UBIWARE Deliverable D1.3:

UBIWARE Platform Prototype v.1.0

May, 2008

Date	May 8, 2008
Document type	Report
Dissemination Level	UBIWARE project consortium
Contact Author	Vagan Terziyan
Co-Authors	Artem Katasonov, Olena Kaykova, Oleksiy Khriyenko, Sergiy Nikitin
Work component	WP1-WP5
Deliverable Code	D1.3
Deliverable Owner	IOG, JYU
Deliverable Status	Mandatory, Internal
Intellectual Property Rights	Unaffected

Table of Contents

Introduction

The UBIWARE project aims at a new generation middleware platform which will allow creation of self-managed complex industrial systems consisting of distributed, heterogeneous, shared and reusable components of different nature, e.g. smart machines and devices, sensors, actuators, RFIDs, web-services, software components and applications, humans, etc. The technologies, on which the project relies, are the Software Agents for management of complex systems, and the Semantic Web, for interoperability, including dynamic discovery, data integration, and inter-agent behavioral coordination.

Work in this project is divided into seven work packages which are running in parallel:

1. Core agent-based platform design
2. Managing Distributed Resource Histories
3. Security in UBIWARE
4. Self-Management and Configurability
5. Context-aware Smart Interfaces for Integrated Data
6. Middleware for Peer-to-Peer Discovery
7. Industrial cases and prototypes.

Work-packages 1 through 6 are research work packages; however, the research efforts are combined with agile software development processes. Prototypes of the UBIWARE platform, integrating the work in these 6 work packages at different levels of their readiness, are developed during each project year, as UBIWARE 1.0, UBIWARE 2.0 and UBIWARE 3.0.

UBIWARE deliverable D1.1 reported on the research results from work packages 1 through 5 (it was decided not to perform the work at the WP6 during the first project year due to limitation in resources). This deliverable, D1.3, presents the integrated development results from those work packages, i.e. the current state of the UBIWARE platform prototype. Naturally, during the development stage, solutions described previously in D1.1 have undergone some changes and improvements.

Formally, D1.3 consists of the software system itself and this accompanying report. The present report includes a separate chapter for the results of every work-package involved. Note that the work package 1 sets the general architecture of UBIWARE, and therefore the description of results from WP1 is important for understanding the results from other WPs.

UBIWARE Deliverable D1.3:
Workpackage WP1:
Task T1.2_w1:

1 UbiCore – Core Distributed AI platform design

The main objective of the core platform is to ensure a predictable and systematic operation of the components and the system as a whole by:

- enforcing that the smart resources, while might to have own "personal" goals, act as prescribed by the roles they play in a organization and by general organizational policies,
- maintaining the "global" ontological understanding among the resources, meaning that a resource A can understand all of (1) the properties and the state of a resource B, (2) the potential and actual behaviors of B, and (3) the business processes in which A and B, and maybe other resources, are jointly involved

During WP1's Year 1 (the *Representation* phase), the Semantic Agent Programming Language (S-APL) was developed which is a semantic tool sufficient for describing all of the following: resources' properties, agents' behaviors, organizational policies, business processes, ontologies, etc. Having a common language for all those ensures that any type of information in UBIWARE is semantic and facilitates intertwining those: e.g. one can easily prescribe that in a certain business process, if some properties of some resource have some certain values this must lead to some specific action taken by some agent unless prohibited by some organizational (e.g. security) policy.

1.1 Platform Architecture

The central to the core platform is the architecture of a UBIWARE agent depicted in the Figure 1. The basic 3-layer agent structure is similar to, for example, the Agent Factory's ALPHA/AFAPL (see Section 2). There is the behavior engine implemented in Java, a declarative middle layer, and a set of sensors and actuators which are again Java components. The latter we refer to as *Reusable Atomic Behaviors (RABs)*. We do not restrict RABs to be only sensors or actuators, i.e. components sensing or affecting the agent's environment. A RAB can also be a reasoner (data processor) if some of the logic needed is impossible or is not efficient to realize with the S-APL means, or if one wants to enable an agent to do some other kind of reasoning beyond the rule-based one.

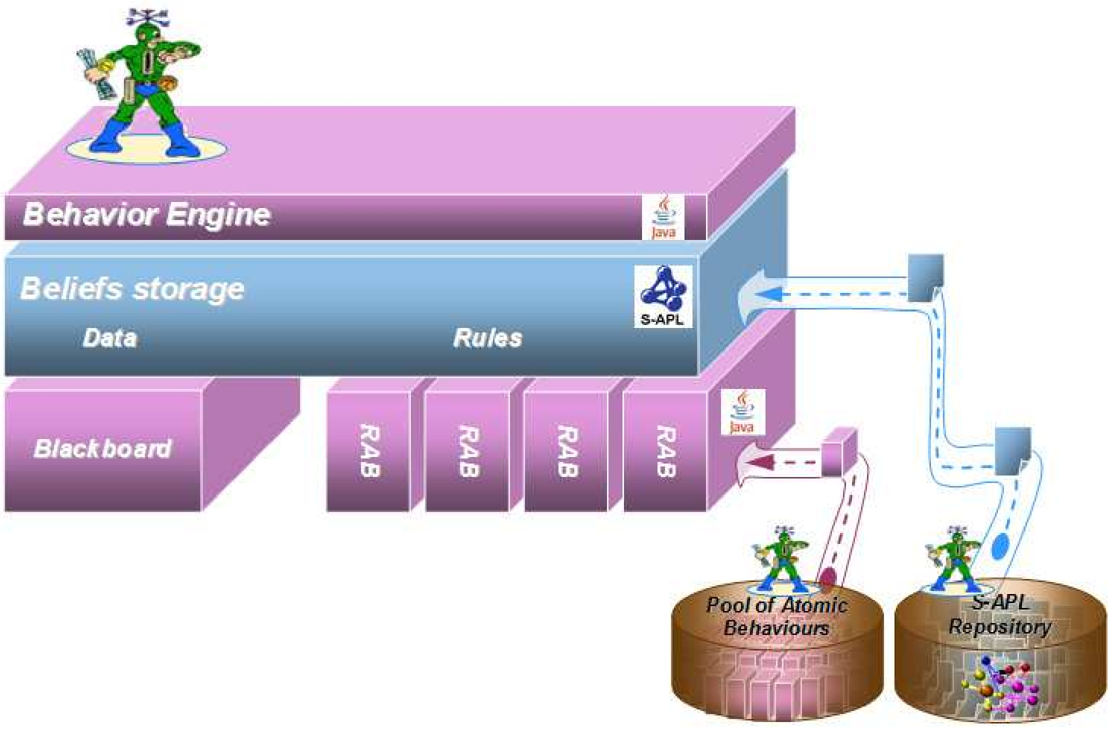


Figure 1. The UBIWARE agent architecture

The middle layer is the S-APL beliefs storage. The main factor that differentiates S-APL from traditional APLs like AgentSpeak or ALPHA is that S-APL is RDF-based. An additional immediate advantage is that in S-APL the difference between the data and the program code (rules and plans) is only logical but not any principal. They use the same storage, not two separate. This also means that: a rule upon its execution can add or remove another rule, the existence or absence of a rule can be used as a premise of another rule, and so on. None of these is normally possible in traditional APLs treating rules as special data structures principally different from normal beliefs which is n-ary predicates. S-APL is a very symmetric in this respect – anything that can be done to or with a simple statement can also be done to any belief structure of any complexity.

Technically, our implementation is built on the top of the Java Agent Development Framework (JADE, Bellifemine et al. 2007), which is a Java implementation of IEEE FIPA specifications. The S-APL behavior engine is an extension (subclass) of JADE's Agent class, while the base class for all RABs is an extension of JADE's SimpleBehavior class.

As Figure 1 stresses, an S-APL agent can obtain the needed data and rules not only from local or network documents, but also through querying S-APL repositories. Such a repository, for example, can be maintained by some organization and include prescriptions (lists of duties) corresponding to the organizational roles that the agents are supposed to play. In our implementation, such querying is performed as inter-agent action with FIPA ACL messaging, but does not involve any query or content languages beyond S-APL itself. As can be seen from Figure 1, agents also can to load RABs remotely. This is done as an exception mechanism triggered when a rule prescribes engaging a RAB while the agent does not have it available. Thus, organizations are able to provide not only the rules to follow but also the tools needed for that.

We also equip each agent with a blackboard, through which RABs can exchange arbitrary Java objects. Similar solution can be found e.g. in the Cougaar framework (Helsinger, 2004). The reason for that is not to unnecessarily restrict the range of applications that could be realized with S-APL. Without such a blackboard, RABs would be always forced to translate all data into RDF (even when the S-APL code of the agent is not concerned with the content of data, or could not process it) or at least serialize it as text string to put the as object of a statement. This could restrict the performance and, more importantly, significantly reduce the wish to use S-APL. Blackboard is also necessary to accommodate objects like Socket, HttpServletResponse or similar to enable an agent to process and respond to HTTP requests, which may be needed in many applications. With the blackboard extension, the developers of a specific application can use S-APL in different ways:

- Semantic Reasoning. S-APL rules operating on S-APL data.
- Semantic Data. RABs (i.e. Java components) operating on S-APL semantic data.
- Workflow management. RABs operating on Java blackboard objects, with S-APL used only as workflow management tool, specifying what RABs are engaged and when.
- Any combination of the three options above.

1.2 Semantic Agent Programming Language (S-APL)

This section briefly describes S-APL language. For a detailed description, see (Katasonov, 2008).
S-APL has as an axiom that everything inside an agent's mind is a belief. All other mental attitudes such as goals, commitments, behavioral rules are just compound beliefs. Thus, an S-APL document is basically a statement of some agent's current or expected (by an organization) beliefs.

S-APL is based on Notation3 (N3) (http://www.w3.org/DesignIssues/Notation3.html). N3 was proposed by Tim Berners-Lee as a more compact, better readable and more expressive alternative to the dominant notation for RDF, which is RDF/XML. One feature of N3, which goes beyond the plain RDF, is the concept of formula that allows RDF graphs to be quoted within RDF graphs, e.g. {:room1 :hasTemperature 25} :measuredBy :sensor1. An important convention is that a statement inside a formula is not considered as asserted, i.e., as a general truth. In a sense, it is a truth inside a context defined by the statement about the formula and the outer formulas. This is in contrast to the plain RDF where every statement is asserted as a truth. In S-APL, we refer to formulae as context containers. The top level of the S-APL document, i.e. of what is the general truth for the agent, we refer to as general context or just G. Below, we describe the main constructs of S-APL. We use three namespaces: ”sapl:” for S-APL constructs, ”java:” for RABs, and ”p:” for RAB parameters. Empty namespace ”:” is used for resources that are assumed to be defined elsewhere locally.

The two constructs below are equivalent and define a simple belief. The latter is introduced for syntactic reasons.

```
:room1 :hasTemperature 25

{:room1 :hasTemperature 25} sapl:is sapl:true
```

The next two constructs add context information:

```
{:room1 :hasTemperature 25} :measuredBy :sensor1

{:room1 :hasTemperature 25} sapl:is sapl:true ; :measuredBy :sensor1
```

The former states that ”sensor1 measured the temperature to be 25” without stating that ”the agent believes that the temperature is 25”. In contrast, the latter states both. This demonstrates a specific convention of S-APL: rather than doing several statements about one container, ”{...} P O; P O” leads to linking the statements inside the formula to two different containers. Then, using sapl:true it is also possible to link some statements to a container and to one of its nested containers.

The goals of the agent and the things that the agent believes to be false are defined, correspondingly, as:

```
sapl:I sapl:want {:room1 :hasTemperature 25}

{:room1 :hasTemperature 25} sapl:is sapl:false
```

sapl:I is a special resource that is defined inside the beliefs of an agent owl:sameAs the URI of that agent. A specific convention of S-APL is that e.g. ”sapl:I sapl:want {A B C}. sapl:I sapl:want {D E F}” is the same as ”sapl:I sapl:want {A B C. D E F}”. In other words, the context containers are joined if they are defined through statements with the same two non-container resources.

The commitment to an action is specified as follows:

```
{sapl:I sapl:do java:ubiware.shared.MessageSenderBehavior}

sapl:configuredAs {

          p:receiver sapl:is :John.

          p:content sapl:is {:get :temperatureIn :room1}

          sapl:Success sapl:add {:John :informs :temperature}

}
```

The ”java:” namespace indicates that the action is a RAB. Otherwise, it would be an abstract action (capability) that a rule was supposed to translate into a plan. When the behavior engine finds such a belief in G, it executes the RAB and removes the commitment. 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. Using such statements, one can easily specify sequential plans: {sapl:I sapl:do ...} sapl:configuredAs {... sapl:Success sapl:add {{sapl:I sapl:do ...} sapl:configuresAs {...}}} .

The commitments to mental actions are as follows:

```
sapl:I sapl:remove {:John :informs :temperature}
sapl:I sapl:add {:John :informs :temperature}
```

sapl:remove uses its object as a pattern that is matched with G and removes all beliefs that match. sapl:add does not need to be normally used, since just stating something is the same as adding it to beliefs. This construct is needed when one wants to postpone the creating of the belief until the stage of the agent run-time cycle iteration when commitments are treated (see in the end of this section), or when one uses the object as a variable holding the ID of a statement or a container (see below).

The conditional commitment is specified as:

```
{ {?room :hasTemperature ?temp} :measuredBy *. ?temp > 30 }
  => {...}
```

?room and ?temp are variables. => and > are shorthands for sapl:implies and sapl:gt. * means ”anything”. The object of sapl:gt and other filtering predicates (>=, <, <=, =, !=) is an expression that can utilize arithmetic operations, functions like abs, floor, random, etc. and string-processing functions like length, startsWith, substring, etc. When the behavior engine finds in G a belief as above and finds out that all the conditions in the subject context container are met, it copies to G, substituting variables with their values, all the beliefs from the object container. Those can be plain beliefs and/or commitments, unconditional or conditional. S-APLallows a variable value to substitute a part of a resource, e.g. ”logs/?today/received”. Such a liberty is in contrast with, e.g., the approach used in the CWM semantic reasoner (http://www.w3.org/2000/10/swap/doc/cwm) where a variable value can only be a substitute for the whole resource; however, it was shown to greatly simplify the programming.

As with any commitments, the conditional commitment is removed after successful execution. In order to create a persistent rule, the => statement has to be wrapped as:

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

It is possible to define a guard for a conditional commitment so it is dropped if the guard becomes false:

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

We introduce sapl:existsWhile as a way of creating commitment guards because APLs normally require the ability of defining commitments that are dropped as unachievable or not relevant anymore. However, in S-APL sapl:existsWhile can of course be used with any type of beliefs.

There are also a couple of alternatives to =>:

```
{...} -> {...} ; sapl:else {...}
{sapl:I sapl:want {...} } >> {...}
```

-> and >> are the shorthands for sapl:impliesNow and sapl:achievedBy. -> specifies a conditional action rather than a commitment: it is checked only once and removed even if it was false. One can also combine it with sapl:else to specify the beliefs that have to be added if the condition was false. >> works the same as => with the only difference that if the left side of it refers to some goals, commitments or interface (GUI or HTTP) events, those are removed automatically when the rule fires. Thus, >> has the meaning of logical transition rather than pure inference.

A specific convention of S-APL is that if there are several possible solutions to the query in the left side of =>, -> or >>, the right side is copied by default for the first-found solution only. One can use sapl:All wrappings to define that the right part has to be copied several times: for every unique value of some variable of every unique combination of the values of some variables. These wrapping can be used in either the left or the right side:

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

sapl:All on the right side is allowed to enable defining different wrappings for different (top-level) resulting statements, e.g. ”{...} => {X Y Z. {{?x L ?y } sapl:All ?y. A B ?x} sapl:All ?x}”. On the left side of =>, sapl:All must always wrap the whole contents of the container. Other solutions set modifier wrappings are also available, namely sapl:OrderBy, sapl:OrderByDescending, sapl:Limit, and sapl:Offset. The meaning of those are the same as of their equivalents in SPARQL (http://www.w3.org/TR/rdf-sparql-query/). One can also wrap a condition in the left side of => with sapl:Optional to have the same effect as SPARQL’s OPTIONAL, and connect two conditions with sapl:or to have the same effect as SPARQL’s UNION. It is also possible to specify exclusive conditions, i.e. ones that must not be know to be true, by using the wrapping sapl:I sapl:doNotBelieve {...}.

One can also define new calculated variables:

```
{?person :height ?h. ?feet sapl:expression ”?h/0.3048”. ?m sapl:min ?feet } => {...}
```

sapl:expression gives to the new variable the value coming from evaluating an expression. sapl:min is a special predicate operating on the set of matching solutions rather than on a particular solution. The other predicates from the same group are sapl:max, sapl:sum, sapl:count (number of groups when grouped by values of some variables) and sapl:countGroupedBy (number of members in each group).

Variable can also refer to IDs of statements and context containers, and one can use the predicates sapl:hasMember, sapl:memberOf, rdf:subject, rdf:predicate and rdf:object. After obtaining the ID of the container with ”?x :according to :Bill”, one can do the following things:

```
{ ... {?x sapl:is sapl:true} :accordingTo :John } => {...}
?x sapl:is sapl:true
sapl:I sapl:add ?x
sapl:I sapl:remove ?x
sapl:I sapl:erase ?x
?x sapl:hasMember {:room1 :hasTemperature 25}
```

The first construct defines a query that is evaluated as true if any belief that is found in the context container ?x has a match in the context container {} :accordingTo :John. The second one links the statements from ?x to G, while the third copies them to G. The fourth uses the contents of ?x as the pattern for removing beliefs from G, while the fifth removes the container ?x itself. Finally, the sixth add to the container ?x a new statement.

There are several ways to create a variable holding IDs of some statements (we hope that the meanings are clear):

```
{ {* :hasTemperature 25} sapl:ID ?x } :accordingTo :Bill
{?x rdf:predicate :hasTemperature} :accordingTo :Bill
{?x sapl:is sapl:true} :accordingTo :Bill
?c :accordingTo :Bill. ?c sapl:hasMember ?x
```

One can use a query like ”{ {?x sapl:is sapl:true} :according to :Bill. {?x sapl:is sapl:true} :accordingTo :John } => {...}”, which is evaluated as true if there is at least one belief from the first container that has a match in the second container. One can also use sapl:true, sapl:add, sapl:remove, sapl:erase, sapl:hasMember and sapl:memberOf to do the same things as explained above, but for one statement only.

There is also possibility to define meta-rules:

```
{ {...} => {...} } 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 below). Therefore, the actual difference is in intended purpose: meta-rules are supposed to modify/block normal rules or commitments, to implement some organizational, e.g. security, policies.

The final remark on S-APL syntax is that the behavior engine automatically generates and maintains several special beliefs that can be used in queries. One example is ”sapl:Now sapl:is <current system time in milliseconds>”.

1.3 S-APL engine run-time cycle

Figure 2 presents a simplified view of the run-time cycle that the UBIWARE agent's behavior engine implements to act based on S-APL behavior models.

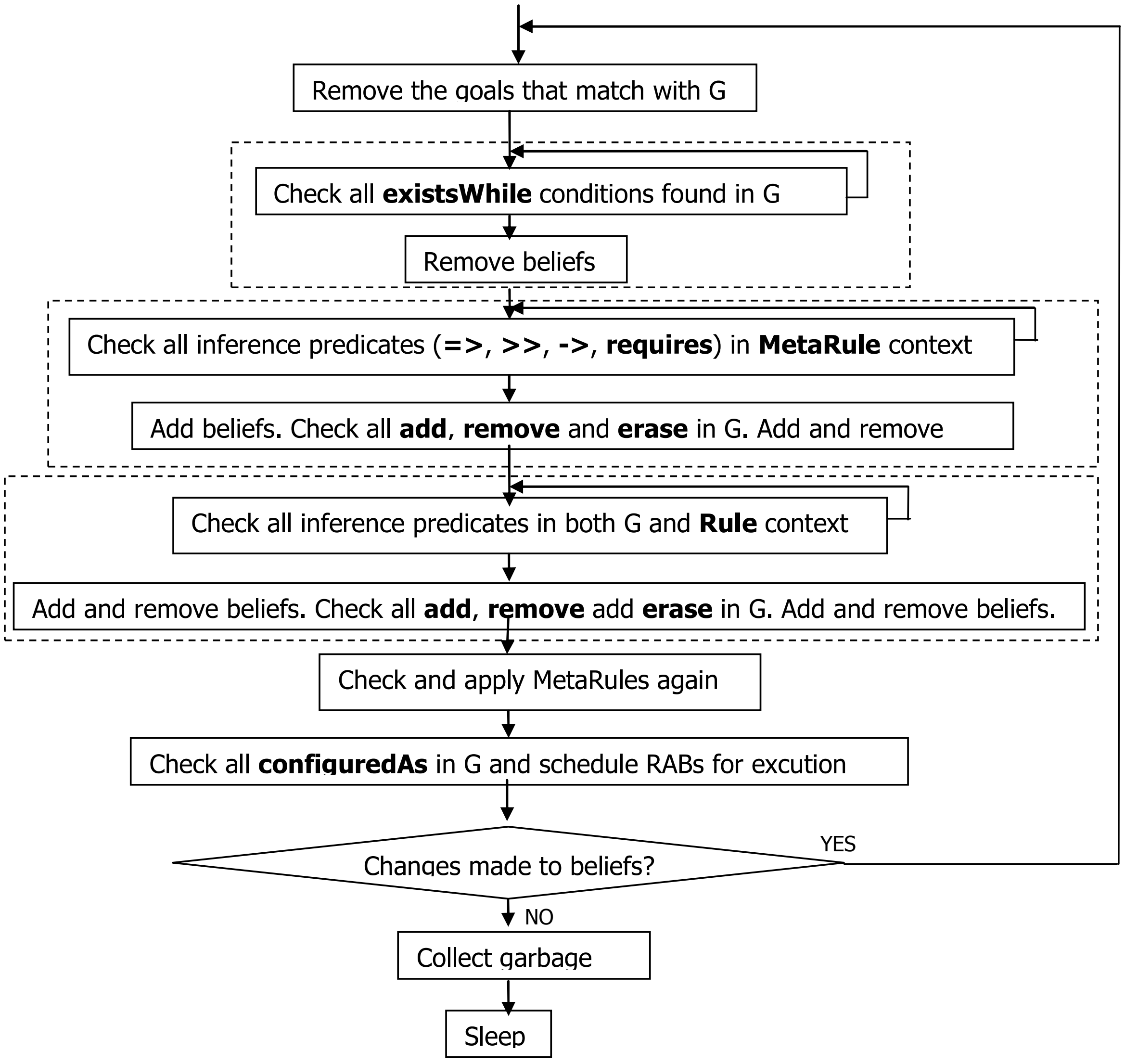

Remove the goals that match with G
Check all existsWhile conditions found in G
Remove beliefs
Check all inference predicates (=>, >>, ->, requires) in MetaRule context
Add beliefs. Check all add, remove and erase in G. Add and remove
Check all inference predicates in both G and Rule context
Add and remove beliefs. Check all add, remove add erase in G. Add and remove beliefs.
Check and apply MetaRules again
Check all configuredAs in G and schedule RABs for excution
Changes made to beliefs?
YES
NO
Collect garbage
Sleep

Figure 2. S-APL run-time cycle

In each iteration, the engine performs the following:

- First, it checks and removes all goals, both in "sapl:I sapl:want {}" or "sapl:I sapl:achieving {}" that match with G, i.e. have already been achieved.
- Then, it checks all sapl:existsWhile conditions found in G and removes the beliefs, whose existsWhile conditions are not met anymore.

- Then, it checks all the inference predicate (sapl:implies =>, sapl:impliesNow ->, sapl:achievedBy >>, and sapl:requires) found in {…} sapl:is sapl:MetaRule context. If some rules are executed, the result is that some new beliefs are added to G. If there are several executable rules in one iteration, all are executed. Immediately after that, the engine check all sapl:add, sapl:remove and sapl:erase commitments in G, and then adds and remove the needed beliefs as well as the commitments themselves.
- Then, exactly the same is done for inference predicates in both is G (conditional commitments) and in the context {…} sapl:is sapl:Rule (behavior rules). Beliefs are added only after both G and Rules are processed. Then, also the executed conditional commitments are removed.
- Then, the MetaRules are processed again (to block/modify any action commitments that were made).
- Finally, all the {sapl:I sapl:do …} sapl:configuredAs (action commitments) that are found in G are executed (this leads to putting the corresponding RAB into the execution queue) and then removed.
- If any changes to beliefs were made, the engine schedules the new iteration. The new iteration will actually start after all the active RABs will be executed once.
- Otherwise, the engine collects garbage (unreferenced contexts and statements) and blocks the run-time cycle until a new message arrives or some of running RABs makes some modification to the beliefs. If there are any active RABs, they will continue execution.

As it stressed in the Figure 2, actual modifications to the belief storage (adding and removing) are to be done after all the checks are completed. In this way, e.g. the effect of execution of one rule could not prevent another executable rule from execution.

Garbage collection involves, in an iterative fashion:

- Removing all contexts that are not referenced, i.e. do not appear as the subject or the object of any statement.
- Removing all statements that are not referenced, i.e. are not linked to any context
- Removing all empty contexts, i.e. not having any member statements.

In result, any "hanging" sub-graph of beliefs will be removed as garbage. Therefore, it is always enough to explicitly remove just the top statement, e.g. {} sapl:configuredAs {}, and let the garbage collector to remove the rest – the contexts appearing as the subject and the object.

1.4 UBIWARE Platform elements

The UBIWARE agent architecture depicted in Figure 1 implies that a particular UBIWARE-based software application will consist of a set of S-APL documents (data and behavior models) and a set of specific atomic behaviors needed for this particular application.

Since reusability is an important UBIWARE concern, it is reasonable that the UBIWARE platform provides some of those ready-made. Therefore, the UBIWARE platform as such can be seen as consisting of the following three elements:

- The behavior engine
- A set of “standard” S-APL models.
- A set of “standard” RABs.

Then, also the development outputs of other UBIWARE work packages are exactly some sets of such “standard” S-APL models and RABs that can be used by the developers to embed into their applications certain UBIWARE features such as security or flexible declarative (re-)configuration.

The table below lists the Reusable Atomic Behaviors (RABs) that are included into UBIWARE 1.0. For detailed description of those, see (Katasonov, 2008).

Print	Prints to the screen (and to the log) specified text.
PrintBeliefs	Prints to the screen (and to the log) the contents of specified context container.
ModifyGUI	Sends a request to a GUI to perform some actions, mainly related to modifying GUI (enabling/disabling elements, putting text to labels, etc.).
ExternalAppStarterBehavior	Starts an external software application.
UnzipperBehavior	Unzips a ZIP archive.
HttpDataFetcherBehavior	Downloads a document (e.g. an HTML page) from an Internet server.
EmailSenderBehavior	Sends an email.
FilesSensorBehavior	Generates a belief structure describing the contents of a file-system folder, including all the subfolders.
XmlReaderBehavior	Reads an XML file and generates a predefined hierarchical belief structure.
XmlWriterBehavior	Writes XML data based on a predefined belief structure.
TextTableReaderBehavior	Reads a text file containing a data-table, for example a CSV file.
ExcelReaderBehavior	Reads data from a worksheet of a Microsoft Excel file.
SQLReaderBehavior	Submits an SQL query to a relational database and generates from the results a hierarchical belief structure.
RdfXmlReaderBehavior	Reads am RDF/XML file and loads its contents into the agents’ beliefs into the specified context container.

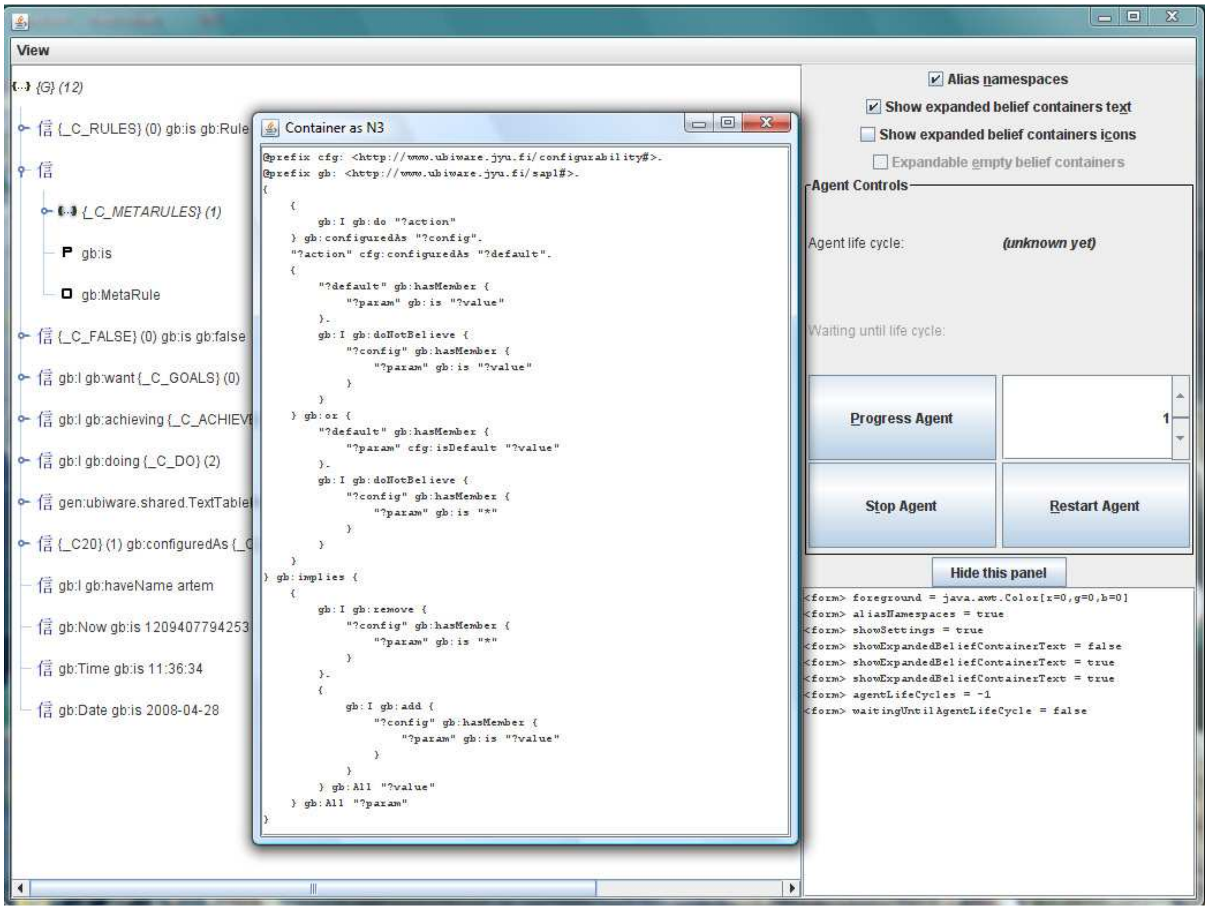
BeliefsBackupBehavior	Saves a sub-graph of agents' beliefs as an S-APL Notation3 string.
BeliefsLoadBehavior	Loads an S-APL Notation3 string (given directly or from file) into the agents' beliefs.
HttpResponseSenderBehavior	Sends an HTTP response to an request arrived earlier.
CreateAgentBehavior	Creates a new agent in the same JADE container.
MessageSenderBehavior	Sends a message to another agent on the same platform.
MessageReceiverBehavior	Listens for incoming message(s) matching defined parameters.
SecurityCheckBehavior	Checks with Directory Facilitator whether the agent in question plays a role that would authorize it for something.
DFRegisterBehavior	Register with the Directory Facilitator the services (e.g. roles) provided by this agent.
DFLookupBehavior	Finds with the Directory Facilitator names of agents playing a particular role.
SeveralButtonsGUI	Gives a very simple interface with N buttons and possibility to enable and disable those buttons.
DebugBehavior	Starts the visualizer / debugger interface (see next subsection).

The following table below lists the standard S-APL models provided with UBIWARE 1.0. Some of those are described in chapters on other WPs.

startup	Enables agents to access S-APL documents remotely, from an OntologyAgent.
RABLoader	Enables agents to access RABs remotely, from a RABRepositoryAgent.
OntologyAgent	The script for an OntologyAgent.
RABRepositoryAgent	The script for an RABRepositoryAgent.
Reasoning.RDFSReasoner	Implements RDF Schema reasoning rules (class-subclass relation, etc.)
Reasoning.OWLReasoner	Implements a subset of OWL reasoning rules (symmetric, inverse and transitive properties, sameAs)
Communication.Listener	Receives messages (see Chapter 2)
Communication.Informer	Agent answering queries (see Chapter 2)
Communication.Follower	Agent doing what it is asked to do (see Chapter 2)
Communication.Believer	Agent adding to its beliefs what it was informed about (see Chapter 2)

Communication.Ontology	Defines an ontology of communicative actions for, e.g., policy-based control (see WP3).
Security.SBACReasoner	Semantics-Based Access Control Reasoner (see Chapter 3).
Configurability.RABConfigurator	Modifies unconditional commitments to actions (e.g. RABs) based on the global configuration settings for those actions (see Chapter 4)

1.5 Visualizer / debugger



{G} (12)
{_C_RULES} (0) gb:is gb:Rule
{_C_METARULES} (1)
gb:is
gb:MetaRule
{_C_FALSE} (0) gb:is gb:false
gb:I gb:want {_C_GOALS} (0)
gb:I gb:achieving {_C_ACHIEVE
gb:I gb:doing {_C_DO} (2)
gen:ubiware.shared.TextTable
{_C20} (1) gb:configuredAs {_C
gb:I gb:haveName artem
gb:Now gb:is 1209407794253
gb:Time gb:is 11:36:34
gb:Date gb:is 2008-04-28

```
@prefix cfg: <http://www.ubiware.jyu.fi/configurability#>.
@prefix gb: <http://www.ubiware.jyu.fi/sapl#>.
{
    {
        gb:I gb:do "?action"
    } gb:configuredAs "?config".
    "?action" cfg:configuredAs "?default".
    {
        "?default" gb:hasMember {
            "?param" gb:is "?value"
        }.
        gb:I gb:doNotBelieve {
            "?config" gb:hasMember {
                "?param" gb:is "?value"
            }
        }
    } gb:or {
        "?default" gb:hasMember {
            "?param" cfg:isDefault "?value"
        }.
        gb:I gb:doNotBelieve {
            "?config" gb:hasMember {
                "?param" gb:is "*"
            }
        }
    }
} gb:implies {
    {
        gb:I gb:remove {
            "?config" gb:hasMember {
                "?param" gb:is "*"
            }
        }.
        {
            gb:I gb:add {
                "?config" gb:hasMember {
                    "?param" gb:is "?value"
                }
            }
        } gb:All "?value"
    } gb:All "?param"
}
```

[✓] Alias namespaces
[✓] Show expanded belief containers text
[ ] Show expanded belief containers icons
[ ] Expandable empty belief containers

Agent life cycle: *(unknown yet)*

Waiting until life cycle:

1

```
<form> foreground = java.awt.Color[r=0,g=0,b=0]
<form> aliasNamespaces = true
<form> showSettings = true
<form> showExpandedBeliefContainerText = false
<form> showExpandedBeliefContainerText = true
<form> showExpandedBeliefContainerText = true
<form> agentLifeCycles = -1
<form> waitingUntilAgentLifeCycle = false
```

Figure 3. The visualizer interface

In the future, an important additional element of the UBIWARE platform is a set of tools facilitating the development of UBIWARE-based applications. One tool in this group has already been under development and its first version is included with UBIWARE 1.0. It is the beliefs-visualizer / debugger interface. Any agent can start its own visualizer (it is done by using DebugBehavior RAB) to enable the developer to monitor the agent's beliefs and also to control the agent's execution.

The screenshot of the visualizer is shown in Figure 3. The current state of the visualizer:

- Shows the content of the beliefs storage of the agent as a tree hierarchy (with ability to expand / collapse branches).
- Upon request, presents any given branch of the beliefs storage as an S-APL Notation 3 text.
- Enables the developer to execute the agent step-by-step, with step meaning a S-APL engine run-time cycle iteration (as in Section 1.3)

UBIWARE Deliverable D1.3:
Workpackage WP2:
Task T1.2_w2:

2 UbiBlog – Managing Distributed Resource Histories

In UBIWARE, every resource is represented by a software agent. Among major responsibilities of such an agent is monitoring the condition of the resource and the resource's interactions with other components of the system and humans. The beliefs storage of the agent will, therefore, naturally include the history of the resource, in a sense "blogged" by the agent. Obviously, the value of such a resource history is not limited to that particular resource. A resource may benefit from the information collected with respect to other resources of the same (or similar) type, e.g. in a situation which it faces for the first time while other may have faced that situation before. Also, mining the data collected and integrated from many resources may result in discovery of some knowledge important at the level of the whole ubiquitous computing system. A scalable solution requires mechanisms for inter-agent information sharing and data mining on integrated information which would allow keeping the resource histories distributed without need to copy those histories to a central repository.

During WP2's Year 1 (the *Sharing* phase), needed mechanisms were designed for effective and efficient sharing of information between different agents, e.g. representing different resources. S-APL was used as the communication content language, which has enabled:

- One agent to query another agent for some information, using the query constructs similar to that of SPARQL but with even wider range of possible filtering conditions
- One agent to inform another agent, i.e. to proactively push some information of any complexity.
- One agent to request another agent to perform some actions, either an atomic behavior or a complex plan involving a set of rules and atomic of complex behaviors.

As to UBIWARE 1.0 platform, the development contribution of WP2 consists of:

- A set of standard S-APL models: Communication.Listener, Communication.Informer, Communication.Follower and Communication.Believer (see below).
- A set of enhancements to the standard RABs MessageSenderBehavior and MessageReceiverBehavior which are used for message exchange between UBIWARE agents (see Katasonov , 2008)
- Some modification in the behavior engine to enable implementation of the approach.

Agent communication in S-APL

IEEE FIPA developed a set of standard specifications for agent communication including ACL for message envelopes and SL for contents. While the standard position of ACL is unquestionable, the value of SL is less certain. Although meaning ”semantic language”, SL is not based on W3C’s RDF semantic data model. Rather, SL follows the traditional agent design approaches where the agents’ beliefs and thus also the atoms of their communications are nary predicates. However, N-ary predicates do not make the meaning of data as explicit as RDF triples do. Also, only the whole message can be linked to an ontology, as compared to the ability of RDF to link every individual resource to its own ontology, if needed.

In this section, we describe how we use S-APL as the content language in agent communications. Since one of the important communicative actions is querying for information, this role of S-APL overlaps with that of SPARQL. The problem with SPAQRL is that while being a language for querying RDF, it is not RDF itself. Also obviously, a content language for agent communication must support other types of communicative actions, for example, request for action. For these reasons we did not considered using SPARQL as such. Rather, when designing S-APL we included into it features analogous to most of the SPARQL’s ones (see Chapter 1).

The beliefs storage of an S-APL agent can be queried externally by other agents, of course subject to security and other policies. The core of a query is the same as if the agent itself would query its beliefs to check the premises of a rule, i.e. it can use all the constructs allowed for the left side of => (see Chapter 1). The core of the query has to be wrapped with sapl:I sapl:want { {sapl:You sapl:answer {..query..} }}. The use of ”sapl:I sapl:want” may look unnecessary. However, this allows distinguishing between sapl:I sapl:want {...} and e.g. :Boss sapl:want {...}, i.e. mediating a wish of another agent. Both cases may require exactly the same action to be taken, however, may affect differently on whether the agent will comply or not.

As the response, the agent has to send the matching part of its belief storage, or, if no match, the query itself wrapped with sapl:I sapl:doNotBelieve {...}. Below, we list two small S-APL programs that an agent has to load in order to be able to be queried this way. The first one, Listener.sapl, instructs the agent to continuously wait for incoming messages marked with ”SAPL” ontology. The additional rule of the program adds for every incoming request an additional existsWhile statement so that the request is removed after 5 seconds if no rule has taken it for processing.

```
/*Listener.sapl*/
{sapl:I sapl:do java:ubiware.shared.MessageReceiverBehavior}
sapl:configuredAs {p:matchOntology sapl:is "SAPL".
                   p:waitOnlyFirst sapl:is false}.
{ {{?requestID p:received *} sapl:ID ?id. sapl:Now sapl:is ?time.
  sapl:I sapl:doNotBelieve {?x sapl:existsWhile *. ?x sapl:hasMember ?id}
 } => {
   {?id sapl:is sapl:true} sapl:existsWhile
   {sapl:Now sapl:is ?newtime. ?newtime < ?time+5000}
 }
} sapl:is sapl:Rule
```

The second program, Informer.sapl, implements the processing of a query and sending back the response (we do not include here any access control or other checks, just the basic logic). As can be seen, the logic of moving from query to response is as simple as {?query sapl:is sapl:true} -> {?requestID :response {?query sapl:is sapl:true}}. The parameter p:cleanContent of MessageSenderBehavior set to true causes removing from the response residual wrappings like sapl:Optional, sapl:or, as well as filtering statements and sapl:hasMember/sapl:memberOf statements. Wrappings like sapl:All, sapl:OrderBy disappear automatically as soon as their objects are not variables anymore.

```
/*Informer.sapl*/
{ {?requestID p:received
        {p:sender sapl:is ?agent. p:conversationID sapl:is ?convID.
        p:content sapl:is
                {sapl:I sapl:want {sapl:You sapl:answer ?query}}}.
  sapl:I sapl:doNotBelieve {sapl:I :handle ?requestID}
 } => {
        sapl:I :handle ?requestID.
       {?query sapl:is sapl:true} ->
        {?requestID :response {?query sapl:is sapl:true}. ?requestID :result true}
        ; sapl:else {?requestID :response {sapl:I sapl:doNotBelieve ?query}.
                    ?requestID :result false}.
       {?requestID :response ?response. ?requestID :result ?result
       } => {
        {sapl:I sapl:do java:ubiware.shared.MessageSenderBehavior}
        sapl:configuredAs
                {p:receiver sapl:is ?agent. p:ontology sapl:is "SAPL".
                p:content sapl:is ?response. p:performative sapl:is inform.
```

```
                    p:cleanContent sapl:is ?result. p:addBeliefs sapl:is false.
                    p:conversationID sapl:is ?convID.
                    sapl:End sapl:remove {?requestID * *. sapl:I :handle ?requestID}}}
          }
    } sapl:is sapl:Rule
```

For example, let’s assume that there is an agent with Listener and Informer programs loaded and having the following beliefs in its storage:

```
:factory1 :hasSpace :room3, :room1, :room2.
:room1 :hasTemperature 25; :hasHumidity 80.
:room2 :hasTemperature 20; :hasHumidity 90.
{?room :hasTemperature 30} =>
{sapl:I ex:sendAlarm {:source sapl:is ?room}}
```

Then, the following queries (only the core is shown) will get the corresponding responses:

- (Q) :factory1 :hasSpace :room1
  (R) :factory1 :hasSpace :room1
- (Q) {{:factory1 :hasSpace ?room} sapl:All ?room} sapl:OrderBy ?room
  (R) :factory1 :hasSpace :room1. :factory1 :hasSpace :room2. :factory1 :hasSpace :room3
- (Q) :factory2 :hasSpace ?room
  (R) sapl:I sapl:doNotBelieve {:factory2 :hasSpace ?room}
- (Q) {:factory1 :hasSpace ?room. {?room :hasTemperature ?temp} sapl:is sapl:Optional} sapl:All ?room
  (R) :factory1 :hasSpace :room2. :room2 :hasTemperature 20. :factory1 :hasSpace :room3. :factory1 :hasSpace :room1. :room1 :hasTemperature 25
- (Q) {{{?r1 :hasTemperature ?temp. ?temp > 24} sapl:or {?r2 :hasHumidity ?hum. ?hum > 85}} sapl:All ?r1 } sapl:All ?r2
  (R) :room2 :hasHumidity 90. :room1 :hasTemperature 25
- (Q) ?left => ?right. ?left sapl:hasMember {:room1 :hasTemperature ?t}
  (R) {?room :hasTemperature 30} => {sapl:I ex:sendAlarm {:source sapl:is ?room}}

The last of the example queries above demonstrates an important feature of S-APL – ability of agents to exchange rules. Therefore, an agent can ask another agent how that will react if some situation occurs. Alternatively, an agent may query another agent if that knows a plan leading to achieving a goal. The above example also demonstrates a specific convention of S-APL – the data against which a query is evaluated can be implicitly universally quantified through use of variables. This is why the query for a specific room ”{:room1 :hasTemperature ?t} => {}” yields a universal rule ”{?room :hasTemperature 30} => {}”.

The next simple program below, Believer.sapl, instructs the agent to add to its beliefs everything that it receives in messages with the ”inform” performative. This program can, for example, be used by an employee agent which is committed to comply with any of its boss' statements. Some similar code is also needed for an agent that queried some information from an Informer (see above) to add the response to its beliefs. Note that substituting the line "?content sapl:is sapl:true" with something like "{?content sapl:is sapl:true} :accordingTo :John" will lead to another effect: instead of believing into the information provided, the agent will record it wrapped as "John thinks that ...".

```
/*Believer.sapl*/
{  {?requestID p:received {p:content sapl:is ?content.
                           p:performative sapl:is inform}
  } => {
          sapl:I sapl:remove {?requestID p:received *}.
          ?content sapl:is sapl:true
  }
} sapl:is sapl:Rule
```

Due to the fact that in S-APL there is no principal difference between data and program code, the content of the message to a Believer agent could include commitments, unconditional or conditional, and the Believer would comply and perform those actions. In this sense, the Believer with the code as above is more like a slave fully controlled by its master. In less dependent settings, agents can request other agents to perform some actions, corresponding either to a RAB or an abstract capability. The core of the request is the same as in a normal unconditional commitment, only with sapl:You in place of sapl:I, and in addition wrapped with sapl:I sapl:want: sapl:I sapl:want { {sapl:You sapl:do ...} sapl:configuredAs {...} }. The small program below, Follower.sapl, is to be used then by an agent to perform actions requested in this way (again, only the basic logic is included without access control or other checks).

```
/*Follower.sapl*/
{  {?requestID p:received {p:content sapl:is {
                                    sapl:I sapl:want {{sapl:You sapl:do ?action}
                                            sapl:configuredAs ?params}}}
  } => {
          sapl:I sapl:remove {?requestID p:received *}.
         {sapl:I sapl:do ?action} sapl:configuredAs ?params
  }
} sapl:is sapl:Rule
```

Use of S-APL as a communication language is quite natural because the communication over S-APL is easily organized with S-APL programming alone. An obvious additional benefit is the level of integration that in no-effort is then achieved between the communication and the agent behavior prescriptions. For example, an agent can query another agent for behavior rules – either to understand how that will react if a certain situation occurs or to learn itself how to achieve a certain goal. Similarly, agents can exchange commitments, plans, or basically belief structures of any complexity. To mention, in our current implementation the S-APL repositories (see Chapter 1) are managed by agents with Listener and Informer programs loaded and answering queries of the type ”?x sapl:belongs <some_sapl_program>”.

UBIWARE Deliverable D1.3:
Workpackage WP3:
Task T1.3_w3:

3 SURPAS – Smart Ubiquitous Resource Privacy and Security

The security is often seen as an add-on feature of a system. However, in many systems (and UBIWARE is one of them), the system remains nothing more but a research prototype, without a real potential of practical use, until an adequate security infrastructure is embedded into it. The main objective of this work package is the design of the SURPAS infrastructure for policy-based optimal collecting, composing, configuring and provisioning of security measures in multi-agent systems like UBIWARE. SURPAS follows the general UBIWARE vision – configuring and adding new functionality to the underlying industrial environment on-the-fly by changing high level declarative descriptions. Regarding security, this means that SURPAS will be able of smoothly including new, and reconfiguring existing, security mechanisms, for the optimal and secure state of the UBIWARE-based system, in response to the dynamically changing environment. The optimal state is always a tradeoff between security and other qualities like performance, functionality, usability, applicability and other.

During WP3's Year 1 (the *Access* phase), WP3 developed an infrastructure for semantic access control in UBIWARE, with S-APL used for specification of access control policies. Such policies may prohibit or allow an operation of a certain class to be performed by an agent of a certain class on some resource of a certain class.

As to UBIWARE 1.0 platform, the development contribution of WP3 consists of:

- Introducing into S-APL and the behavior engine the concept of a meta-rule (see Chapter 1).
- Standard S-APL models Security.SBACReasoner and Communication.Ontology.

3.1 Ontological modeling of operations

Semantics-based access control (SBAC) in UBIWARE steams from following observations:

- From the basic S-APL axiom, an operation (a commitment to executing a RAB, or a communicative action) is just a belief structure (agent's mental attitude). Therefore, it is possible to specify *an S-APL query* (as those used in the left side of rules in WP1 or as those sent from between agents wrapped as sapl:I sapl:want {sapl:You sapl:answer <query>} in WP2), which can be matched against G to find if the agent has such a mental attitude.
- Such a query, when universally quantified by using variables, presents a pattern for matching against *a class* of such mental attitudes and, therefore, can be used as the definition of this class.
- It is common to have an inheritance (class-subclass) hierarchy of operations. In terms of S-APL queries, the definition of a subclass usually only introduces *some additional restriction on the variables* used in the definition of the super-class.

In UBIWARE, using SBAC assumes that the inheritance hierarchy of operations is modelled in a following fashion:

- The top of the hierarchy is defined using <class> sapl:is <query>. Normally, <query> has only one (compound) statement which identifies the mental attitude in question. The top-hierarchy class is not supposed to be used in policy definitions, only its subclasses.
- The subclasses are defined using <subclass> rdfs:subClassOf <class>; owl:Restriction <extension to query>. Starting from the second from the top class, the variables ?subject and ?object must be included into the query, so that after evaluating the query it will be known who is the subject (actor) and who is the object of the operation.

Consider the following example:

```
:ActionCommitment sapl:is {
            {sapl:I sapl:do ?behavior} sapl:configuredAs ?parameters
}.
:Action     rdfs:subClassOf :ActionCommitment;
            owl:Restriction {sapl:I owl:sameAs ?subject. sapl:I owl:sameAs ?object}.
:Print      rdfs:subClassOf :Action;
            owl:Restriction {?behavior = :Print. ?parameters sapl:hasMember {p:print
          sapl:is ?print}}.
:Swear      rdfs:subClassOf :Print;
            owl:Restriction {:BadWord sapl:is ?word. sapl:true = "contains(?print,?word)"}.
```

Here, we define some simple ontology the top-class of which is :ActionCommitment and the first subclass that can be used in SBAC statements is :Action. :Action adds to the basic query of :ActionCommitment two additional statements which are supposed to initialize both ?subject and ?object variables to the URI of the agent (it is considered to be both an actor and the object in this case).

Then, a specific action :Print is defined by putting a restriction on the variable ?behavior so that it has to be equal to ":Print". The definition of :Print also bring forward the text that is to be printed and puts it into an additional variable ?print. Finally, even more specific action :Swear is describing that :Swear is :Print where ?print contains one of the words that are known to be bad. Note that this requires an additional query statement ":BadWord sapl:is ?word" and assumes that the agent may have some beliefs of this form. Obviously, the practical idea behind this example is that we may want to prohibit swearing, so that the agent will never be able to print out anything that contains any bad words (see next subsection).

The next example is the content of Communication.Ontology model, which defines the hierarchy of communicative actions between agents (here, the prefix com: is defined as <http://www.ubiware.jyu.fi/communication#>):

```
/*Classes of communicative actions*/
com:Message sapl:is {
  ?messageID p:received { p:sender sapl:is ?subject. p:performative sapl:is ?performative.
                          p:content sapl:is ?content. p:ontology sapl:is ?ontology }
}.
com:SpeechAct   rdfs:subClassOf com:Message;
                owl:Restriction {sapl:I owl:sameAs ?object}.
com:ProactiveSpeechAct rdfs:subClassOf com:SpeechAct;
                       owl:Restriction { ?content sapl:hasMember {sapl:I sapl:want {
                                               {sapl:You ?actType ?actContent}
                                                       sapl:or
                                               {{sapl:You ?actType ?actContent} * *}
                                         }
                       }}.
com:Query       rdfs:subClassOf com:ProactiveSpeechAct;
                owl:Restriction {?actType = sapl:answer}.
com:Order       rdfs:subClassOf com:ProactiveSpeechAct;
                owl:Restriction {?actType = sapl:do}.
com:Persuade    rdfs:subClassOf com:ProactiveSpeechAct;
                owl:Restriction {?actType = sapl:add}.
```

Note that ?subject of a communicative actions is one who has sent the message, while ?object is one who received it, i.e. the agent in question. The class com:Query corresponds to the communicative actions that an Informer (see Chapter 2) handles. The class com:Order corresponds to the communicative actions that a Follower handles, while the class com:Persuade to the communicative actions that a Believer handles. Obviously, the practical idea here is that we can now define SBAC policies that will restrict what agents answer queries or follow orders of what other agents - without the need to modify the models Informer, Follower or Believer, so they can remain as simple as described in Chapter 2.

3.2 Specifying access control policies

SBAC policies in UBIWARE are specified using simple SBAC statements (here, prefix sbac: is defined as <http://www.ubiware.jyu.fi/sbac#>):

- {<class of subjects> <class of operations> <class of objects>} sapl:is sbac:Prohibition.
- {<class of subjects> <class of operations> <class of objects>} sapl:is sbac:Privelege.

The effect is then that any mental attitude is removed if there is a Prohibition that covers it while there is no Privelege that covers it. For the example of prohibiting swearing, one could define the domain ontology and the policy as follows:

```
sapl:I owl:sameAs fg:Bill.
fg:Bill rdf:type fg:Boss.
fg:John rdf:type fg:Employee.
fg:Boss rdf:subClassOf fg:Employee.

{fg:Employee :Swear fg:Employee} sapl:is sbac:Prohibition.
```

Such policy implies no employee is allowed to swear. It is also possible to use * in place of the subject and the object leading to that the policy will apply to any resource.

For the example about communicative actions, one could define the domain ontology and the policy as follows:

```
sapl:I owl:sameAs fg:John.
fg:John rdf:type fg:Employee.
fg:Bill rdf:type fg:Boss.
fg:Boss rdfs:subClassOf fg:Employee.
bill owl:sameAs fg:Bill.

{* com:ProactiveSpeechAct *} sapl:is sbac:Prohibition.
{fg:Boss com:Query *} sapl:is sbac:Privelege.
```

Such a policy defines that the agent in question must not comply with any proactive speech act, with the only exception of a com:Query originated from an agent belonging to the class fg:Boss.

The SBAC run-time operation is implemented then by one rule and one meta-rule in the standard Security.SBACReasoner S-APL model (has to be loaded by the agent). SBACReasoner must be used together with RDFSReasoner so that the class hierarchy of resources will be inferred. So, in the first example above, RDFSReasoner is needed to infer that the agent is question is an Employee, because Boss is a subclass of Employee.

```
{
//Make the owl:Restriction of actions full by adding the owl:Restriction of their super-classes
  {
        {{
              ?X owl:Restriction ?own. ?X rdfs:subClassOf ?C.
              ?C owl:Restriction ?super. ?super sapl:hasMember ?id.
              sapl:I sapl:doNotBelieve {?own sapl:hasMember ?id}
        } sapl:All ?X } sapl:All ?C
  } => {
        ?own sapl:hasMember ?super
  }
} sapl:is sapl:Rule.


{
//Remove a mental attitude if there is a Prohibition that covers it while there is no Privelege that
covers it
  {
    {
     {?no_subject ?no_action ?no_object} sapl:is sbac:Prohibition.
        {?no_action sapl:is ?base. { ?base sapl:is sapl:true } sapl:ID ?id}
              sapl:or {?no_action rdfs:subClassOf ?act_root.
                    ?act_root sapl:is ?act_base.
                    { ?act_base sapl:is sapl:true } sapl:ID ?id.
                    ?no_action owl:Restriction ?act_restriction.
                    ?act_restriction sapl:is sapl:true}.
              {?subject = ?no_subject} sapl:or {{?subject owl:sameAs ?no_subject}
                    sapl:or {?subject rdf:type ?no_subject}}.
              {?object = ?no_object} sapl:or {{?object owl:sameAs ?no_object}
                    sapl:or {?object rdf:type ?no_object}}.
              sapl:I sapl:doNotBelieve {
                    {?yes_subject ?yes_action ?yes_object} sapl:is sbac:Privelege.
                    {?subject = ?yes_subject}
                          sapl:or {{?subject owl:sameAs ?yes_subject}
                           sapl:or {?subject rdf:type ?yes_subject}}.
                    {?object = ?yes_object} sapl:or {{?object owl:sameAs ?yes_object}
```

```
                                      sapl:or {?object rdf:type ?yes_object}}.
                              {?no_action = ?yes_action}
                                     sapl:or {{?no_action rdfs:subClassOf ?yes_action}
                                     sapl:or {          ?yes_action rdfs:subClassOf ?no_action.
                                             ?yes_action owl:Restriction ?restriction2.
                                             ?restriction2 sapl:is sapl:true}}
                     }
             } sapl:All ?id
     } => {
             sapl:I sapl:erase ?id.
             {sapl:I sapl:do :Print} sapl:configuredAs {
                     p:print sapl:is "Security has blocked a case of ?no_action"}
     }
   } sapl:is sapl:MetaRule.
```

UBIWARE Deliverable D1.3:
Workpackage WP4:
Task T1.3_w4:

4 Self-Management, Configurability and Integration

UBIWARE aims to be a platform that can be applied in different application areas. This implies that the elements of the platform have to be adjustable, could be tuned or configured allowing the platform to run different business scenarios in different business environments. Such flexibility calls for existence of a sophisticated configuration layer of the platform. All building blocks of the UBIWARE platform, i.e. software agents, agent behaviors, resource adapters, etc, become subject to configuration. On the other hand, a flexible system should have a long lifespan. Hence, the platform should allow extensions, component replacements, and component adjustments during the operation time. This work package aims at introducing configurability as a pervasive characteristic of UBIWARE and developing the technology which will systemize and formalize this feature of the platform.

During WP4's Year 1 (the *Component* phase), we developed solutions for configurability of basic UBIWARE elements such as resource adapters and Reusable Atomic Behaviors, with S-APL used as the tool for both describing the configuration and for applying it.

As to UBIWARE 1.0 platform, the development contribution of WP4 consists of:

- A set of RABs - TextTableReaderBehavior, SQLReaderBehavior, ExcelReaderBehavior, XmlReaderBehavior, XmlWriterBehavior - as a part of the general approach towards resource adaptation (see Section 4.1)
- Standard S-APL models Configurability.RABConfigurator (see Section 4.2)

4.1 Resource adaptation

Roughly speaking, resource adaptation involves accessing data from an industrial resource (either physical through sensors or a digital like a database) in its own proprietary format and transforming this data into an ontological S-APL presentation. The UBIWARE's general approach towards configurable resource adaptation is depicted in Figure 4.

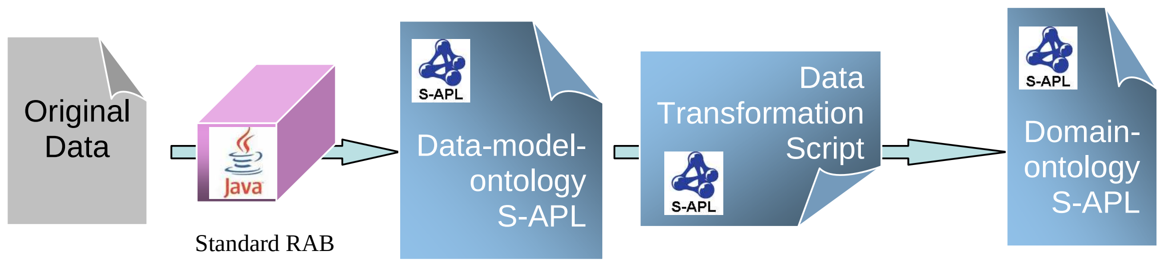


Figure 4. UBIWARE approach to resource adaptation

The resource adaptation is performed in two phases:

- First, the original data from the resource is transformed into an S-APL representation based on *the ontology of the data model* of the original data. This transformation is performed by a Reusable Atomic Behavior (RAB).
- Second, the data from the data-model-ontology S-APL is transformed into the final domain-ontology S-APL using S-APL own means, i.e. rules.

At present, most of the data in the industrial applications is encoded using either *table data model* (relational databases, comma-separated files, etc) or *XML tree data model*. Ubiware 1.0 provides a set of standard transformation RABs, including TextTableReaderBehavior (file), SQLReaderBehavior (relational database), ExcelReaderBehavior (MS Excel table), and XmlReaderBehavior (XML file). These 4 RABs already cover a large share of industrial adaptation cases.

TextTableReaderBehavior, SQLReaderBehavior and ExcelReaderBehavior are different in a sense that they read different media, but all 3 follow the *table data model* and produce similar output. The ontology of tables consists of such concepts as ***table***, ***row*** and ***column***. All 3 RABs produce the following output structure that is based on this ontology:

```
<input name> :table {
        <row number> :row {
                <column name or number> :column <value>.
                 …
        }.
         …
}
```

XmlReaderBehavior follow the *XML tree data model*. The ontology of XML trees consists of such concepts as ***tree***, ***branch***, ***leaf***, ***tag***, and ***attribute***. XmlReaderBehavior produces the following output structure that is based on this ontology:

```
<input file name> :tree {
        {<child number> :tag <tag name>} :branch {
                {<child number> :attribute <attribute name>} :leaf <attribute value>.
                <child number> :leaf <literal value>.
```

```
                                {<child number> :tag <tag name>} :leaf :empty.
                                {<child number> :tag < tag name>} :branch {
                                        …
                                }
                        }
        }
```

As can be seen, XML tree is described as a hierarchical structure of branches and 3 types of leafs: XML attributes, literal nodes, and empty elements (e.g. <br/>). Child numbering is inside one given branch.

Note UBIWARE 1.0 also includes XmlWriterBehavior which performs the transformation in the opposite direction. Given a S-APL structure as above, it produces corresponding XML string. Obviously, the whole resource adaptation process in Figure 3 may be performed in the opposite direction - we transform some domain-ontology-based S-APL into XML-tree-based S-APL and then apply XmlWriterBehavior to get the final XML document. We use this approach in the industrial cases to produce HTML interfaces.

The transformation between data-model ontology S-APL and domain ontology S-APL is an easy and straightforward task. Below is an example (based on Fingrid case, but not the exact code):

```
        {
                * :table {?rowId :row {   ALARMTEXT :column ?text. TIME :column ?time
                                           AORMASKGRP2 :column ?area.
                                      }
                }.
                ?year sapl:expression "substring(?time,6,10)".
        }  => {
                {
                        fg:EventHistory fg:hasEvents {{
                                fg:alarm :is ?text. fg:time :is ?time.
                                fg:year :is ?year. fg:area :is ?area.
                        } fg:id ?rowId}.
                } sapl:All ?rowId .
        }
```

Since the resource adaptation is performed in two phases, each of the phases can be subject to (re-)configuration. The S-APL transformation script is configurable by default since modifiability is an inherent S-APL feature. The behavior of the transformation RAB can be adjusted through the mechanism described in the next subsection.

4.2 Configurability of atomic behaviors

In UBIWARE, the meta-rule mechanism, which has been initially introduced for realizing security access control policies, is also used for configurability of Reusable Atomic Behaviors (RABs) (Actually the approach works for any unconditional commitments - also to complex actions not having “java:” or “:” namespaces, but we will speak here about RABs).

When using this approach, the configuration settings of RABs are specified using the following structure (here, prefix cfg: is defined as <http://www.ubiware.jyu.fi/configurability#>):

```
<RAB> cfg:configuredAs {
        <parameter to override or add> sapl:is <value>.
        <parameter to add if not defined> cfg:isDefault <value>.
}
```

The idea is that any call of this <RAB> will be then modified - meaning that list of parameters specified in the call itself will be extended / overridden. Any parameter specified with predicate sapl:is will be overridden (or added if was not given), while any predicate specified in with cfg:isDefault will be added only if it was not given in the call itself. Note that the RAB configuration settings are specified using cfg:configuredAs, not sapl:configuredAs (same local name, different namespaces).

Consider the following example of a RAB configuration:

```
java:ubiware.shared.TextTableReaderBehavior cfg:configuredAs {
        p:selectColumns cfg:isDefault "1 3".
        p:encoding sapl:is "UTF-16".
}.
```

Then, if the call to this RAB was like:

```
{sapl:I sapl:do java:ubiware.shared.TextTableReaderBehavior} sapl:configuredAs {
        p:input sapl:is “test.csv”.
        p:columnSeparator sapl:is ",|\\s+".
        p:encoding sapl:is "default".
}
```

The actual call will be performed as:

```
{sapl:I sapl:do java:ubiware.shared.TextTableReaderBehavior} sapl:configuredAs {
        p:input sapl:is “test.csv”.
        p:columnSeparator sapl:is ",|\\s+".
        p:encoding sapl:is "UTF-16".
        p:selectColumns sapl:is "1 3".
}
```

In result, the assumed encoding of the CSV document being loaded with be modified to Unicode, as well as the columns that are included into the result will be reduced to only first and third (default value is all). Note that in this example, if the RAB call was specifying p:selectColumns itself, the value would not be changed.

The run-time operation of this approach is then implemented then single meta-rule in the standard Configurability.RABConfigurator S-APL model (has to be loaded by the agent):

```
{
  {
          {sapl:I sapl:do ?action} sapl:configuredAs ?config.
          ?action cfg:configuredAs ?default.
          {?default sapl:hasMember {?param sapl:is ?value}.
                  sapl:I sapl:doNotBelieve {?config sapl:hasMember {?param sapl:is ?value}}}
          sapl:or {?default sapl:hasMember {?param cfg:isDefault ?value}.
                  sapl:I sapl:doNotBelieve {?config sapl:hasMember {?param sapl:is *}}}
  } => {
          {
                  sapl:I sapl:remove {?config sapl:hasMember {?param sapl:is *}}.
                  {
                          sapl:I sapl:add {?config sapl:hasMember {?param sapl:is ?value}}
                  } sapl:All ?value
          } sapl:All ?param
  }
} sapl:is sapl:MetaRule.
```

UBIWARE Deliverable D1.3:
Workpackage WP5:
Task T1.3_w5:

5 Smart Interfaces: Context-aware GUI for Integrated Data (4i technology)

This workpackage studies dynamic context-aware Agent-to-Human interaction in UBIWARE, and elaborates on a technology which we refer to as 4i (FOR EYE technology). From the UBIWARE point of view, a human interface is just a special case of a resource adapter. We believe, however, that it is unreasonable to embed all the data acquisition, filtering and visualization logic into such an adapter. Instead, external services and application should be effectively utilized. Therefore, the intelligence of a smart interface will be a result of collaboration of multiple agents: the human's agent, the agents representing resources of interest (those to be monitored or/and controlled), and the agents of various visualization services. This approach makes human interfaces different from other resource adapters and indicates a need for devoted research. 4i technology will enable creation of such smart human interfaces through flexible collaboration of an Intelligent GUI Shell, various visualization modules, which we refer to as MetaProvider-services, and the resources of interest.

During the Project Year 1, the work in this WP will develop the general principles of the 4i approach and will aim at developing an appropriate GUI Shell.
The WP development task *Task T1.2_w5* for the Year 1 is concentrated on development of a simple MetaProvider for context dependent resource visualization, development of initial GUI-Shell able to communicate with the MetaProvider.

5.1 Background

Now, when human becomes very dynamic and proactive resource of a large integration environment with a huge amount of different heterogeneous data, it is quite necessary to provide a technology and tools for easy and handy human information access and manipulation. Semantically enhanced context-dependent multidimensional resource visualization (Khriyenko, 2007a, Khriyenko, 2007b) provides an opportunity to create intelligent visual interface that presents relevant information in more suitable and personalized for user form. Context-awareness and intelligence of such interface brings a new feature that gives a possibility for user to get not just raw data, but required information based on a specified context.

Now it has become evident that we cannot separate visual aspects of both data representation and graphical interface from interaction mechanisms that help a user to browse and query a data set through its visual representation. Following GUN-Resource centric approach (Kaykova et al., 2005), let us consider user interfaces for context-based resource access and contextually related information retrieving. The challenging task is to create a visual interface that provides integrated information from variety of information providers in context-dependent way. Following new technological trends, it is time to start a new stage in user visual interface development – a stage of semantic-based context-dependent multidimensional resource visualization. Presented 4i technology is a step to achieve this goal (Khriyenko, 2007a).

4i (FOR EYE) is an ensemble of GUN Resource Platform Intelligent GUI Shell (smart middleware for context dependent use and combination of a variety of different MetaProviders depending on user needs) and MetaProviders, visualization modules/platforms that provide context-dependent filtered representation of resource data and integration on two levels (information/data integration of the resources to be visualized and integration of resource representation views with a handy resource browsing) (see Figure 5). Context-awareness and intelligence of such interface brings a new feature that gives a possibility for user to get not just raw data, but required integrated information based on a specified context. GUI Shell allows user dynamic switching between MetaProviders for more suitable information representation depending on a context or resource nature. From other side, MetaProvider plays fore main roles:

- Context-aware resource visualization module that presents information regarding to specified context in more suitable and personalized for user form;
- Interface for integrated information visualization with intelligent context-aware filtering mechanism to present only relevant information, avoiding a glut of unnecessary information;
- Visual Resource Platform that allows resource registration, search, access and modification of needed information/data in a space of registered resources;
- Mediator that facilitates resource to resource (R2R) communication.

Such switching and filtering process is a kind of visual semantic browsing based on semantic description of the context and resource properties.

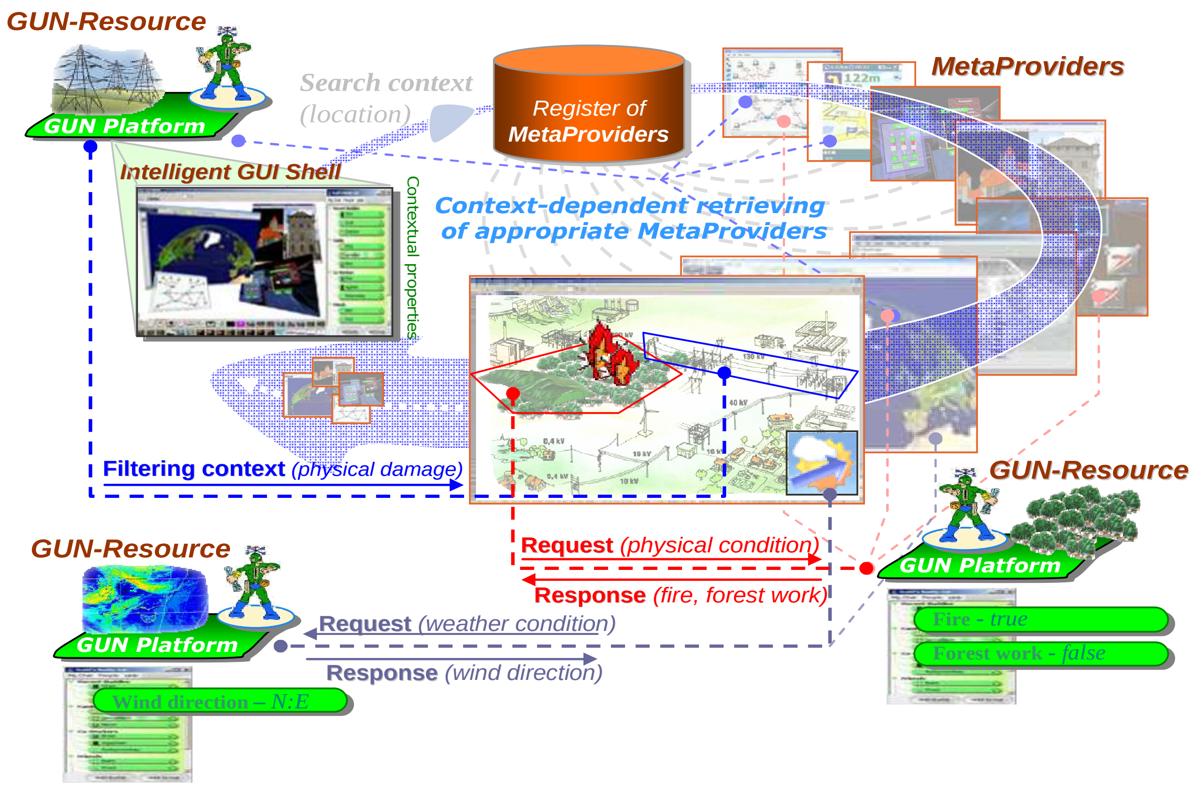

GUN-Resource
GUN Platform
Search context (location)
Register of MetaProviders
MetaProviders
Intelligent GUI Shell
Contextual properties
Context-dependent retrieving of appropriate MetaProviders
Filtering context (physical damage)
GUN-Resource
GUN Platform
Request (physical condition)
Response (fire, forest work)
GUN-Resource
GUN Platform
Request (weather condition)
Response (wind direction)
Wind direction – N:E
Fire - true
Forest work - false

Figure 5 - Intelligent Interface for Integrated Information (4i technology) (Khriyenko, 2007a).

Now, when unlimited interoperability and collaboration demand data and information sharing, we need more open semantic-based applications that are able to interoperate and collaborate with each other. Ability of the system to perform semantically enhanced resource search/browsing based on resource semantic description brings a valuable benefit for today Web and for the Web of the future with unlimited amount of resources. Proposed resource visualization approach can find a place and can be utilized in various visual systems and especially in next-generation human-centric open environments for resource collaboration with enhanced semantic and context-based visual resource browsing. It can be considered as a new valuable extension of text-based Semantic MediaWiki to Context-based Visual Semantic MediaWiki. This is a good basis for the different business, production, maintenance, healthcare, social process models creation and multimedia content management.

5.2 SmartInterface

As mentioned above, SmartInterface is presented by GUI-Shell (central browser of the system) and remote distributed visualization modules - MetaProviders.

5.2.1 GUI-Shell

GUI-Shell is presented by Html-page and remote server part that plays role of search engine and performs all necessary complex calculations. Main Html-page contains five areas (see Figure 6):

- *Resource Search Area*: Based on keywords, which describe a resource type (class of resources) and a content of a resource, user gets a list of corresponding resources;
- *Resource Description Area*: Here, properties of current (currently selected) resource are displayed to the user. In current version of the prototype, user may just observe resource properties and cannot make any changes to them;
- *Visualization Context Area*: Here system provides a list of visualization contexts for currently visualized resources;
- *MetaProviders Area*: Depending on chosen resource visualization context, a list of appropriate visualization modules (MetaProviders) is presented for user in this area;
- *Resource Visualization Area*: This is an area, where visualization page of chosen MetaProvider is loading and performs.

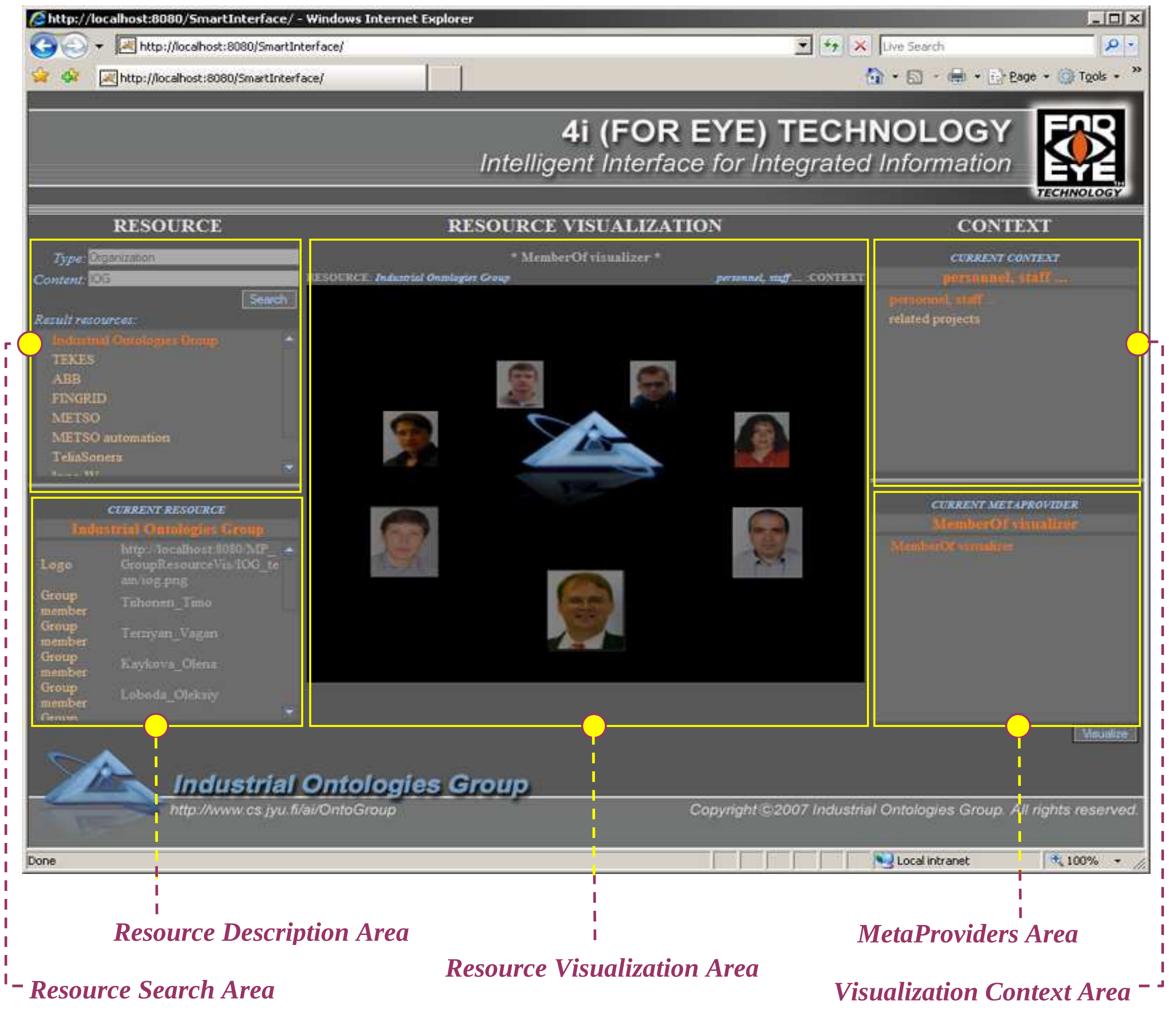


Figure 6 – GUI-Shell.

Another important part of the GUI-Shell is server part of it. As a search engine, it performs several tasks:

- based on keyword of resource type and resource content descriptions, returns a list of matched resource back to Html-page;
- parses Ontology and returns a list of correspondent visualization contexts relevant for class of selected resource;
- returns a list of appropriate MetaProviders based on specified resource (resource class) and visualization context.

Doing the first year prototype, we decided to use XML files for storing resource's, visualization context's, MetaProvider's descriptions and for information exchange between the parts of the system. Below you can see the XML structures we have used:

Resource description:

```
<resource>
    <resId>...</resId>                           resource id
    <resClass>...</resClass>                     class of the resource
    <name>...</name>                             resource name
    <resTypeDes>
       <rtdItem>...</rtdItem>                 }  keywords for resource type (class) description…
       ...
    </resTypeDes>
    <resContDes>
       <rcdItem>...</rcdItem>                 }  keywords for resource content description…
       ...
    </resContDes>
    <properties>
       <property>
          <prop_id>...</prop_id>
          <prop_name>...</prop_name>                    }  other resource properties…
          <prop_value>...</prop_value>
       </property>
       ...
    </properties>
</resource>
```

Resource visualization context description:

```
<context>
    <contId>...</contId>                         context id
    <name>...</name>                             name of the context
    <forClasses>
       <class>...</class>                     }  set of resource classes for which current visualization
       ...                                       context is applicable
    </forClasses>
</context>
```

MetaProvider description:

```
<mp>
   <mpId>...</mpId>                            MetaProvider id
   <name>...</name>                            name of the MetaProvider
   <link>...</link>                            link to the MetaProvider
   <rescont>
      <resClass>...</resClass>                 class of a resource to be visualized…
      <cont>
            <contId>...</contId>               visualization context for the resource…
            <_in>...</_in>                   } set of required properties of the resource…
            ...
            <_ins refProp="...">             } set of required properties of context related resources,
                  <_in>...</_in>               which have relation to the subject resource via “refProp”
                  ...                          property…
            </_ins>
            <_out>...</_out>                   Out property – in our case it is” resId” (id of a resource)
      </cont>
      ...
   </rescont>
   ...
</mp>
```

Present structures are built with a purpose to be easily converted to RDF representation without any logic to be lost.

Following figure (see Figure 7) presents a full interaction model of the system. On the first step, user performs search for initial resource to start through-resource browsing process. Based on keywords that describe type and content of a resource, GUI-Shell searches resources via accessible databases or with a help of MetaProviders that have own data storages. Then, user can choose proper resource from a list of found resources. On the second step, GUI-Shell brows ontology and gets a list of relevant resource visualization contexts. Based on specified context for current resource, GUI-Shell searches for appropriate visualization modules (MetaProviders). There are can be number of them (with different implementations, sets of features, etc.) When certain MetaProvider has been chosen, it sets up communication with GUI-Shell and, based on specified resource and context, decides what the relevant resources (in this context) are and what data needed for proper visualization is. On the fourth step, MetaProvider starts to retrieve necessary information from own databases, through communication with GUI-Shell or other MetaProviders. When all (at least necessary part) the data is collected, MetaProvider performs visualization in *Resource Visualization Area* of the GUI-Shell. Fifth interaction step between GUI-Shell and MetaProviders is reserved for feedback when MetaProvider informs the GUI-Shell about selected resource. Thus, whole through-resource browsing cycle ends and starts here.

Current implementation is not based on UBIWARE Agent Platform. For the first year prototype of 4I Technology we decided to check the possibility of newly elaborated idea implementation and do not base development on underdevelopment prototype of UBIWARE Platform. For the next year we are going to implement whole system based on the first year version of UBIWARE Platform.

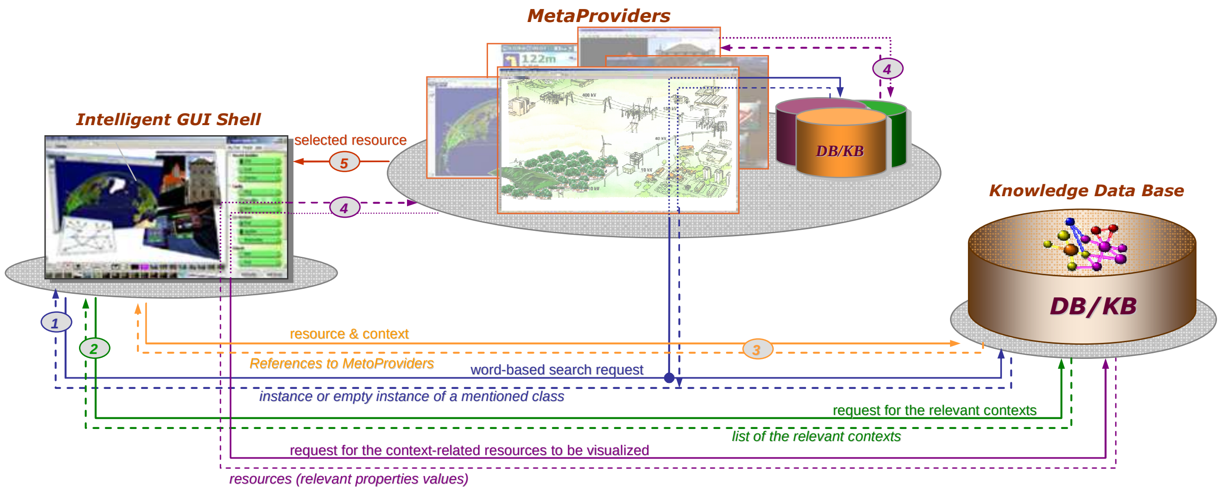

MetaProviders
Intelligent GUI Shell
Knowledge Data Base
DB/KB
DB/KB
selected resource
5
4
4
1
2
3
resource & context
References to MetoProviders
word-based search request
instance or empty instance of a mentioned class
request for the relevant contexts
list of the relevant contexts
request for the context-related resources to be visualized
resources (relevant properties values)

Figure 7 – SmartInterface interaction model.

Communication between GUI-Shell and MetaProvider is planed to be organized based on Agent communication. Further, in Agent based implemented version, MetaProviders themselves will decide (reason) what is the necessary data and will perform retrieving of it with a help of the GUI-Shell or via own data retrieving channels. But, in current version, to avoid non-reusable implementation of communication between GUI-Shell and visualization modules, we put a description of input data (necessary resources' properties) for MetaProviders to their descriptions. Thus, GUI-Shell collects required input data and sends it through the request on MetaProvider loading.

To organize dynamic search on-the-fly, we applied AJAX technology[1] during the GUI-Shell development. AJAX (Asynchronous JavaScript and XML), is a group of inter-related web development techniques used for creating interactive web applications. A primary characteristic is the increased responsiveness and interactivity of web pages achieved by exchanging small amounts of data with the server "behind the scenes" so that entire web pages do not have to be reloaded each time there is a need to fetch data from the server. Thus, this technology allows us to build dynamic SmartInterface and increase the interactivity, speed, functionality and usability of it.

5.2.2 MetaProvider – "Member-Of visualizer"

Concerning development of the simple MetaProvider ("Member-Of visualizer") that visualizes resource in contexts were "member-of" property plays main role, with the purpose to visualize it in more natural for human way, we decided to utilize X3D [2] related technologies. X3D is a royalty-free open standards file format and run-time architecture to represent and communicate 3D scenes and objects using XML. It is an ISO ratified standard that provides a system for the storage, retrieval and playback of real time graphics content

[1] AJAX technology - http://www.w3schools.com/AJAX/default.asp
[2] X3D technology - http://www.web3d.org/

embedded in applications, all within an open architecture to support a wide array of domains and user scenarios. The development of real-time communication of 3D data across all applications and network applications has evolved from its beginnings as the Virtual Reality Modeling Language (VRML) to the considerably more mature and refined X3D standard. To provide communication with X3D scene, we utilized SAI (Scene Access Interface) that allows a programmer to change or build X3D worlds, and AJAX technology.

In current implementation the only strict requirement for MetaProvider development is "callFunction" input parameter. This parameter contains the name of callback function of the GUI-Shell which MetaProvider should call and send the ID of the current (selected) resource as a parameter.

"Member-Of visualizer" is a simple MetaProvider that visualizes a resource and related to it resources via "member-of" upper-property. For example, members of an organization (group) or projects leaded by this organization, project consortium or management board members, etc. MetaProvider is developed based on X3D technology to present resource in more demonstrative for human form. As the one of the best representation forms, the image based representation has been used to visualize a resource. Person is visualized via his/her photo, organization – via its logo. That is why, required data for such visualizer is:

- resource ID, image (photo, logo, etc.), image width and height - of subject resource;
- same data set - of other related resources.

"Member-Of visualizer" is a JSP page that gets request from GUI-Shell and, based on incoming parameters, utilizes server part of the MetaProvider to create an appropriate X3D file to be loaded in imbedded X3D plug-in. Visualizer applies the same visualization approach for all the resources (see Figure 8).

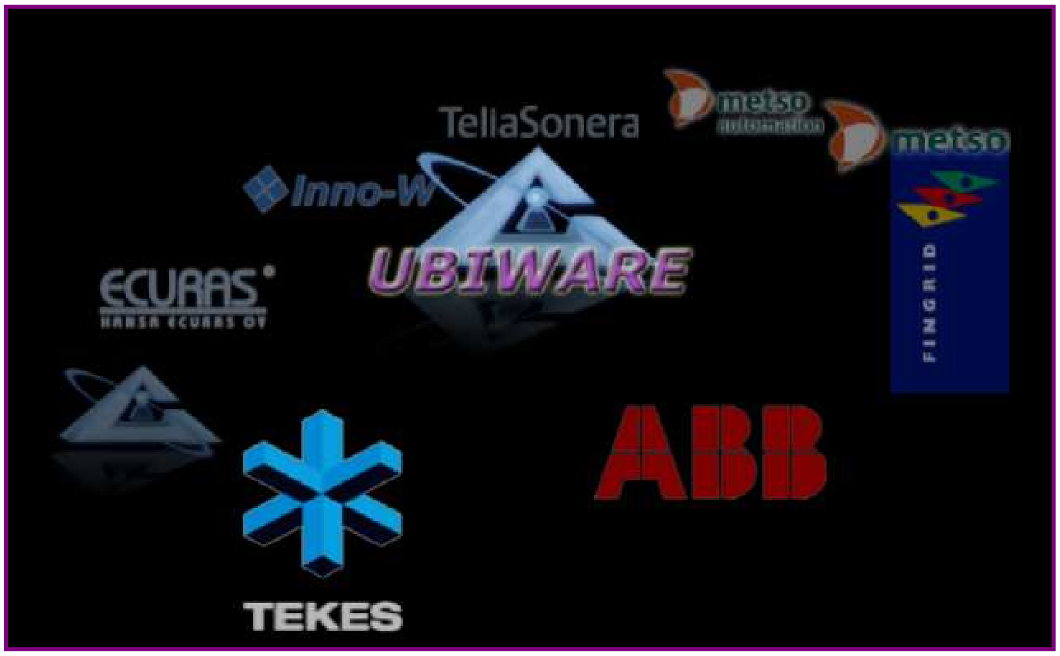


Figure 8 – "Member-Of visualizer" X3D Scene *(project consortium).*

This is a 3D scene with an image (photo or logo of subject resource) in the center of it. All the other related resources (their images) are located around of the main one organizing a circle/disk. Interface allows rotation of the circle/disk to get the best view point. Pointing on an image causes highlighting of it and makes correspondent resource selected. The click on any, related to the subject one, resource results its visualization. If the subject resource is clicked, the circle/disk of related resources performs slow rotation on 360 degree to present all the resources to the user. To be able communicate with GUI-Shell, MetaProvider gets a callback function in a request from it. Each time, when user selects a resource on the MetaProvider, this function is called and GUI-Shell gets a correspondent resource ID to show the resource properties and continue resource visualization process.

5.2.3 Useful features

With a purpose to make interaction with a SmartInterface more user-friendly and do not demand a lot of manipulation from user, we defined default visualization contexts for resource classes and MetaProviders for certain visualization context. Further, we plane to add personalization to the prototype and user will be able to choose favorite visualization module (MetaProvider) for certain resource in certain visualization context. This personalized data will be stored in own user-profile and will be applied with log-in to the system.

Also, we are thinking about a smart and intelligent technique for automatic dynamic selection of a visualization context. The logic will be based on a history of visualization contexts and resources that user has browsed/visualized previously. This context ranking technique will allow us to sort a list of visualization contexts in more appropriate order for user and give him/her a hint for next logical step in though resource browsing process. Thus, it can become a smart search system that leads the user in proper direction/way.

5.3 Future work

WP5’s Year 2 will elaborate on probably the most important part of 4i vision, which can be called “context provision”. Especially when considering a human, presenting information on a resource of interest alone is not sufficient - information on some “neighboring” objects should be included as well, which form the context of the resource. For example, a resource can be presented on a map thus shown in the context of objects which are spatial (geographic) neighbors of it. What is important is that in different decision-making situations, different contexts are relevant: depending on the situation the relevant neighborhood function may be e.g. physical spatial, data-flow connectivity, what-affects-what, similar-type, etc. The ability to determine what type of context in right one for the situation and collecting the information that forms the context of that type for a specific resource is central in 4i vision.

During WP5’s Year 2 (the *Context-awareness* phase), therefore, the following research questions are to be answered:

- What should be the architecture of MetaProvider-services so that they will be able to effectively retrieve, integrate and deliver the context information both to for presenting to humans (in a visual form) and for agents' processing (semantic data)?
- What should be the architecture of the Intelligent GUI Shell, so that it will allow situation-dependent selection of MetaProviders (i.e. different types of context) and cross-MetaProvider browsing and integration?

The WP tasks for the Year 2 are the following:

Task T2.1_w5 (research): Answer to the questions above. Design of mechanisms for context-aware (visual) representations creation and combining.

Task T2.2_w5 (development): Incorporating the research findings to the UBIWARE prototype.

Bibliography

Bellifemine, F. L., Caire, G., and Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE*. Wiley.

Collier, R., Ross, R. and O'Hare, G.M.P. (2005). Realising reusable agent behaviours with ALPHA. In *Proc. 3rd Conference on Multi-Agent System Technologies (MATES-05)*, LNCS vol. 3550, pp. 210–215.

Helsinger, A., Thome, M., and Wright, T. (2004). Cougaar: a scalable, distributed multi-agent architecture. In Proc. IEEE International Conference on Systems, Man and Cybernetics. Volume 2, pp. 1910–1917.

Katasonov, A. (2008) UBIWARE Platform and Semantic Agent Programming Language (S-APL): Developer's guide, Online: http://users.jyu.fi/~akataso/SAPLguide.pdf.

Kaykova, O., Khriyenko, O., Kovtun, D., Naumenko, A., Terziyan, V., Zharko, A., (2005). General Adaption Framework: Enabling Interoperability for Industrial Web Resources, In: *International Journal on Semantic Web and Information Systems*, Idea Group, ISSN: 1552-6283, Vol. 1, No. 3, July-September 2005, pp.31-63.

Khriyenko, O., (2007a). "4I (FOR EYE) Technology: Intelligent Interface for Integrated Information", In: *Proceedings of the 9th International Conference on Enterprise Information Systems (ICEIS-2007)*, Funchal, Madeira – Portugal, 12-16 June 2007.

Khriyenko O., (2007b). "Context-sensitive Multidimensional Resource Visualization", In: *Proceedings of the 7th IASTED International Conference on Visualization, Imaging, and Image Processing (VIIP 2007)*, Palma de Mallorca, Spain, 29-31 August 2007.

Appendix A: UBIWARE Publications List

Katasonov A., Terziyan, V., Agent Communications with S-APL as the Content Language, In: Proceedings of the International Workshop on Middleware for the Semantic Web in conjunction with the Second IEEE International Conference on Semantic Computing (ICSC-2008), August 4-7, 2008, Santa Clara, CA, USA, IEEE CS Press, 6 pp. (submitted 22 April 2008).

Katasonov A., Terziyan, V., Semantic Approach to Engineering Multi-Agent Systems, In: Proceedings of the 1st international workshop on Agents for Autonomic Computing (AAC 2008) in conjunction with the 5th International Conference on Autonomic Computing (ICAC 2008). June 6, 2008, Chicago, USA, IEEE CS Press, 8 pp. (submitted 21 April 2008).

Khriyenko O., SMART HUMAN ADAPTER - Multi-agent Context-sensitive Visual Resource Browser, In: Proceedings of the Twelfth International Workshop on Cooperative Information Agents (CIA-2008), September 10-12, 2008, Prague, Czech Republic, Springer LNCS, 15 pp. (submitted 21 April 2008).

Katasonov A., Kaykova O., Khriyenko O., Nikitin S., Terziyan V., Smart Semantic Middleware for the Internet of Things, In: *Proceedings of the 5-th International Conference on Informatics in Control, Automation and Robotics*, 11-15 May, 2008, Funchal, Madeira, Portugal, 11 pp.

Terziyan V., SmartResource – Proactive Self-Maintained Resources in Semantic Web: Lessons learned, In: *International Journal of Smart Home*, Special Issue on Future Generation Smart Space, 2008, SERSC publisher, ISSN: 1975-4094, 18 pp.

Katasonov, A., Terziyan, V., SmartResource Platform and Semantic Agent Programming Language (S-APL), In: P. Petta et al. (Eds.), *Proceedings of the 5-th German Conference on Multi-Agent System Technologies (MATES'07)***,** 24-26 September, 2007, Leipzig, Germany, Springer, LNAI 4687 pp. 25-36.

Terziyan V., Predictive and Contextual Feature Separation for Bayesian Metanetworks, In: B. Apolloni et al. (Eds.), *Proceedings of KES-2007 / WIRN-2007*, Vietri sul Mare, Italy, September 12-14, Vol. III, Springer, LNAI 4694, 2007, pp. 634–644.

Nikitin S., Terziyan V., Pyotsia J., Data Integration Solution for Paper Industry - A Semantic Storing, Browsing and Annotation Mechanism for Online Fault Data, In: *Proceedings of the 4th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, May 9-12, 2007, Angers, France, INSTICC Press, ISBN: 978-972-8865-87-0, pp. 191-194.

Salmenjoki K., Tsaruk Y., Terziyan V., Viitala M., Agent-Based Approach for Electricity Distribution Systems, In: *Proceedings of the 9-th International Conference on Enterprise Information Systems*, 12-16, June 2007, Funchal, Madeira, Portugal, ISBN: 978-972-8865-89-4, pp. 382-389.

Khriyenko O., 4I (FOR EYE) Technology: Intelligent Interface for Integrated Information, In: *Proceedings of the 9th International Conference on Enterprise Information Systems (ICEIS-2007)*, Funchal, Madeira - Portugal, 12-16 June 2007.

Khriyenko O., 4I (FOR EYE) Multimedia: Intelligent semantically enhanced and context-aware multimedia browsing, In: *Proceedings of the International Conference on Signal Processing and Multimedia Applications (SIGMAP-2007)*, Barcelona, Spain, 28-31 July 2007.

Khriyenko O., Context-sensitive Multidimensional Resource Visualization, In: *Proceedings of the 7th IASTED International Conference on Visualization, Imaging, and Image Processing (VIIP 2007)*, Palma de Mallorca, Spain, 29-31 August 2007.

Naumenko A., Semantics-Based Access Control in Business Networks, Jyvaskyla Studies in Computing, *PhD Thesis*, Volume 78, Jyvaskyla University Printing House, 215 pages, 2007.

Srirama, S., and Naumenko, A., (2007). Secure Communication and Access Control for Mobile Web Service Provisioning, In: *Proceedings of International Conference on Security of Information and Networks (SIN2007)*, 8-10th May, 2007.

Naumenko, A., SEMANTICS-BASED ACCESS CONTROL - Ontologies and Feasibility Study of Policy Enforcement Function , In: *Proceedings of the 3rd International Conference on Web Information Systems and Technologies (WEBIST-07)*, Barcelona, Spain - March 3-6, 2007, Volume Internet Technologies, INSTICC Press, pp. 150-155.

Naumenko A., Katasonov A., Terziyan V., A Security Framework for Smart Ubiquitous Industrial Resources, In: R. Gonzalves, J.P. Müller, K. Mertins and M. Zelm (Eds.), In: *Enterprise Interoperability II: New challenges and Approaches, Proceedings of the 3rd International Conference on Interoperability for Enterprise Software and Applications (IESA-07)*, March 28-30, 2007, Madeira Island, Portugal, Springer, pp. 183-194.

Katasonov A., Kaykova O., Khriyenko O., Loboda O., Naumenko A., Nikitin S., Terziyan V., The Central Principles and Tools of UBIWARE, *Technical Report (Deliverable D 1.1)*, UBIWARE Tekes Project, Agora Center, University of Jyvaskyla, May-October 2007, 118 pp.