendMailRAB} sapl:configuredAs
                                        {p:receiver sapl:is ?x . … }
}
```

Variables in the right side of => are substituted with their values as using String.replaceAll. Therefore, DB/?AgentName/received/?model.sapl is a legal expression using two variables: ?AgentName and ?model. This is in contrast to the left side of => where a whole resource (subject, predicate or object of a statement) can only be a variable.

Beware: When using sequential structures of conditional commitments (i.e. when one conditional commitment leads to adding another conditional commitment), and defining additional variables at inner levels, take care not to use for inner variables some names that start with the same sequence

of characters as some already defined variables. For example, if an outer level has defined a variable ?x, while an inner level defines a variable ?x1, a problem will appear because when copying the right side of the outer-level =>, all strings matching "?x" will be substituted with value of ?x, e.g. :John leading to that all places where ?x1 is mentioned will become :John1.

Rules / conditional commitments that apply to all matching values:

```
{ {?x :Loves ?y} sapl:All ?x } => {...?x...?y...}
```

or

```
{ ?x :Loves ?y } => { {...?x...?y...} sapl:All ?x }
```

```
{ {{ ?x :Loves ?y } sapl:All ?x} sapl:All ?y } =>
      {...?x...?y... }
```

or

```
{ ?x :Loves ?y } =>
      { {{...?x...?y... } sapl:All ?x} sapl:All ?y } }
```

Normally, the implication is executed for *the first found* matching combination of variables' values. The use of sapl:All as shown in the first example above will lead that the rule will be executed for *every* matching value of ?x (taking first found value for ?y, if several values for ?y fit the same value for ?x) . This basically means that the contents of the right side of => will be copied several times into G, while using different values for variables. In the second example, the rule is executed for every matching combination of the variables.

As can be seen, sapl:All can be put on either the left side or on the right side of =>, and both cases are equivalent. It is also possible to put e.g. sapl:All ?x on the left side and sapl:All ?y on the right side, they will be just combined, i.e. the result is equivalent to the second example above.

sapl:All on the left side is allowed in order to enable external querying: when one agent requests from another agent some information using any of the constructs described in this section. On the other hand, sapl:All on the right side is allowed to enable defining different wrappings for different (top-level) resulting statements, e.g. `{...} => {X Y Z. {{A B ?x. {?x L ?y } sapl:All ?y} sapl:All ?x}}`. On the left side of =>, sapl:All must always wrap the whole contents of the container.

Note that the object of sapl:All, sapl:OrderBy, sapl:OrderByDescending as well as sapl:Some (see below) must be a variable. If the object is not a variable, the wrapping {} sapl:All ... will be removed and the contents of the container will be added at the present level.

Rules / conditional commitments that apply to a random matching value:

```
{ {?x :Loves ?y} sapl:Some ?x } => {...?x...?y...}
```

or

```
{ ?x :Loves ?y } => { {...?x...?y...} sapl:Some ?x }
```

In result, a random value for ?x will be selected among the matching ones. Everything above on sapl:All applies to sapl:Some.

sapl:Some can be combined with sapl:All. In any such a combination, sapl:Some are taking into account *after* sapl:All. In other words, in this case sapl:Some affects only the selection among alternative solutions having the same values for all sapl:All variables.

Other solution set modifiers

sapl:All can be combined with sapl:OrderBy, sapl:Limit, and sapl:Offset e.g.:

```
{{        {{{?x :Loves ?y} sapl:All ?x} sapl:OrderBy ?y
                    } sapl:Offset 2}  sapl:Limit 3} =>

                    {...?x...?y...}
```

Use of sapl:OrderBy rearranges the available solutions in the order of ascending ?y. Note that sapl:OrderBy is applied after sapl:All and sapl:Some (see below), and that if several sapl:OrderBy is given, the outermost wrapping defines the primary order while the second-inner one the secondary order etc.

In place of sapl:OrderBy, one can use `sapl:OrderByDescending`. Then, the solutions will be arranged in the order of descending value for the variable.

sapl:Limit limits the number of solutions taken, while sapl:Offset specifies the index of the first solution to be taken (e.g. in the example, the 3rd though 5th solutions are to be taken).

Similarly to sapl:All and sapl:Some, these set modifiers can be used in both left and right sides of =>. If used on both sides, sapl:OrderBy on the right side will add secondary ordering criteria to primary defined on the left side. In used on both sides, sapl:Limit and sapl:Offset on the right side will override those defined on the left side.

Optional conditions:

```
        {:John :Loves ?x .
     {?x :hasAddress ?a} sapl:is sapl:Optional
} => {{...?x...?a...} sapl:All ?x}
```

In result, the rule will be executed for every ?x loved by John, regardless of whether the agent knows the address of ?x, while for ?x with known address, the address information will be available and can be utilized somehow.

For solutions where a variable is unbound:

- If a statement in the right side of => has the variable as its subject, predicate or object, such a statement will just not be copied (and thus probably a whole sub-graph starting at the statement).
- If a statement in the right side of => has its subject, predicate or object as a literal that contains the variable, the variable will be replaced with an empty string.

sapl: Optional can also appear on inner container levels of the query.

Alternative conditions:

```
{{?x :Loves :Mary} sapl:or {?x :Loves :Jane}} => {...}
```

This will result in the set of values for ?x which is the set of those loving either Mary or Jane.

It is possible to define alternative conditions in a way so that different solutions will use different variables, or that some of the solutions will have some of the variables unbound. This situation is similar to that created by use of sapl:Optional, therefore see above for the description of the effects.

sapl:or can also appear on inner container levels of the query.

Matching with *:

```
:John :Loves *
```

This means “John loves something”. In principle, this is almost the same as `:John :Loves ?x`, only that in saves the agent’s engine from bother of recording the matching value for ?x.

Matching with variables standing for context containers:

```
?x :accordingTo :Bill
```

In this case, ?x is to be bound physically to the ID of the context container and logically to the whole underlying graph. Therefore, the above means “all the information that Bill provided”.

```
{{sapl:I sapl:do a:SendMail} sapl:configuredAs ?x.
?x sapl:hasMember {p:receiver sapl:is Mary}} => {...}
```

In this case, ?x will be bound to the ID of the configuration container of a:SendMail action commitment which contains “p:receiver sapl:is Mary” statement. `sapl:hasMember` and `sapl:memberOf` are defined as inverse predicates; so instead of the second line above one could also write `{p:receiver sapl:is Mary} sapl:memberOf ?x`.

It is possible to define queries like

```
{?x :accordingTo :Bill.
    {?x sapl:is sapl:true} :accordingTo :John } => {...}
```

Such query is evaluated as true if any belief that is found in the context container {} :accordingTo :Bill has a match in the context container {} :accordingTo :John.

It is also possible to add or remove beliefs using:

```
{...} => {?x sapl:is sapl:true}
{...} => {sapl:I sapl:add ?x}
{...} => {sapl:I sapl:remove ?x}
```

The difference between the first two is that in the first case, existing statements will be linked to G, while in the second case, the copies of them will be created and added. To remind, sapl:remove will use ?x as the pattern. If the goal is to remove the context itself, sapl:erase has to be used instead.

Using `sapl:memberOf` or `sapl:hasMember`, it is possible to add new statements to already existing context containers, e.g.:

```
{{sapl:I sapl:do a:SendMail} sapl:configuredAs ?x}} =>
    {{p:performative sapl:is inform} sapl:memberOf ?x}
```

It is also possible to check if the value of a variable is a context container ID. It is done with

```
?x rdf:type sapl:Container
```

Filtering using special (embedded) predicates

Filtering:

S-APL defines some predicates, which are to be evaluated by the agent's engine rather than matched against G. Those are:

```
?x = 5

?x != 5

?x > 5

?x < 5

?x >= 5

?x <= 5

?x sapl:regex "S.*h"
```

These predicates are normally used with a variable in the subject. However, it is also allowed to have a constant there, for example, a variable bound earlier. Therefore, the below is code is legal. It specifies some plan that is to be executed for all ?x loved by John, with a part of this plan that should be executed for all ?x except Jane.

```
{{:John :Loves ?x} sapl:All ?x} => {… …  {?x != :Jane} -> {…}}
```

The predicate sapl:regex directly uses the corresponding Java feature; for syntax information, consult the Java API docs about the class *java.util.regex.Pattern*.

The first six are shorthands for `sapl:eq`, `sapl:neq`, `sapl:gt`, `sapl:lt`, `sapl:gte`, `sapl:lte`. When using shorthands, we careful to have whitespaces before and after it; otherwise a syntax error will be reported. Numeric values are compared arithmetically; string values are compared lexicographically. The literal on the right (and only right) side of them is an expression that can refer to variables (including the one on the left) and can utilize following operations:

- for numerical values: `+ - * / ( )` , unary minus, and also functions (for all except the last one, see Java API docs on the class *java.lang.Math* for details):
  - `abs(x)` – absolute value of x
  - `sqrt(x)` – square root of x
  - `pow(x,y)` – x in the power of y
  - `random()` – a random value in the interval (0;1)
  - `ceil(x)` – smallest integer that is bigger than x

- `floor(x)` – biggest integer that is smaller than x
- `round(x)` – closest integer to x
- `round(x,y)` – the value closest to x that has at most the y decimal places
- `min(x,y)` – the min value among x and y (this function can also be used for strings)
- `max(x,y)` – the max value among x and y (this function can also be used for strings)
- `timeString(x)` – the XML Schema xsd:dateTime representation (e.g. 2008-02-12T14:33:51.000+02:00) of x, which is interpreted as system time in milliseconds
- `timeString(x,format)` – the string representation of x, which is interpreted as system time in milliseconds, produced according to the format given (e.g. "dd.MM.yyyy") – consult Java API docs about the class *java.text.SimpleDateFormat.* Note that that timeString(x,"F") will give the day of the week: Monday 1 - Sunday 7.

- for strings: + (meaning concatenation) and also functions (for all except the last two, see Java API docs on the class *java.lang.String* for details):
  - `length(str)` – number of characters in str
  - `trim(str)` – the string with leading and trailing whitespaces trimmed off
  - `substring(str,x,y)` – substring of str starting at index x (zero-based) till index y-1. If the last operand is omitted, the substring is taken till the end of the string. If x or y < 0, the length of str is used instead (this is in contrast to Java function that throws an exception).
  - `indexOf(str,sub,x)` – first index (zero-based) at which the substring sub occurs in str starting from index x. If not found, returns -1. If the last operand is omitted, search is performed from the beginning of str
  - `lastIndexOf(str,sub,x)` – last index (zero-based) at which the substring sub occurs in str searching backwards from index x,. If not found, returns -1. If the last operand is omitted, search is performed from the end of str.
  - `startsWith(str,sub)` – `sapl:true` if str starts with the substring sub, and `sapl:false` otherwise
  - `endsWith(str,sub)` – `sapl:true` if str ends with the substring sub, and `sapl:false` otherwise
  - `contains(str,sub)` – `sapl:true` if str contains the substring sub, and `sapl:false` otherwise
  - `replace(str,sub1,sub2)` – string created from str by replacing every substring that equals to sub1 with sub2
  - `replaceAll(str,sub1,sub2)` – string created from str by replacing every substring that matches the regular expression given insub1 with sub2
  - `toLowerCase(str)` – string converted to the lower case
  - `toUpperCase(str)` – string converted to the upper case

  - `getNameSpace (uri)` – the namespace of the URI given (the part of URI from the beginning until but including # or the last /)
  - `getLocalName(uri)` – the local name of the URI (the part of URI after # or the last / till the end)
  - `timeMillis(x)` – the system time in milliseconds calculated from parsing x as an XML Schema xsd:dateTime object.
  - `timeMillis(x,format)` – the system time in milliseconds calculated from parsing x according to the format given (e.g. "dd.MM.yyyy") – consult Java API docs about the class *java.text.SimpleDateFormat.*

- specifically for context container IDs:
  - numberOfMembers (id) – the number of statements in this container
- specifically for variable names:
  - exists (variable) - `sapl:true` if variable is defined, and `sapl:false` otherwise (intended for use with sapl:Optional)
  - valueOf (variable) – the value of the variable whose name is held in the variable given.

Note that all the filtering conditions are checked only after all the normal conditions (including exclusive) are checked.

Creating new variables:

```
?x3 sapl:expression "?x1+?x2"
```

The construct above defines a new variable (must not be bound yet) and assigns to it the result of the evaluation of the expression given (can use all operations and functions described above). Such statements are processed *before starting processing the filtering conditions*, so the new variable can be utilized in those.

There is also a set of statistical predicates:

```
?x sapl:max ?y

?x sapl:min ?y

?x sapl:sum ?y

?x sapl:count ?y and ?x sapl:count "?y ?z"

?x sapl:countGroupedBy ?y and ?x sapl:countGroupedBy "?y
?z"
```

sapl:min, sapl:max and sapl:sum calculate a statistic based on of the set of all possible solutions. They work on numeric values as well as on strings. In case of strings, sapl:sum produces a concatenation of them. Note that possible sapl:orderBy and sapl:orderByDescending wrapping the query that uses sapl:sum will determine the order in which the strings are concatenated.

sapl:count selects from the set of all possible solutions the set of set of unique combinations of defined variables (same way as sapl:All do) and counts the number of members in this set.

While sapl:count, in a sense, creates groups of solutions (each group has a unique combination of defined variables) and counts the number of such groups, sapl:countGroupedBy counts the number of solutions in each such group.

Statistical predicates are processed *after processing the filtering conditions*. However, it is also provided for including conditions defined on the statistic variables using the same predicates as in filtering (those therefore are processed the last). Note that statistic predicates do not affect the process of selection among alternative solutions. Rather, the calculated statistical value is replicated to all the possible solutions. sapl:sum, sapl:max, sapl:min and sapl:count will put exactly the same value into all the solutions, sapl:countGroupedBy will put different values depending on the group to which the solution belongs.

Tips and tricks

This section should provide you with some advice and show you typical mistakes associated with Ubiware programming.

Common mistakes

Here is a list of the most common mistakes. We divided them into two types. First type are mistakes that happen when you don't pay attention, but in general when you see the mistake you can easily correct it. In other words you know why it is a mistake and you can easily correct it yourself. You just didn't notice. They are the most common mistakes. Here is the list:

- You forgot a dot at the end of the statement.
- You forgot to put a bracket at the end of the container
- You mistyped the name of a variable or some URI
- You are using a certain prefix that has not been declared previously
- You have a rule that is being executed endlessly. The reason is usually that in the right side you forgot to delete some belief that triggers it

To the second type of mistakes belong those that appear because you misunderstood something. Usually they are less obvious and you don't see immediately where the problem is. Here is the list of these mistakes based on our teaching experience:

- When using sapl:All, you are trying to insert it into wrong spot. sapl:All usually wraps either the whole left side of the implication or right side of the implication. It does not wrap the rule itself.
    - Correct: `{{ … query … } sapl:All ?x} => {… effect …}`
    - Wrong: `{{ … query … } => {… effect …}} sapl:All ?x`

Debugging tips

Throughout agent's life the belief storage is constantly changing. You can see the agent's beliefs in the beginning of its life from the initial agent script. However, often there is a need to see the beliefs of the agent during the execution. For this reason it is highly recommended to use DebugBehavior RAB. This RAB shows you the state of agent's mind – the content of agent's beliefs. Apart from this, it also allows you stop the agent's execution and make it progress one or more life cycle iterations to see the progress of the agent's execution in a step-by-step fashion. If you want to run the debugger, insert this RAB call into your code:

```
{ sapl:I sapl:do java:ubiware.core.behaviors.DebugBehavior }
sapl:configuredAs {p:stop sapl:is "false"}
```

For more information on DebugBehavior, please read the RAB reference guide.

Formatting tips

Even though it might seem as unimportant to some programmers, it is highly recommended to keep your code clean and tidy. Make it clear which code is just pure data and which code contains implications. Also, within the implication section try to distinguish and mark between different types of implications based on the function – e.g. GUI handlers, message handlers, reasoning rules, etc. An example of formatted code is in *Figure 5*.

```
@prefix sapl: <http://www.ubiware.jyu.fi/sapl#> .
@prefix java: <http://www.ubiware.jyu.fi/rab#> .
@prefix p: <http://www.ubiware.jyu.fi/rab_parameters#> .
@prefix x: <http://www.ubiware.jyu.fi/xxxxxxxxxx#> .

// just for debugging
{sapl:I sapl:do java:ubiware.core.behaviors.DebugBehavior} sapl:configuredAs sapl:null .

// message receiver for all messages
{sapl:I sapl:do java:ubiware.shared.MessageReceiverBehavior} sapl:configuredAs {
  p:waitOnlyFirst sapl:is "false"
} .


// GUI for deployment
{sapl:I sapl:do java:ubiware.shared.gui.buttonGUI.NewGUIBehavior} sapl:configuredAs {
  p:ID sapl:is x:GUI1 .
  p:title sapl:is "Deploy ubi package" .
  p:button sapl:is "Deploy" .
}.


{
  // rule that sends waits for 'send email' actions
  {
    ?msgID p:received {
      p:sender sapl:is ?sender .
      p:content sapl:is {
        ?who x:shouldReceive ?text
      }
    } .
  } => {
    // send email to the recipient
  } .
} sapl:is sapl:Rule .
```

Figure 5: Example of formatted SAPL code

Notice that every RAB call follows the same structure. The first like contains the name of the RAB and the lines below are parameters. Also, every rule consists of three main parts – left side, arrow with brackets and right side. If there is a container inside another container, it is indented to the right. This way you can easily see the "depth" of the container.

Common misconceptions

There are a few improperly understood concepts. This is a short list of things you should keep in mind:

- S-SAPL is not an imperative language like Java, C or Pascal. Therefore don't try to "force" the language to work in an imperative way unless there is no other way to solve the problem. Usually this brings more problems than benefits.
- S-APL does not store data into variables like most of the imperative languages. It stores the data in form of beliefs, which are triples. Therefore the unit of data is a belief ("sentence"), not a variable ("word").

Advanced topics

This section contains topics that are not used very often when programming in Ubiware, but they bring some additional features. This section is not indented for a beginner. We recommend it only for programmers that have understood the previous topics and already have at least some practical experience with S-APL and Ubiware platform.

Agent's desires/goals

There is a special container used to express agent's goals (desires). This container is handled in a special way – it never disappears, even if it's empty and it has a set of beliefs that an agent is trying to have in its G context. An example of such goal would be:

```
sapl:I sapl:want {:John :Loves :Mary}
```

This goals means that the agent wants to have `:John :Loves :Mary` in its beliefs.

Action that attempts to achieve a goal, fulfill a commitment, or handle an event:

```
{ sapl:I sapl:want {:John :Loves :Mary} } >> {...}
```

`>>` is the shorthand for `sapl:achievedBy`. When the agent's engine executes such a statement, it treats it the same way as `sapl:implies (=>)`, but in addition:

- moves all goals, i.e. sapl:I sapl:want {} referenced to in the subject context to the special context for goals under work, which is "sapl:I sapl:achieving {}".
- removes all unconditional commitments to an action, i.e. {} sapl:configuredAs {} referenced in to in the subject context (it is possible if actions has some namespaces that are neither `java:` nor default empty ":")

- removes all the interface events, i.e. {} sapl:eventFrom … referenced to in the subject context.

sapl:achievedBy is introduced for the sake of shortness. In a sense, it has the meaning of logical transition from its left side to its right side, rather than simple logical implication as sapl:implies.

Advanced concepts related to implications

Matching with variables standing for statements:

```
{ {:John :Loves *} sapl:ID ?x } :accordingTo :Bill
```

In this case, ?x will to bound to the ID (or IDs if wrapped with sapl:All) of the statement like "`:John :Loves :Mary`".

```
{?x rdf:subject :John} :accordingTo :Bill
```

In this case, ?x will be bound to the ID (or IDs if wrapped with sapl:All) of all the statements that have as their subject :John. Similarly, one can use rdf:predicate and rdf:object.

```
{?x sapl:is sapl:true} :accordingTo :Bill
```

In this case, ?x will be bound to the ID (or IDs if wrapped with sapl:All) of the first found statement in the context container.

```
?c sapl:hasMember ?x
```
or
```
?x sapl:memberOf ?c
```

In this case, ?x will be bound the ID (or IDs if wrapped with sapl:All) of the first found statement in the context container identified by variable ?c.

It is possible to define queries like

```
{{?x sapl:is sapl:true} :accordingTo :Bill.
     {?x sapl:is sapl:true} :accordingTo :John } => {...}
```

Such query is evaluated as true if at least one belief that is found in the context container {} :accordingTo :Bill has a match in the context container {} :accordingTo :John.

It is also possible to add or remove beliefs using:

```
{...} => {?x sapl:is sapl:true}

{...} => {sapl:I sapl:add ?x}
```

```
{...} => {sapl:I sapl:remove ?x}
```

The difference between the first two is that in the first case, the existing statement will be linked to G, while in the second case, the copy of it will be created and added. To remind, sapl:remove will use ?x (statement with the sub-tree it starts) as the pattern. If the goal is to remove the statement itself, sapl:erase has to be used instead.

Using `sapl:memberOf` or `sapl:hasMember`, it is possible to link existing statements to already existing context containers.

Using `rdf:subject`, `rdf:predicate` and `rdf:object`, it is possible to change those values for a statement.

Prerequisites:

```
{ {...} => {...} } sapl:requires {...}
```

When the agent's engine finds such a belief in G, it evaluates first the conditional commitment in the subject context. If it is to be executed, the engine checks the conditions in the object context (continuing with the set of solutions (combinations of bindings of the variables) defined on the left of =>). If they are all met, the commitment is executed. If some are not met, they are wrapped as "sapl:I sapl:want {}" (i.e. as goals) and added to G. This addition is not done, if the goals are already present in "sapl:I sapl:want {}" or "sapl:I sapl:achieving {}".

If the left side of => uses `sapl:All` (see Section **Error! Reference source not found.**), a goal is dded for every possible solution. Using `sapl:All` on the right side of => will have no such effect.

Commitment with a guard condition:

```
{ {...} => {...} } sapl:is sapl:true ;
                   sapl:existsWhile {...}
```

When the agent's engine finds an sapl:existsWhile belief in G and finds out that a condition in the object context is not met, it removes from G all the beliefs specified in the subject context. Normally, this is to be used as mechanism for dropping unachievable or not-relevant-anymore commitments. However, this can also be used for specifying beliefs that depend on some other beliefs.

Special beliefs

Quantified beliefs:

```
{:John :Loves ?x. ?x :Is :Girl} sapl:forSome ?x
```

This one means “John loves some girl”.

```
{ {?x :Is :Man} => {?x :Is :Mortal} } sapl:forAll ?x
```

This one means “Every man is mortal”.

Note that UBIWARE platform at the current stage does not provide any implicit reasoning. This means that while it is possible to query for existence of a quantified belief (e.g. “does the agent believe that every man is mortal?”), the reasoning is not provided for querying for the consequences of such a belief (e.g. “does the agent believe that John is mortal?”) .

Old concepts that will not be supported in the future

Local IDs

Creating links to statements through local IDs

```
{:John :Loves :Mary} sapl:ID __S1.

{__S1 sapl:is sapl:true} :accordingTo :Bill
```

This case is interpreted the same as `{:John :Loves :Mary} sapl:is sapl:true; :according to :Bill`. This allows, however, to define more complex graphs if a need will arise. The ID __S1 is local in a sense that it is used at the parsing stage only and does not correspond to actual ID that will be given to the created statement. This means that a local ID has the scope of a given single S-APL document only. Local IDs can only be referred to in “<id> sapl:is sapl:true” constructs, not in any others.

Note that putting several statements inside {} sapl:ID <id> does not make sense, but does not cause an error. If this is done, the ID will be linked to the first of the statements.