# Policy Management Engine for *Ubiware Agent*

Michael Cochez

May 27, 2010

**Abstract**

The *Ubiware* platform is being developed for about two years. The structure of the behavioral engine is in a stable state and supporting components are being developed. One of the facets of a multi agent system (MAS) is policies. Policies are rules which have to be followed by the entities in the platform. For the third version of the platform, the work package concerning policies has been extended. This paper documents the main points of the research I conducted concerning policies between November 2009 and May 2010.

# Contents

1 **Introduction** **2**
1.1 What is *S-APL*? . . . 2
1.2 Ubiware platform . . . 3
1.3 Original workpackage description . . . 4
1.3.1 Research problem statement . . . 4
1.3.2 Development challenges . . . 5

2 **Literature review** **5**
2.1 Objects . . . 5
2.2 Mode . . . 5
2.3 Trigger . . . 6
2.4 Scope expression . . . 6
2.5 Action . . . 6

3 **Policies applied in the Ubiware platform** **6**
3.1 Objects . . . 6
3.2 Mode . . . 7
3.3 Trigger . . . 7
3.4 Scope expression . . . 7
3.4.1 Storing the zone information internally in the agent . . . 8
3.4.2 Storing the information outside the agent . . . 8
3.5 Results for the workpackage . . . 9

4 **Conclusions** **10**

# 1 Introduction

In this introduction, I will explain the fundamentals of the *Ubiware platform*. Agents beliefs are represented in *S-APL*, a language invented by Arthem Katasonov. I will give a quick overview of that language and of the functionality of *Ubiware agents* the work on the Ubiware platform is divided in smaller work packages. In the last subsection, I explain what how the original work package concerning policies was.

## 1.1 What is *S-APL*?

*S-APL* (Semantic - Agent Programming Language) is a language used for *Ubiware Agent* programming and can be summarized as in [3] "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) and utilizes the syntax for rules very similar to that of N3Logic. N3 was proposed as a more compact, better readable and more expressive alternative to the dominant notation for RDF, which is RDF/XML. One special feature of N3 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. In *S-APL*, we refer to formulae as context containers."
The *S-APL* syntax, as described in the developers guide written by Athem Katasonov [2] is as follows:

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

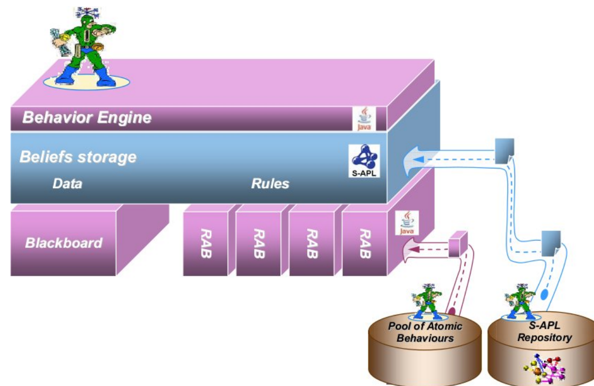


Figure 1: The agent architecture

For the remainder of this document I assume using the namespace sapl: as
`<http://www.ubiware.jyu.fi/sapl#> .`

$G$ is considered the main *Ubiware Agent* belief container and contains everything the *Ubiware Agent* assumes to be general truth.

## 1.2 Ubiware platform

The ubiware platform provides an environment for *Ubiware Agents*. The functionality of a *Ubiware Agent* can be summarized as in [3]

> The architecture of an *S-APL* agent is depicted in Figure 1. The basic 3-layer agent structure is common for the APL approach. 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 concerned with 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 *S-APL*, or if one wants to enable an agent to do some other kind of reasoning beyond the rule-based one. The middle layer is the beliefs storage. What differentiates *S-APL* from traditional APLs is that *S-APL* is RDF-based. This provides the advantages of the semantic data model and reasoning. An additional advantage is that in *S-APL* the difference between the data and the program code is only logical but not any principal. Data and code use the same storage, not two separate ones. 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 are n-ary predicates. *S-APL* is very symmetric with respect to this – anything that can be done to a simple statement can also be done to any belief structure of any complexity.

## 1.3 Original workpackage description

The original workpackage was described as follows:

*Policies* in UBIWARE is a general term to describe in what kind of state a group of agents; an organization and thus by extension the whole system is allowed to be. A closely related concept is *Preference*, i.e. a Preference tells that if the system is able to make a choice, then it should favor a certain option. For example, if an agent has to buy a certain product, it should always prefer to buy the cheaper one if it does not take much more time to deliver.

In a concrete UBIWARE scenario, one might have certain conditions to which the MAS must comply. An example of a Policy could be that if an agent senses a certain problem, it must start an alarm procedure. One important set of policies are the security policies. This package has thus become broader as it was previously defined. Probably the way security policies are defined is going to change in order to fit into the broader view of a policy. Policies have a higher priority than the preferences in this working package since a Preference could be seen as a policy of taking one option instead of another one, it is thus a special case of a policy. Further on, when talking about Policies, we are also talking about Preferences. The tasks in this workpackage are divided in a research and a development part. The next sections elaborate those.

### 1.3.1 Research problem statement

The main task is defining where policies can be applied in the UBIWARE platform. This means answering to the following questions:

- Can we define a policy on top of an organization or only on specific agents?
- Can an organizational policy be enforced by applying policies to individual agents?
- Which other parts of the platform have to be aware of the existence of policies?
- Defining how annotating policies in *S-APL* must be done (this might require an extension of the *S-APL* specification):
- How can policies in general be defined in *S-APL*? In which form can they be applied to the planners/organizations/agents/other reasoners?
- Can they have a unified structure?
- How can a policy be annotated semantically? Does the policy definition give enough information for use of the policy?
- Determining how the policies can be implemented in the platform:
- How can we apply policies to individual agents/how do we deploy policies?
- How can we change the scenario management to also take the policies into account?
- Do we allow run-time changes of policies? Can we cope with changing policies?
- Which components are allowed to change policies? This involves security policies itself!
- What do we do when a policy changes while a certain plan is already executing?
- What to do when a policy changes and brings an agent into an inconsistent state?

### 1.3.2 Development challenges

Design of a prototype of a policy enforcement engine for both the organizational policies and policies of individual agents. Determining whether the platform has to be changed in order to be able to use the policies (policy enforcement engine) or whether they can be implemented by combining existing technologies available in the platform. Implementing the chosen solution in the UBIWARE prototype.

# 2 Literature review

In the literature, different aspects of policies are considered, they have different names and there is no consensus about what is necessary. The following components reoccur frequently:

## 2.1 Objects

The studied papers have a focus on heterogeneous systems composed of different types of objects ([6]). This means that there is some kind of directory or other tree structure which contains objects (components of a certain type). The types of the components are divergent. Objects can be instances of classes, interfaces, external resources, policies itself, managers, users and so on. With other words, any artifact one can identify in the real world can be represented as an object in the system. The objects have a clearly defined interface which consists of certain methods and attributes. Objects are both the subject and target of policies. The point of having policies themselves as objects too is that [6] there are advantages in treating policies as managed objects and structuring them into domains, so that an authorization policy can be defined to control which managers are permitted to modify a set of policies or to define "meta policies" about policies.

## 2.2 Mode

According to [7] "Policies are one aspect of information which influences the behavior of objects within the system. *Authorization policies* define what a manager is permitted or not permitted to do. They constrain the information made available to managers and the operations they are permitted to perform on managed objects. *Obligation policies* define what a manager must or must not do and hence guide the decision making process" Objects in the system thus do not have their own behavior, but are completely guided by their obligation policies. The referred paper talks about positive/negative authorization/obligation.

- positive authorization : I am allowed to do so (I can buy a car)
- negative authorization : I am not allowed to do so (I steal a car)
- positive obligation : I am asked/obliged to do so (I buy a car)
- negative obligation: I am asked/obliged not to do so (I do not buy a car)

The main difference between negative authorization and negative obligation is that the authorization is checked outside of the object. The authors in [4] on the other hand do not provide any means of negative obligation, but still distinguishes the other types.

### 2.3 Trigger

An event which causes a policy to apply to certain objects. In the literature negative obligation policies do not have triggering conditions. Sloman [6] includes the triggering condition into the policy itself and calls those policies “State based policies”. The triggers are incorporated into the selection expression to select the set of subjects or targets to which the policy will be applied. Furthermore, policies can have constraints to restrict their applicability. Those constraints are subdivided in temporal constraints, parameter value constraints and preconditions. In [5] events are seen as triggers for policies. Events are defined as separate concepts and are subdivided in three classes: polling, notification and timing.

### 2.4 Scope expression

Policies should be specified for sets of objects instead of individual objects[7], and as stated in the same paper “A search over all reachable objects, within a distributed system to determine these sets is impractical.” The proposed way to define those sets is using advanced set operators and sub domains, thus giving an algebra to calculate the set of objects affected by the policy. [4] takes a different approach. Resolving domains is externalized from the policy processor. “Here (on the operational level) the subject and target objects are domain expressions or object descriptions with a well defined syntax. The domain service is asked for resolution of domain expressions when the policy is activated.”

### 2.5 Action

The particular action which is authorized or obligated. In [5] actions are described in the policy and have a success/no-success result. This result might trigger other actions. [6] regards actions as invocation of a method calls on objects in the system.

## 3 Policies applied in the Ubiware platform

The previous section showed how different aspects of policies are described and studied in the literature. In this section, I will apply those aspects to the Ubiware platform.

### 3.1 Objects

The Ubiware platform has a unified approach to objects. The only existing object type in the system is the *Ubiware agent*. In order to represent other artifacts from the real world, an adapter agent is used. An example thereof is the use of a *Personal user agent* to represent human users which want to interact with the system. Since there are no objects in the system besides *Ubiware agents*, they will be both the target and subject of the policies.

## 3.2 Mode

As described in [2], the Ubiware agent has a concept called role. A role is a piece of code an agent can use to perform certain actions. Role descriptions and obligation policies are not distinguishable. Except for the fact that having a role does not mean that the agent is going to perform it. It only indicates that it knows how to perform the role. (Or even ‘knew’ how to in case it has already removed the role description) According to [1] an agent is (among other things) autonomous: “Autonomous, in the sense that it can act without direct intervention from humans or other software processes, and controls over its own actions and internal state.” Result thereof is that agents should thus be able to take their own decisions. This also implies that external enforced policies are also not suitable way to model preferences. Authorization policies are not contradicting because they do not limit the reasoning of the agent. They only describe whether an agent is or is not allowed to perform a certain action. These policies can thus be seen as part of the environment the agent resides in.

## 3.3 Trigger

Triggers as they exist in the studied systems are not needed in the Ubiware system. The triggers are fired by certain sensors, time events or changes of attributes. In Ubiware the triggers can and will be implemented by the agents themselves. The agents have the ability to perceive signals from their environment, can receive notifications in the form of messages and have a notion of time. Furthermore they can fire internal rules when certain beliefs exist in their beliefs structure. The fact that agents have capabilities to perform the trigger themselves, shows the overlap between obligation policies and roles once more.

## 3.4 Scope expression

While working on the scope expression for the Ubiware system I found out that the Ubiware system has a conceptual problem. There is no division into domains of agents nor do they have distinct types. This implies that we are not able to select agents to which a policy should be applied. When an agent in the platform, requests a certain role, it registers itself with that role to the directory facilitator. One might reason that to select a set of agents to apply a certain policy to, we can ask the directory facilitator for the agent which play a cetain role. The problem with that approach is however that we cannot tell whether the agent is at this moment playing the role, nor whether it still knows how to play it. Another problem is that we do not know in which environment it is playing the role. Let me give an intuitive example of a Dutch auction: Consider a Seller having role S and 5 Buyers B_1 ... B_5 having role B and an Auction_Authority A. Let us now say that we want to express that the Seller is allowed to advertise it is latest price to the Buyers. There is thus the following kind of policy:

> positive authorization for (Agent with role S) sendMessageAbout-Price (Agent with role B)

This seems to work fine at first sight. However, a problem arises when multiple auctions are going on at the same time on the platform. In this case

any agent with role S would be allowed to sendMessageAboutPrice to any agent which has role B. Even if these agents are not playing those roles in the same Dutch auction scenario. It appears thus to be incorrect to take the set of agents which have a certain role according to the directory facilitator.

We need thus a possibility to say:

> positive authorization for (Agent with role S in auction #b452l)
> sendmessageaboutprice (Agent with role B in auction #b452l)

Where #b452l is an ID for the auction.

What we notice is that we have a need to keep track on in which zones, organizations, groups, organizational units(OU), auctions, ... an agent is involved. We have elaborated the following two approaches to tackle this problem:

### 3.4.1 Storing the zone information internally in the agent

In this approach, the agent itself keeps track of groups it is involved in. One (or multiple, negotiating) agents create a group by choosing a group-ID or getting one from an *Authority agent* which all agent in the group trust. Then the role descriptions must have rules written based on certain zones, or a role is only valid within a certain context of the agent. The knowledge about the zone could be codified as follows:

```
eg. sapl:I sapl:haveRoleInZone {
    zone:ID sapl:is #uniqueZoneID.
    role:ID sapl:is #uniqueRoleID .
    { ... rules ...} sapl:is rulesforrole.
    {... policies ...} are externalpolicies.
    {...policies...} are internalpolicies.
    }
```

It should then be possible to manage them by meta-rules. The original idea was to have external and internal policies to support the idea of restricting which policies can be applied at the same time. External policies would then restrict with which other policies this policy can co-exist. One problem is that all current code needs to be transformed to work in this vision. Furthermore, there is no control on whether an agent is really member of a group if it says so. Policies would then be enforced by rules inside agents. This idea did not find much support when proposed to the other members of the research group. Mainly because this approach has a flaw in the sense that the agent himself is able to change the policies whenever it wants to.

### 3.4.2 Storing the information outside the agent

In the light of other changes in the third version of the platform (where a few agents will be infrastructural agents), there is a possibility to use one of those agents to keep track of the organizations the agents reside in. For the scope of this document, I will call this agent Zone-Agent (ZA). This agent would keep a database in some form about which agent has which role in which organization. Another agent, which I will call Policy Agent (PA) will then keep the policies' descriptions and refer to the ZA whenever it needs to translate a scope expression.

The chosen solution is further elaborated and will be implemented as follows: Ubiware agents are only able to perform external actions (action outside of their

belief structure) by using Reusable Atomic Behaviors (RAB [3]). This includes message sending and receiving, sensing, acting, ... The authorization policies will be applied at the point where an agent tries to perform an action. When this happens, the PA will be asked whether this action is permitted. The PA will be able to solve questions of the form:

```
sapl:I sapl:want {
  sapl:you sapl:reply {
    agentName pol:isAllowedTo {
      {sapl:I sapl:do action}
      sapl:configuredAs
      {p:parametername sapl:is "value"}
    }
  }
}
```

It will do so by checking its internal knowledge about this particular agent. If needed, the PA will ask the ZA whether this particular agent belongs to a zone with attributes that allows the agent to perform a certain action. The Policy agent and the Zone agent will get their information from agents sending certain information to them. The point is that to be allowed to send this information, an agent must already be in a zone which allows him to do so. This indicates that there must be one agent known initially to the Policy agent which is allowed to change the policies. We decided that this role would be reserved for the root agent which is a personal user agent which is always started when the platform starts up.

## 3.5 Results for the workpackage

- Policies in the *Ubiware platform* are only sense-full when considering authorization policies.
- To apply a policy to an individual agent, it is sufficient to apply it to an organization the agent is member of.
- When an ambiguity arises, the PA is responsible to solve the problem.
- The PA and ZA will have to be implemented and will be the only agents directly aware and interacting with policies. The exact description and ontology of the policies will be shared by those two components.
- The enforcement of a policy does not necessarily need an extension of the S-APL language. There might be a need to differ between a failing action and one which is forbidden because of a policy. This is still an open issue.
- When policies are applied to other components they are able to check whether every action planned is going to be allowed by the selected component.
- A policy will be annotated using zones and attributes thereof.
- Since policies are not changing the behavior of the agent directly, there is no need to annotate what the result of applying the policy will be.
- Policies are subject to change at any point and agents must take into account that they might not be allowed to perform an action even if they had received a permission before.

# 4 Conclusions

Policies in *Ubiware* are only sense-full when considering authorization policies. This because obligation policies would lead to a contradiction with the fact that an agent is an autonomous entity. An agent should thus itself decide whether it is doing what its environment enforces it to do and take the consequences if it does not. Authorization policies can only be checked outside of the agent, because of security. When checking policies outside of the agents, there is a need to use a system to make sets of agents. We decided to use a division of the agents in zones. Each zone will have a set of policies connected to it. Those ideas will be implemented in a Zone and Policy agent, which together answer the question whether a certain agent is allowed to take a certain action with certain parameters.

# References


[1] Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[2] A. Katasonov. Ubiware platform and semantic agent programming language (S-APL) - developer's guide. Technical report.

[3] A. Katasonov and V. Terziyan. Semantic agent programming language (S-APL): A middleware platform for the semantic web. In *Semantic Computing, 2008 IEEE International Conference on*, pages 504 –511, 4-7 2008.

[4] T. Koch, C. Krell, and B. Kramer. Policy definition language for automated management of distributed systems. In *Systems Management, 1996., Proceedings of IEEE Second International Workshop on*, pages 55 –64, 19-21 1996.

[5] Emil C. Lupu and Morris Sloman. Towards a role-based framework for distributed systems management. *Journal of Network and Systems Management*, 5:5–30, 1997. 10.1023/A:1018742004992.

[6] M. Sloman, J. Magee, K. Twidle, and J. Kramer. An architecture for managing distributed systems. In *Distributed Computing Systems, 1993., Proceedings of the Fourth Workshop on Future Trends of*, pages 40 –46, 22-24 1993.

[7] Morris Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2:333–360, 1994. 10.1007/BF02283186.