JaSA: Semantically Aware Agents to improve Adaptivity and Context-Awareness of Smart Environments

Alice Grandi

2nd cycle degree programme in Computer Engineering
Second Engineering Faculty, Cesena (Italy)
alice.grandi@studio.unibo.it

Abstract

The aim of this project is to accomplish the need for adaptivity and context-awareness in smart environment applications, combining agent-oriented programming with ontological reasoning. This concretises in the realization of Jason Semantically Aware (JaSA) agents; their potentiality will be shown within JaCa-smart[1], an extension of the Jason-CArtAgO framework that enables Jason agents to act in Smart-M3, an open source software platform developed within the SOFIA- Smart Objects For Intelligent Applications european research project [2].

1 Introduction

As Zambonelli and Parunak predicted in 2004 [16], models and methodologies for software development have dramatically changed nowadays in order to fit with modern computing systems, which characteristics are the *leit-motiv* of well known research fields as pervasive and ubiquitous computing.
In computational systems of today user interaction and computational intelligence can be considered together as part of the broader concept of *ambient intelligence*. This notion is defined by Marzano and Aarts [1] through the following key technology features:

1. **Embedded** - there are many devices, connected through a network and integrated into the environment;
2. **Context aware** - the computational system can manage the context in which it operates;

[1] See [15].
[2] http://www.sofia-project.eu/

3. **Personalized** - the system is made to meet the users' needs;
4. **Adaptive** - the system responds suitably to external changes;
5. **Anticipatory** - the system anticipates the users' desires without conscious mediation.

This work focuses in increasing adaptivity and context awareness in software systems related to ambient intelligence, commonly known as *Smart Environments*. The first step towards this goal consists in sifting an interaction model for this kind of systems and then finding the programming paradigm that could best fit to that model.
Agent-Oriented Programming revealed to be very fit; this, added to the usefulness of combining agents with ontological reasoning, as many literature propositions [6, 7, 8] prove, gave rise to the idea to extend Jason agents to make them semantically aware. Those agents will then be used within JaCa-smart, to show how they can improve the potentiality of Smart Environment systems.

2 Towards semantically aware agents

2.1 Interaction model for Smart Environment Applications

A suitable definition of Smart Environments that clarifies their interaction model is given by Abowd et al. [2] as

> *one that adapts to the needs of the information consumer, in terms of input/output capabilities, as he/she moves and accesses information in a dinamically changing environment. Potential changes include the number of information consumers or information providers, or changes in location of the individual or objects in the environment.*

Another enlightening suggestion is given by Coutaz et al. [4]

> *In ubiquitous computing, the interaction space is ill-defined, unpredictable and emerges opportunistically.*

From these words can be inferred that the most evident feature of the interaction space of Smart Environments is *non determinism*; for this reason the traditional GUI interaction, in which designers develop solutions for a predetermined interaction space usually characterized by single-user interactions, is not adequate.
Non determinism in Smart Environments is a feature and a need; this resides in the *autonomy* of components, that cannot be set aside in these systems, and also in the need for *openness* and *situatedness*, as different (and in a changing number) actors and a changeable environment are involved. The main motivations for this issue are stated by Zambonelli et al. [16], with reference to modern software systems in general:

- *In an open world, autonomy of execution enables a component to move across systems and environments without having to report back to (or wait for acknowledgment by) its original application.*
- *When components and systems are immersed in a highly dynamic environment whose evolution must be monitored and controlled, an autonomous component can be effectively delegated to take care of (a portion of) the environment independently of the global flow of control.*
- *Several software systems are not only made up of software components, but also integrate computer-based systems, which are by their very nature autonomous systems, and can be modeled accordingly.*
- *As the size of a software system increases, the need of delegating control to components is no longer simply a matter of performance, but becomes a matter of conceptual simplicity.*

Another fundamental feature of Smart Environment interaction space is *locality*; in fact, as Zambonelli et al. [16] point out

> *In an open world, components need some form of context-awareness to execute and interact effectively. However, for a component to be made aware of its context effectively (and efficiently), this context must necessarily be locally confined.*

Strong in the concepts that emerged up to now, the characteristics of the interaction space of a Smart Environment can be subsumed in the following:

1. **Situatedness** - components interact with the environment, and are affected by the environment itself, that is a primary abstraction within the system;
2. **Openness** - the system can dinamically change in structure, cardinality, organisation, ...;
3. **Autonomy** - components have independent flows of control;
4. **Locality** - each component is context-aware, by controlling a portion of the environment and interacting with neighboring components.

These characteristic lead again to the model proposed by Zambonelli et al. [16], to which we can refer to give a model for the interaction space of Smart Environments:

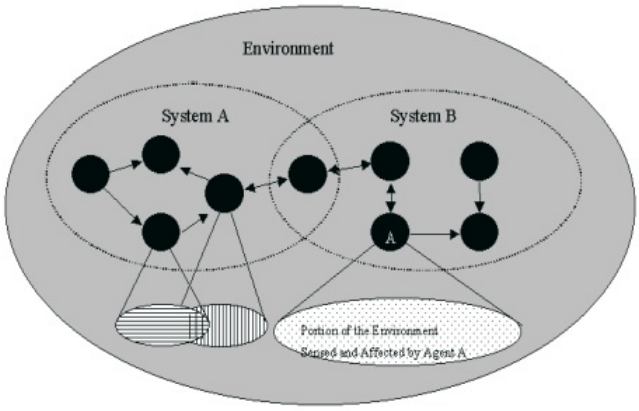

Figure 1: Smart Environments interaction model

What emerges first is that systems (dashed ellipses) and their components (black circles) are immersed in an environment, and are typically modeled as a set of environment partitions. Each component can sense and affect a local portion of the environment. It's also worth noting that, if the portions of the environment that two components access overlap with each other, they can interact indirectly with each other via the environment.

2.2 Agent-Oriented Programming is the way

To build a system which interaction model is the one depicted in Figure 1, it would be very useful for programmers to be able to create a set of interacting autonomous first-class-abstraction components, adaptable to changing conditions, each one with its well defined job(s) to do, in relation to the portion of environment it has to occupy with.
This abstraction exists and is named MAS (Multi-Agent System), which components are called Agents; building a system on this basis is agent-oriented programming. Without going into details (that can be discovered through plentiful literature about this), let's just recall the main features of an agent [9], that perfectly fit to the interaction model of Smart Environments without requiring further motivations:

- **Autonomy**: is the fundamental and definitory feature of an agent, as every other feature descends from it; agents are autonomous since they encapsulate the thread of control and only information can cross the agent boundaries, in order to realize inter-agent interaction. It's important to underline that in this context autonomy means self-government, self-regulation, self-determination - hence an agent doesn't need to have a goal, but a goal is a possible feature;

- **Proactivity**: agents encapsulate not only control, but also the rule to govern it;
- **Situatedness**: an agent comes with its own model of action, that is strictly coupled with the context where the action itself takes place;
- **Reactivity**: it comes hand in hand with situatedness, as any non-trivial action model requires some form of perception of the environment, for example to check action pre-conditions, or to verify the effects of actions on the environment. This feature can also be seen as a deliberate reduction of proactivity;
- **Interactivity**: since agents are subsystems of a MAS, they interact within the global system, by means of information exchanging; for this reason agents are aforesaid to be *social*.

Even if a goal is not a definitory feature for an agent, it may be included in relation to Smart Environments; in fact, as Honkola et al. [5] state

> *... smart space applications are better seen as scenarios that can be executed to meet the goals of the user.*

This assertion symbolizes the natural correspondence between a goal and a software component (as already seen, an agent) within a Smart Environment Application. For this reason a Smart Environment can be described by means of task models, defined in [8] as "*... description of an interactive task to be performed by the user of an application through the application's user interface*". This fact leads to take in account another possible feature for agents; in fact, as Van Welie et al. [14] demonstrate, task models should be able to represent the psychological, social, environmental and situational aspects of agents and their tasks. Injecting *intelligence* into agents is therefore a need; to do this, the Belief-Desire-Intention (BDI) model can be used for constructing agents that shift from weak agency to the strong one, as intelligence comes in addition to the five key features of agents mentioned above.
Let's now go deeper into the BDI model to explain better how it fits to the developement and working of Smart Environment Applications.

2.2.1 Use of BDI agents in Smart Environment Systems

BDI agents are suitable to build Smart Environment Systems since they are based upon the abstraction of *Mental State*; Rao and Georgeff clearly explain this in [12] defining Mental States as a worth abstraction for developing agents, so they can act in a class of application domains characterized by some practical limitations and requirements as:

- At any instant of time there are many different ways in which an environment may evolve;
- At any instant of time there are many actions or procedures the agent may execute;

- At any instant of time the agent may want to achieve several objectives;
- The actions or procedures that (best) achieve the various objectives are dependent on the state of the environment (i.e., on the particular situation, context);
- The environment can only be sensed locally;
- The rate at which computations and actions can be carried out is within reasonable bounds to the rate at which the environment evolves.

It's easy to recognize that these are exactly the characteristics of Smart Environments, as previously mentioned: non-determinism, situatedness, locality.
The behaviour of agents with Mental States is defined by internal states, that are imitative of cognitive mental states; they can be divided into *Epistemic states*, representing agents knowledge (percepts, beliefs), and *Motivational states*, representing agents objectives (goals, desires).
These agents can select an action to execute on the basis of their actual mental states; this is what is called *practical reasoning*, and agents realize this in two different stages: the first one is called *deliberation*, during which the agent decides what it desires to achieve, formulating its so-called intentions. The second one is *means-ends reasoning*, in which the agent decides how to achieve its desires, selecting a given course of actions (one or more so-called plans).
Rao and Georgeff [11] propose an abstract architecture to build this kind of agents named BDI model, which name implies the concepts it's founded on:

- **B**eliefs: they constitute the *informational state* of the agent, as they express the knowledge about the current state of the world;
- **D**esires: they form the *motivational state* of the agent, with the restriction that multiple active desires must not be in contradiction;
- **I**ntentions: they are emergent properties reified during the reasoning cycle of an agent by selecting a given plan for achieving a given goal or desire; hence they are the *deliberative state* of the agent.

There are many advantages in using agents with Mental States to build MAS; as Omicini et al. state in [10], first of all it eases the development of agents exhibiting complex behaviour, providing us a familiar way of understanding and explaining agents. In fact the developer of the MAS can take the perspective of a cognitive entity engaged in complex tasks, thinking about what He would do in the same situation. What's more, agents with Mental States simplify the maintainance and verification of agent-based application, and are a very useful abstraction to build agents that have to communicate and interact with users or other system entities.

2.3 Integrating BDI agents in Smart-M3

Proved that BDI agents are a powerful tool to suit our purposes, let's now present our reference software platform, Smart-M3. Then we will see a proposal

of integration between BDI agents and Smart-M3 architecture, JaCa-smart; this will be the starting point from which building Smart Environment Applications constituted by agents with semantic capabilities.

2.3.1 Smart-M3: an open source platform within SOFIA project

As already stated in the introduction, while trying to sift an interaction model for Smart Environment Applications, the models and patterns used to build traditional software systems are inadequate in the pervasive computing domain; this obviously concerns architectural patterns, too. An interesting approach is taken in SOFIA (Smart Objects For Intelligent Applications), an European research project which aim is to make information in the physical world available for smart services, in order to create new user interaction experience and interface concepts relying on cross-industry interoperability.
The architectural pattern used in SOFIA is the blackboard pattern, semantically enriched with the use of ontologies to enable interoperability without requiring standardization. SOFIA also takes the agent and publish/subscribe concepts to create an infrastructure for systems where small and mobile devices can interact through an event-based and space-based coordination model; this infrastructure is termed Smart-M3, as it enables Multi-domain, Multi-device, Multi-vendor interoperability.
As shown in figure 2, the main components of Smart-M3 are its agents, named

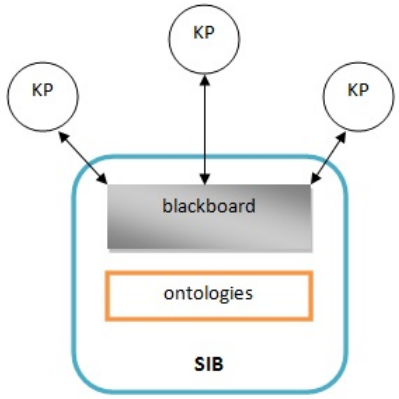

Figure 2: Smart-M3 infrastructure

Knowledge Processors (KPs); they can be heterogeneous devices spanning from embedded domains to the Web, and operate autonomously and anonymously by sharing information through the blackboard. The other main component is the Semantic Information Broker (SIB), that acts as the information store of the Smart Space and contains the blackboard, ontologies, reasoner and required service interfaces for the KPs.

BDI agents have been integrated within this simple architecture in a technological solution named JaCa-smart; let's see how.

2.3.2 JaCa-smart

JaCa-smart is based on JaCa, an agent-oriented programming framework born from the conjunction of the Jason[3] agent programming language and CArtAgO[4], a platform which provides Java-based APIs and a runtime environment to program and execute artifact-based environments for multi-agent systems.
The model and technology details of JaCa-smart can be found in [15]; here it's just worth explaining that it enables a mediated and semantically-annotated communication among Jason agents; in fact a CArtAgO artifact is used as a bridge to bring Smart-M3 SIB(s) to the cognitive level of agents so they can communicate through the space using its low-level functionalities, and also share their beliefs and goals with other agents populating the smart environment(s). This way agents can communicate at an intentional level in a trasparent manner, relying on a given and publicly accessible ontology expressing BDI concepts (beliefs, goals, agents, and so on).

2.4 Goal of this work

As many literature propositions [6, 7, 8] and our previous discussion affirm, a very important challenge posed by the development of smart environments is the requirement for agents to behave in an intelligent way. JaCa-smart does not provide a solution to this issue, but it can be considered an appropriate starting point from which building a framework to realize effective intelligent systems, with increasing adaptivity and context-awareness compared with current available technology used to develop smart environment systems.
This is exactly the aim of this work, and there are two main reasons why the growth of technologies for smart environments should follow this direction; the first one is related to the potentiality a smart system can have, and is well explained by Niezen et al. in [8]

> *When the goals, plans, desires, and beliefs of different agents are explicitly represented in the ontologies, this information allows them to share a common understanding of their "mental" states, helping them to cooperate and collaborate. If we are then able to represent the human user's mental states in the ontology, it may help software agents to reason about the specific needs of the users in a pervasive environment.*

The second reason is closely related to the first one, but deals with software engineering; as Klapiscak and Bordini [6] state

[3] http://jason.sourceforge.net
[4] http://cartago.sourceforge.net

When agent programmers are familiar with existing ontologies for the domain of interest, they can take advantage of the knowledge already represented there to make their programs much more compact and elegant, besides the obvious interoperability and reuse advantages.

These two reasons, that will be windedly justified later, clearly suggest that Smart Environment Applications can be truly adaptive and context-aware only if its agents can reason about their knowledge.
As seen, JaCa-smart is just an enabling technology for communication by means of ontological information, but awareness of ontologies remains only within the Smart-M3 SIB(s); the idea underlying this project is then to expand BDI agents with ontological reasoning, so the agents themselves will be aware of ontologies, and the use of structured information in Smart Environment Applications will be as transparent as possible.

3 JaSA agents

With JaCa-smart as the reference platform for this work, combining agent-oriented programming with ontological reasoning means to inject semantic capabilities within Jason agents; so analising the main components of a Jason agent and their participation when the agent is running will clear up what should be changed (and how) in order to reach the aim stated formerly.

3.1 Working of a Jason agent

As explained in [3], a Jason agent is composed by an AgentSpeak interpreter within an "overall agent architecture" which provides perception, acting and communication with other agents, indispensable to work effectively in a multi-agent system.
The interpreter, named as Agent, is the cognitive part and constitutes the internal representation of a Jason agent; it includes:

- the *belief base*, a data structure where the agent stores beliefs (the agent's knowledge about the world);
- the *plan library*, a data structure for storing AgentSpeak plans for achieving given goals;
- some agent-specific *selection functions* that handle internal and external events, intentions and plans;
- the *belief update function*, which updates the agent's belief base from perception of the environment;
- the *belief revision function*, responsible for determining the necessary modifications in the belief base as a consequence of executing an intention;
- a "circumstance" constituted by pending events, intentions, and other structures that isn't worth analysing here.

A Jason agent works by means of Interpretation Cycles, during each of them the components interact as can be seen in the following figure:

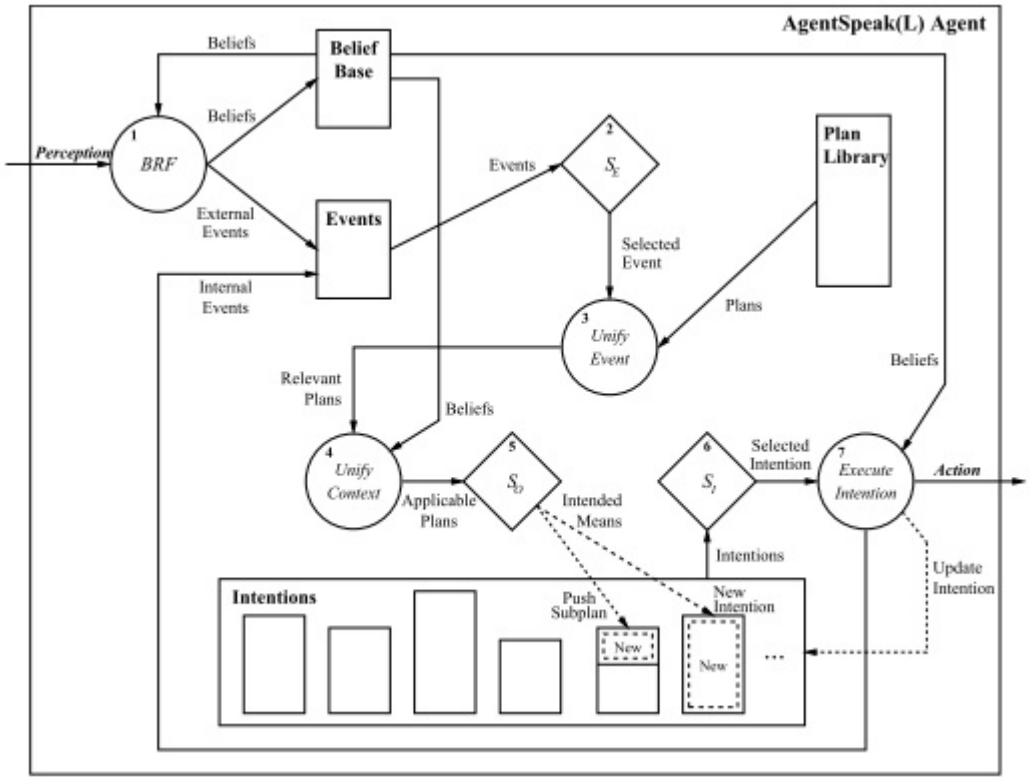

Figure 3: Interpretation Cycle of a Jason agent

At every Interpretation Cycle, the list of `Events` is updated by the `BRF` (Belief Revision Function) with events generable either from perception of the environment (`External Events`) or from the execution of intentions (`Internal Events`, generated when subgoals are part of the body of a plan).
Then the S_E selection function selects an event that must be unified with triggering events `te` in the heads of plans (whose AgentSpeak(L) syntax is `p ::= te : ct <- h`); this generates a set of `Relevant Plans`, whose context part `ct` has to be checked to be part of the agent's beliefs. Those plans who satisfy this condition form the set of `Applicable Plans`, from which the S_O selection function selects the next plan to be executed (`Intended Means` for the event that triggered that plan), and pushes it on the top of an exsisting intention (if the event was an internal one), or creates a new intention in the set of intentions (if the event was external).
At this point a single intention has to be selected to be executed in the current cycle by the selection function S_I; then the cycle goes on in a different way depending on where the intention has been taken from:

- if the intention was taken from the stack of subplans, it has a plan on the top, so the formula in the beginning of its body `h` has to be taken for

execution. If the formula is an *achievement goal*, the agent will perform an action on the environment and a corresponding internal event will be generated. If the formula is instead a test goal, it will be tested for truthfulness in the belief base.

- if the intention was taken from the set of intentions, it could be either a basic action to be performed or a test goal; in the case of a test goal, the belief base will be searched for a belief atom that unifies with the predicate of the test goal and, if the test succeeds, the test goal itself is removed from the intentions. In the case of an action, instead, the agent informs the architecture component responsible for the agent effectors what action is required, and that action is simply removed from the list of intentions.

When all formulae in the body `h` of a plan have been executed, the whole plan is removed from the intentions, and so is the achievement goal that generated it (if there was any). This way a cycle of execution is completed, and can start again from checking the state of the environment (that certainly changed as agents have acted on it).

Now that how a Jason agent works is clear, let's reason about what components should be modified and even how, to make Jason agents semantically aware.

3.2 Needed modifications

A useful cue about what components of a Jason agent should be modified is given by Moreira et al. in [7]; that paper infact sets the grounds for combining agent-oriented programming with ontological reasoning, but the aim there was to define AgentSpeak-DL, a version of BDI agent-oriented programming language AgentSpeak, based on description logic rather than predicate logic.
As far as the aim of this project is to modify Jason agents, that are interpreters for AgentSpeak(L), it won't be improper to be inspired by the ideas used in [7] to explain the advantages of joining AgentSpeak with description logics; they can be rightly extended to the conception of joining agents and semantic capabilities, and are the followings:

1. "queries to the belief base are more expressive as their results do not only rely on explicit knowledge but can be inferred from the ontology;
2. the notion of belief update is refined given that (ontological) consistency of a belief addition can be checked;
3. retrieving a plan for handling an event is more flexible as it is not based solely on unification but on the subsumption relation between concepts; and
4. agents may share knowledge by using ontology languages such as OWL."

These four points suggest the main modifications needed to integrate semantic within Jason agents and now, except for requirements that can arise during implementation, let's try to go deeply into them one by one.

3.2.1 Modifications to the belief base

The first change that could struck makes Jason agents able to manage ontologies, both schemas (or, as in Description Logic, TBox) and instances (ABox). The syntax of beliefs in AgentSpeak(L)[5] consists of a set of atomic formulae `at::= P(t1,...,tn)`, where `P` is a predicate symbol and `t1,...,tn` are standard terms of first order logic; considering that the ontology language used in Smart-M3 (RDF) is based upon *classes*, *relationships* and *properties* for describing ontology schemas and *instances*, *properties* and *values* for building ontology instances (the objects of knowledge), it should be allowed to express beliefs which meaning is like:

- The object `o` is instance of class `A`;
- The objects `o1` and `o2` are in relation through the property `P`;
- The object `o` satisfies property `P` with value `v`;
- Class `B` is sub-class of class `A`;
- Classes `A` and `B` are related through property `P`.

An interesting solution for this issue is suggested by Klapiscak et al. in [6]; it consists in enriching the normal notation used to express beliefs in Jason with an annotation (o/1) that defines to which ontology schema the belief refers to. This forms the notion of *Semantically Enriched Literal* (*SE-literal* for short); the atomic term of the ontology annonation is the reference to a single in-memory ontology instance, an *ontology label*. Klapiscak et al. defined four types of SE-Literals, that are better explained through some examples related to `fastfood`[6] and `society`[7] ontologies (`ff` and `s` for short, respectively):

`pizza(pizza1)[o(ff)]`	**Class Assertion**: `pizza1` is a member of class `pizza`
`owns(food_inc, italiana)[o(s)]`	**Object Property Assertion**: `food_inc` is related to `italiana` through property `owns`
`hasPrice(pizza1, 2.2)[o(ff)]`	**Data Property Assertion**: `pizza1` satisfies property `hasPrice` with value `2.2`
`all_different([italiana, jim])[o(s)]`	**All Different Assertion**: `italiana` and `jim` are mutually distinct individuals

Table 1: Example SE-Literals with meaning

[5] See [3], chapter 2.2.
[6] http://www.dur.ac.uk/t.g.klapiscak/onts/fastfood.owl
[7] http://www.dur.ac.uk/t.g.klapiscak/onts/society.owl

Using SE-Literals part of the belief base can be conceived as a set of in-memory instantiations of OWL ontology schemas; this, in combination with a Description Logic reasoner, facilitates the use of knowledge as agent can make inferences based on its beliefs, and also an enhanced assurance of knowledge consistency can be realised. Moreover, the ontology annotation makes it possible to distinguish between common beliefs and semantic ones, and also between different but clashing terms of separated ontologies.

3.2.2 Modifications to the Plan Library

The previously mentioned modification implies that triggering events `te` of plans can be SE-Literals; infact the syntax of triggering events in AgentSpeak(L) is `te ::= +at | -at | +g | -g` where `at` is an atomic formula as shown previously, and `g` represents a goal with syntax `g ::= !at | ?at`. This means, by extension, that SE-Events and SE-Plans can be part of the agent's Interpretation Cycle so an SE-Event can be selected to obtain a set of relevant plans. When this happens, it would be very important and useful, in case a proper plan cannot be found by simple unification, to retrieve plans in a more general sense by using subsumption. This would extend the plan generality currently supported by Jason, available solely through variables in plan triggers.

3.2.3 Modifications to the "overall agent architecture"

To realise what Moreira et al. stated in point 4., already mentioned in section 3.2, the Inter-agent Communication mechanism available in Jason could be used to introduce agents to novel ontologies at runtime. Using SE-Literals in the content of Inter-agent messages, infact, it would be possible to send new axioms within the ABox of some sender-known ontology so the agent who receives that message can extend its known ontologies, and then share this knowledge with other agents. This has strong implications for a MAS openness, as communication can take place with no prior agreement on terminology.

3.2.4 Further modifications

With reference to points 1. and 2. suggested in [7] and reported in section 3.2, the need for performing more complex queries and face harder problems related to consistency (that obviously arise due to the use of ontologies) could be provided with the use of a Description Logic Reasoner.
The use of semantically enriched literals in the belief base, infact, makes it possible to perform queries no longer formed only by the knowledge available in the belief base in the form of formulae, but also of that inferred by the DL-Reasoner, operating over the ontology instances known to the agent.
As regards belief base consistency assurance, the DL-Reasoner could be used to test whether the addition of an assertion within the ABox leads to an inconsistency according to the axioms expressed in the corresponding TBox; and then proper measures to restore consistency could be taken.

4 JaSA project

As stated in the abstract of this paper, the aim of this work is to combine agent-oriented programming with ontological reasoning, in order to be able to build smart environment systems as multi-agent systems; for this reason all previous discussion concretises not only in the modification of Jason agents, but in the realization of a supporting technology for creating MAS in which ontologies have an essential role.
Let's now see the steps that compose the developement of JaSa technology.

4.1 From JASDL to JaSA

The modifications (previously mentioned in section 3.2) that make Jason agents able to manage ontologies have been already implemented in an extension of Jason named JASDL[6], with reference to the agent's belief base; it provides support for using SE-Literals and SE-Plans, and also consistency assurance through the use of the Pellet DL-Reasoner. What JASDL lacks to reach our aim is the semantic description of the MAS environment, in other words, we need agents to interact with the environment as part of their known ontologies, in order to exploit the advantages of using ontological reasoning as regards the environment, too. Hence the idea to build JaSA tecnology as a sort of merge between JASDL, that offers ontological reasoning upon the agents belief base, and JaCa (mentioned in subsection 2.3.2), that makes Jason (and JASDL) agents able to interact within an artifact-based environment.
This decision results in the definition of a JaSA MAS, to be run with Jason, as follows:

- **Environment**: class c4jason.CartagoEnvironment;
- **Agents**:
  - **agent architecture**: agentArchClass jasa.architecture.JaSAAgentArch;
  - **agent class**: agentClass jasdl.asSemantics.JASDLAgent;
  - **agent belief base**: beliefBaseClass jasa.bb.JaSABeliefBase.

So a JaSA MAS exploits CArtAgO environments, which details can be found in [13]; as regards agents, the standard class used in JASDL must be the agentClass, as it provides an implementation of the belief revision function that checks for the consistency of the agent's belief base with the ontologies he knows, whether an SE-Literal is added. The beliefBaseClass, too, has been redefined but is the same used in JASDL, as regards its functioning; it extends the standard Jason belief base in order to manage semantically annotated beliefs (for its details see [6]).
The JaSA agent architecture extends the CArtAgO agent architecture in order to exploit the artifact programming model, together with JASDL inter-agent communication mechanism. To make this work, another step of the project revealed to be necessary, and it will be explained in the next section.

4.2 From Cartago to Cartago-DL

The need to integrate the CArtAgO programming model with the use of semantically enriched literals gave rise to necessary modifications in the CArtAgO mapping of observable properties into beliefs; to understand this, it's worth spending some words about CArtAgO.
When an agents is immersed in a CArtAgO environment, composed by one or more workspaces, the actions an agent can perform is determined by the operations provided by the artifacts available in the workspace(s). An artifact also provides a set of observable properties, which represent dinamically changing properties describing the MAS environment.
In CArtAgO an observable property is a Java object; its general form can be seen as a tuple, with a functor and one or multiple arguments. An agent can focus one artifact, so that the artifact's observable properties are visible to the agent, as they're mapped into the agent's belief base as standard Jason beliefs, with proper annotations to distinguish them, such as the id of the artifact that is the source of the belief. Since an agent focuses an artifact, every change of the observable properties visible through the artifact will be detected as changes to the belief base.
As already stated, in JaSA we want to introduce a semantic description of the MAS environment, so the beliefs related to the environment have to be semantically enriched in order to comply with JASDL, as regards the functioning of the belief base, the mechanism of plans selection and consistency check. For this reason the JaSA agent architecture (JaSAAgentArch) extends the CArtAgO agent architecture (CAgentArch) so that when an observable property is defined within an artifact, it is added to the belief base of the agents focusing the artifact according to the syntax defined in JASDL for expressing OWL property assertions (see Table 1, page 13).
Since an artifact can show observable properties related to its own state, or observable properties that describe the state of other artifacts populating the environment, the mapping of OWL properties into CArtAgO-DL (the particular version of CArtAgO developed within JaSA project) is the following:

- If the property describes the state of the artifact that shows it, it will be defined within the artifact as a tuple with one argument (the value of the property), and it will be represented in the belief base of the agents focusing the artifact as an SE-Literal with two arguments (where the first one is the id of the artifact), and the proper ontology annotation that states the ontology to which the property belongs to.
  For example, if we have an artifact named Counter, that has a property count with an integer value N, it is defined in the artifact as `count(N)`, and seen by the agents that focused the artifact Counter as the belief `count(CounterId,N)[o(OntologyName)]`.

- If the property shown by the focused artifact describes the state of another artifact, instead, it will be defined in the focused artifact as a tuple which predicate is the name of the property, and two arguments, respectively

the subject and the object (or the value) related through the property. That tuple will be represented in the belief base of the agents focusing the artifact as an SE-Literal with the name of the property as the predicate, the same two arguments, and the proper ontology annotation.
Let's consider two examples to explain this better:

1. *The focused artifact shows an object property which subject is another artifact* - e.g. the artifact detector shows if a person is in a room, so the property contains of an artifact named room: `contains(room,alice)` will be represented in the belief base of the agent focusing the artifact detector as `contains(room,alice)[o(OntologyName)]`;
2. *The focused artifact shows a data property which subject is another artifact* - e.g. the artifact thermostat shows the temperature of another artifact that represents a room, named room01: `hasTemperature (room01,20)` will be represented in the belief base of the agent focusing the artifact thermostat as `hasTemperature(room01,20)[o(OntologyName)]`.

The need to express the observable properties of artifacts according to an ontology, naturally led to the idea to create artifact manuals as OWL ontologies; here another step of JaSA project begins, and it is described it in the next section.

4.3 Artifact manuals as OWL ontologies

As discussed in section 2.1 of this paper, a Smart Environment is characterised by four fundamental properties: situatedness, openness, autonomy and locality. As regards the last two ones, it's easy to recognize that they are inborn in the concepts of agent and multi-agent system, provided by the Jason platform. Situatedness has been introduced in Jason within JaSA project, relying on JaCa (already described in section 2.3.2); the step of JaSA project regarding the development of artifact manuals, instead, addresses openness. Having documents in which artifacts are described so that agents can reason about their content, as these documents are ontologies, enables an open interaction between agents and artifacts, as agents do not need to know the name of the artifact(s) they have to interact with; they just need to know the properties or the operations provided by the artifacts that would be useful to reach their goals. To realize this, a proper and common ontology has been pointed out to be the base for the manual of every artifact being part of a JaSA MAS:

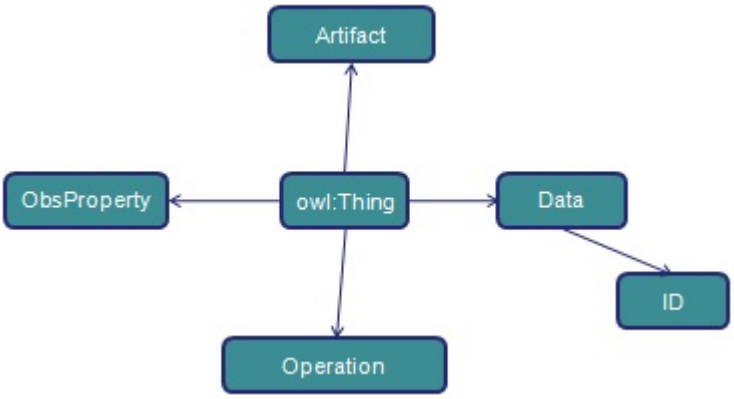

Figure 4: Artifacts base ontology: class tree

In addition to the already mentioned concepts of artifact, observable property and artifact operation, modelled by the classes *Artifact*, *ObsProperty* and *Operation* respectively, the base ontology that constitutes an artifact manual in JaSA includes the class *Data* with its subclass *ID* that represents an identification data for any instance of class owl:Thing; we mainly use it to identificate artifacts, but it can be useful in case of need to identificate other entities being part of the MAS. To be useful to agents reasoning, the artifact manual must include instances, too; we will see this in detail through an example in section 5.
Since the idea of artifact manuals was already present in CArtAgO, the consultation of OWL manuals by JaSA agents has been realized through operations provided by a new default artifact being part of CArtAgO-DL workspaces, named `ManRepoArtifact`; that operations are:

- `storeManual(ArtifactModelName,Uri)`: used by an agent to store a manual at workspace level. Since a manual corresponds to an artifact model, once it is stored at workspace level, it will be possible for every other agent that needs to interact with an artifact that suits that model to consult its manual.

- `consultManual(ArtifactModelName)`: used by an agent to look for a manual in a workspace and, if available, consult it.

Once this step of JaSA project has been completed, just one ingredient is needed to make JaSA a complete technology for Smart Environment applications developement; it constitutes the last substantial step of the project, and it will be described in the next section.

4.4 A new artifact to interact with Smart-M3 SIB(s)

Since the abstract of this paper JaCa-smart solution has been presented as the infrastructure that JaSA agent would use to be able to act in Smart Environment Applications based upon Smart-M3 technology, in particular using a CArtAgO artifact as a bridge to communicate with Smart-M3 SIB(s) (see section 2.3.2). As in JaSA important modifications of Jason (through JASDL) have been introduced, together with a new version of CArtAgO named CArtAgO-DL, it revealed to be necessary to make changes in JaCa-smart, too.
This changes concretised in a new artifact to enable JaSA agents communication with Smart-M3 SIB(s) named `SmartM3Bridged`. The differences between this new artifact and the corresponding one realized in JaCa smart concern operations interfaces and the SIB events handling; let's discuss them in detail separately.

4.4.1 New artifact operations interfaces

The first important change in JaCa-smart bridge regards artifact operations interfaces. As can be seen in [15] in sections 5.2.1 and 5.2.2, the JaCa-smart artifact offers to agents operations that recall the same functionalities offered by a Smart-M3 SIB with predicate `rdf(Subject,Predicate,Object)` as input parameter to model rdf triples (that is the basic piece of information that an agent, or KP, can exchange with a SIB, as shown in figure 2, section 2.3.1).
Since the peculiarity of JaSA agents is to be able to handle SE-Literals, that correspond to axioms conforming a certain ontology, it seemed natural to model rdf triples exchanged with Smart-M3 SIB(s) as SE-Literals too, in order to avoid unnecessary data conversions that could cause overhead and confusion.
Let's consider the example an agent wants to insert into a SIB the triple `(person01,isInRoom,5)`; with JaCa-smart this means that the agent would attempt to execute the intention `insert(rdf(person01,isInRoom,5),_)`, that in JaSa becomes `insert(isInRoom(person01,5)[o(room)],_)`.

4.4.2 SIB events handling

The other important change in JaCa-smart bridge regards SIB events handling, that is what happens when a triple stored inside the SIB undergoes a modification.
In JaCa-smart (for details see [15], section 5.2.1) a change in a triple stored into the SIB is carried into a signal (following the semantics given in CArtAgO) into the MAS environment, in easier words an agent subscribed to certain triples changes, when these occur, will see a new incoming belief with the form `smartM3_event(...)`, so that he can react properly.
In JaSA, instead, subscribing to some triple modification means to define a new observable property whose name is the same of the triple predicate, while the arguments are the triple subject and the triple object. As already explained in section 4.2, this means that there is a mapping between the triple stored in the SIB and a belief in the belief base of the agent subscribed to that triple; when

the triple undergoes a modification, the corresponding belief is automatically updated. On the other hand, unsubscribing to some triple modification means to delete the corresponding observable property defined in `SmartM3Bridged` artifact, when the agent performed the subscription.

At this point the development of JaSA agent-oriented platform can be considered a complete support (but not final, as you can read in the last section of this paper) to realize Smart Environment Applications as multi-agent systems; let's see how in the next section.

5 How to build a JaSA MAS - the thermostat example

In this section we will see a through an example how to setup and run a MAS using JaSA. The code of the example we are going present can be found in folder `/JaSa/examples/thermo`.

5.1 Problem: controlling the temperature of a room

5.1.1 Problem analysis

Let's suppose we want to realize a Smart Environment Application which goal is to control the temperature of a room, that naturally fluctuates according to climatic changes; as the system user, we will be able to set our target temperature inserting it through a GUI.
The temperature of the room is controlled by a thermostat, that senses the current temperature and receives the desired one; if the current temperature of the room is different from the target temperature set by the user, the thermostat reacts setting the desired temperature in the room. Every change in the current temperature of the room or in the target temperature set by the user is perceived by the thermostat, so that the temperature of the room is always the one desired by the user.
The peculiarity of the architecture of our system is that the room, the GUI and the thermostat could be located independently, so they need a common space for communication and information sharing; since the reference infrastructure for building Smart Environment Systems in JaSA project is Smart-M3 (see section 2.3), a Smart-M3 SIB will be used to suit our purposes.
The main entities characterising the domain model of our problem are then:

- a *room*;
- the *user*;
- the *thermostat GUI*;
- the *thermostat*;
- a *Smart-M3 SIB*.

5.1.2 Logical architecture

In the light of the just outlined entities, the logical architecture of the software system that suits our problem can be represented as in figure 5.

5.1.3 Planning architecture

The logical architecture thought over through the Agents & Artifacts metamodel, that is the conceptual framework upon which CArtAgO is based (therefore JaSA), evolves in the planning architecture represented in figure 6.

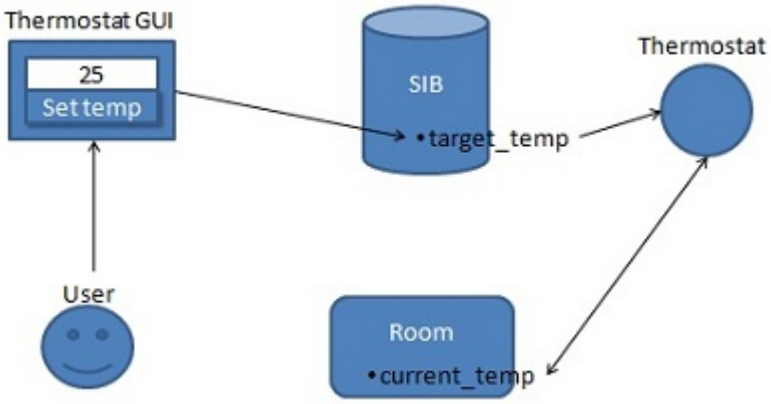

Figure 5: Logical architecture of thermostat application

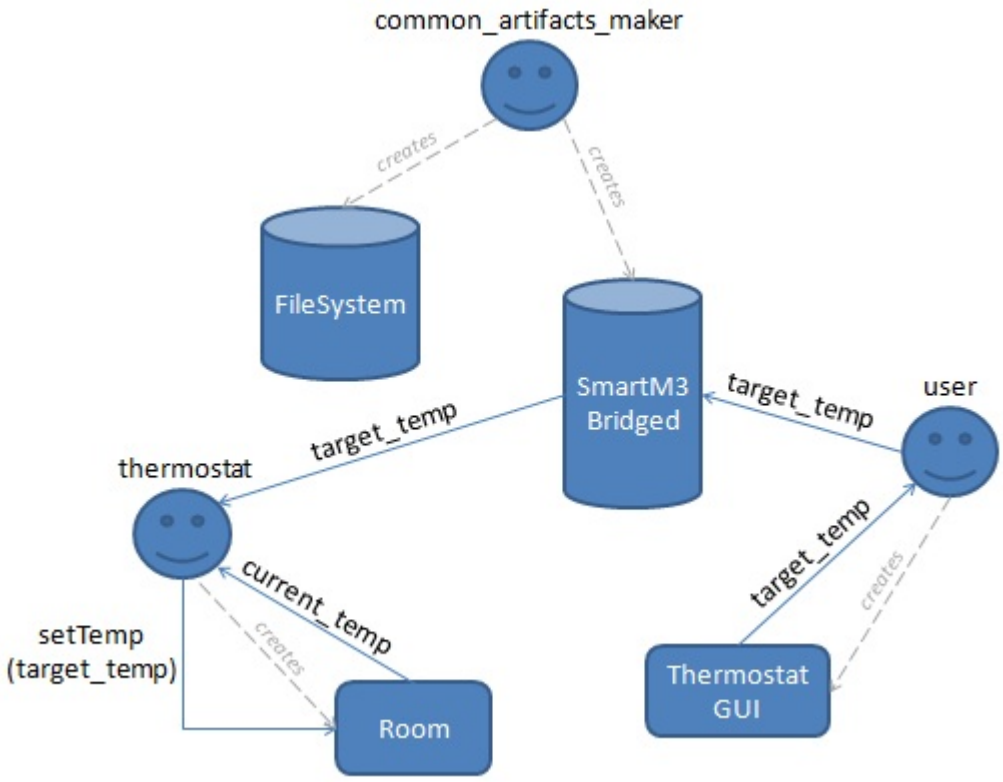

Figure 6: Planning architecture of thermostat application as a JaSA MAS. The smileys represent agents, the other entities are artifacts.

5.1.4 Supporting ontologies

As explained in section 4.3, every artifact that offers observable properties in JaSA must have its own manual, that is an OWL ontology describing the features of the artifact (according to the base ontology which class tree is represented in figure 4).
For this reason we will define three ontologies (the artifact FileSystem doesn't need it, since it doesn't have observable properties):

1. *room.owl* - describes `Room` as an artifact characterised by:
   - an observable property, named `current_temp`, which domain is an artifact ID and range the int data type. It represents the current temperature of the room;
   - an operation, named `setTemp`, which has one argument that is the temperature to be set in the room.

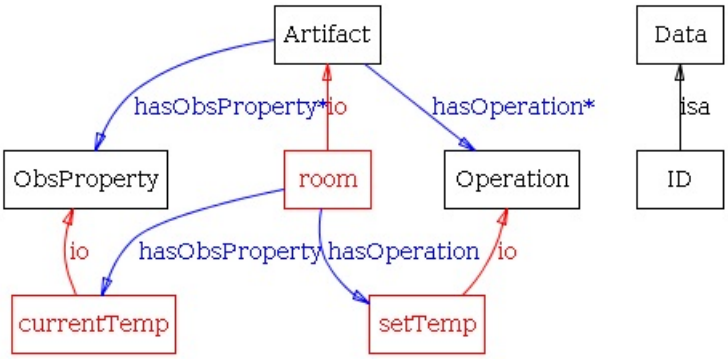

Figure 7: room.owl OntoViz graph

2. *thgui.owl* - describes `Thermostat GUI` as an artifact characterised by:
   - an observable property, named `target_temp`, which domain is an artifact ID and range the int data type. It represents the (last) temperature set by the user through the GUI;

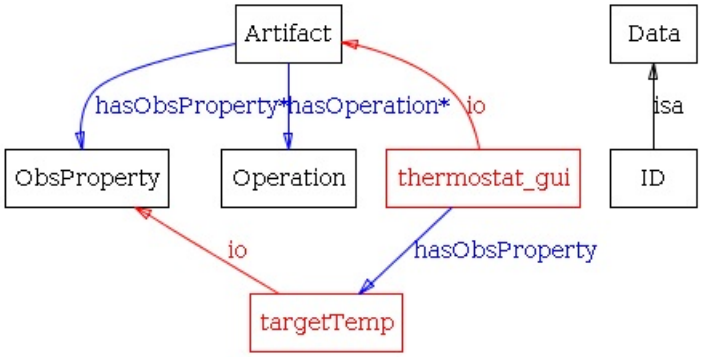

Figure 8: thgui.owl OntoViz graph

3. *SmartM3Thermo.owl* - describes `SmartM3Bridged` as an artifact that offers the Smart-M3 SIB(s) base functionalities (for details see the JaSA javadoc, class `smart_m3.SmartM3Bridged`), plus communication with the `Thermostat GUI` artifact, hence the observable property `target_temp`.

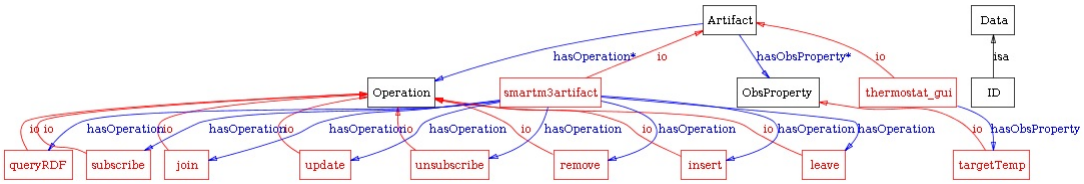

Figure 9: SmartM3Thermo.owl OntoViz graph

5.2 Setting up the MAS configuration file: thermo.mas2j

Let's have a look to the .mas2j configuration file of our example, to understand how it must be set to configure correctly a JaSA MAS.

```
MAS thermo {
   infrastructure: Centralised

   environment: c4jason.CartagoEnvironment

   agents:
       thermostat
       [
       jasdl_ontologies="room_Room,smart_m3_SmartM3Bridged",
       jasdl_room_Room_uri="/onts/room.owl",
       jasdl_smart_m3_SmartM3Bridged_uri="/onts/SmartM3Thermo.owl"
```

```
        ]
        agentArchClass jasa.architecture.JaSAAgentArch
        agentClass jasdl.asSemantics.JASDLAgent
        beliefBaseClass jasa.bb.JaSABeliefBase;

        user
        [
        jasdl_ontologies="thermostat_gui_ThermostatGUI,smart_m3_SmartM3Bridged",
        jasdl_thermostat_gui_ThermostatGUI_uri="/onts/thgui.owl",
        jasdl_smart_m3_SmartM3Bridged_uri="/onts/SmartM3Thermo.owl"
        ]
        agentArchClass jasa.architecture.JaSAAgentArch
        agentClass jasdl.asSemantics.JASDLAgent
        beliefBaseClass jasa.bb.JaSABeliefBase;

        common_artifacts_maker agentArchClass c4jason.CAgentArch;

    classpath: "../../lib/*.jar"; "../../bin";
    aslSourcePath: "asl";
}
```

The points worth noting are the following:

- The `environment` must be set to standard CArtAgO environment;
- A JaSA agent (here `thermostat` and `user`) must be declared following the convention used in JASDL; between square brackets must be written the short names of artifact manuals the agent will use (parameter `jasdl_ontologies`); they will be the argument of ontological annotations of beliefs, and must be the same as the name of the artifact Java class, including the package, with underscores in place of dots (since they create problems with the mas2j parser).
  Then artifact manuals URIs must be declared, using the same short names as declared above (parameter `jasdl_<short name>_uri`; the path is relative to the .mas2j file directory).
  The agent architecture class, agent class and belief base class must be, respectively, `JaSAAgentArch`, `JASDLAgent` and `JaSABeliefBase`, as already explained in section 4.1.
- The `classpath` must include the folder where all jars distributed with JaSA project can be found, and the directory containing the .class files of the artifacts.

5.3 MAS thermo artifacts

5.3.1 FileSystem

It is a simple CArtAgO artifact, that has only one operation, named `getCurrentDir`. Its feedback parameter is the absolute path of the current directory where the MAS project files are placed in the file system.

5.3.2 SmartM3Bridged

It is a CArtAgO artifact, that offers agents the basic operations to interact with a Smart-M3 SIB:

- `join`, useful to join the space and be able to use all the other operations;
- `leave`, useful to leave the space when no more interaction is needed;
- `insert`, to insert a RDF triple into the space;
- `remove`, to remove a RDF triple from the space;
- `update`, to replace a RDF triple already present in the space with a new one;
- `queryRDF`, to search into the space if there are triples matching a given template;
- `subscribeRDF` and `unsubscribe`, to be notified when triples matching a given template are inserted into the space, removed or updated, and then cancel the subscription when it isn't useful anymore.
  When an agent subscribes to a triple template, a new observable property is defined with predicate and arguments corresponding to the ones of the ontological belief passed as argument to the `subscribeRDF` operation; e.g. if an agent subscribes to the triple `hasName(person01,null)[o(ont)]` a new observable property `hasName(person01,<value>)` is defined, where `<value>` is the value found when a triple corresponding to the template indicated in the ontological belief is inserted, removed or updated. This way when a triple present into the space matches the template and undergoes modifications, the corresponding observable property is updated so that the agent that has subscribed sees the modifications in the corresponding belief it has in its belief base (such belief is created by the artifact when the observable property is defined).

5.3.3 Room

It is a CArtAgO GUI artifact characterised by the operation `setTemp(int temp)` to change the value of the current temperature of the room with the desired temperature, and the observable property `current_temp(RoomId,Temp)` that shows the current temperature of a room according to its variations, due to natural changes or user settings.

5.3.4 Thermostat

It is a CArtAgO GUI artifact that shows the observable property `target_temp (ThermostatGUIId,Temp)`, that is the temperature set by a user through a GUI.

5.4 MAS thermo agents

5.4.1 common_artifacts_maker

```
// Agent common_artifacts_maker in project thermo

/* Initial beliefs and rules */

/* Initial goals */
!start.

/* Plans */
+!start : true
<- makeArtifact("fs","test.FileSystem");
   getCurrentDir(Dir);
   .concat("file://localhost/",Dir,"/onts/SmartM3Thermo",OntUri1);
   storeManual("smart_m3.SmartM3Bridged",OntUri1);
   makeArtifact("smart_m3","smart_m3.SmartM3Bridged",["127.0.0.1","10010","X"],SSId).
```

This is a simple Jason agent, that has the task of creating the artifacts co-used by the agents of the MAS, hence `FileSystem` and `SmartM3Bridged`. It also stores the corresponding artifact manuals in the workspace, so that they will be available to all the agents that might need to consult them.

5.4.2 user

```
// Agent user in project thermo
// Receives the user's target temperature from a GUI and sends it to the SIB.

/* Initial beliefs and rules */
set(0).

/* Initial goals */
!start.

/* Plans */
+!start : true
<- !setupManualRepo;
   consultManual("smart_m3.SmartM3Bridged");
   consultManual("thermostat_gui.ThermostatGUI");
   makeArtifact("thermostat_gui","thermostat_gui.ThermostatGUI",[],TId);
   focus(TId);
```

```
   join(Ack)[artifact_name("smart_m3")].

+target_temp(X,T)[o(thermostat_gui_ThermostatGUI),artifact_name(_,"thermostat_gui")] :
set(0)
<- -set(0);
   +set(1);
   +last_target_temp(T);
   insert(target_temp(X,T)[o(thermostat_gui_ThermostatGUI)],_)[artifact_name("smart_m3")].

+target_temp(X,T)[o(thermostat_gui_ThermostatGUI),artifact_name(_,"thermostat_gui")] :
set(1)
<- ?last_target_temp(LT);
   -last_target_temp(LT);
   +last_target_temp(T);
   update(target_temp(X,LT)[o(thermostat_gui_ThermostatGUI)],
   target_temp(X,T)[o(thermostat_gui_ThermostatGUI)],_)[artifact_name("smart_m3")].

/*****************************/
+!setupManualRepo : true
<- ?discover_fs(FSId);
   getCurrentDir(Dir);
   .concat("file://localhost/",Dir,"/onts/thgui.owl",OntUri);
   storeManual("thermostat_gui.ThermostatGUI",OntUri).

+?discover_fs(FSId) : true
<- lookupArtifact("fs",FSId).

-?discover_fs(FSId) : true
  <- .wait(100);
     ?discover_fs(FSId).
```

This agent is an example of JaSA agent, since it handles ontological beliefs and uses CArtAgO-DL artifacts. Its task is to observe the `Thermostat GUI` artifact in order to insert into the SIB the triple `(target_temp,ThermostatGUIId,X)` (where X is a temperature value).
As can be seen looking at the plan triggered by the initial goal `!start`, the `user` agent first of all creates the artifact `Thermostat GUI` and then consults the manuals of the artifact it is going to use. After focusing on the Thermostat GUI artifact, it joins the Smart-M3 SIB created by the `common_artifacts_maker` agent in order to be able to insert triples into the SIB.
With the next two plans, the `user` agent handles modifications in the observable property of the `Thermostat GUI` artifact that is represented by the belief `target_temp(X,T)` with its annotations; every time a new target temperature is set by the human user, this observable property is updated and the new target temperature information is inserted into the SIB as the triple `(target_temp,ThermostatGUIId,T)` (where T is the temperature set by the

human user through the prearranged GUI).

5.4.3 thermostat

```
// Agent thermostat in project thermo
// Receives the user's target temperature from the SIB and, if necessary,
sets it in the target room.

/* Initial goals */
!start.

/* Plans */
+!start : true
<-!setupManualRepo;
  consultManual("smart_m3.SmartM3Bridged");
  consultManual("room.Room");

  ?discover_smart_m3(SSId);
  focus(SSId);
  join(Ack)[artifact_id(SSId)];

  !findControllerAndSubscribe;

  makeArtifact("room01","room.Room",[],RId);
  focus(RId).

+!findControllerAndSubscribe : artifact(Controller)[o(smart_m3_SmartM3Bridged)] &
              hasObsProperty(Controller,targetTemp)[o(smart_m3_SmartM3Bridged)]
<- +controller(Controller);
   subscribeRDF(target_temp(Controller,null)[o(smart_m3_SmartM3Bridged)],SubID);
   +subscriptionID(SubID).

+current_temp(room01,CT)[o(room_Room),artifact_name(_,"room01")] : controller(ArtName) &
 target_temp(ArtName,T)[o(smart_m3_SmartM3Bridged),artifact_name(_,"smart_m3")] & CT\==T
<- setTemp(T)[artifact_name("room01")].

+target_temp(ArtName,T)[o(smart_m3_SmartM3Bridged),artifact_name(_,"smart_m3")] :
controller(ArtName) & current_temp(room01,CT)[o(room_Room),artifact_name(_,"room01")] &
CT\==T
<- setTemp(T)[artifact_name("room01")].

/*****************************/
+!setupManualRepo : true
<- ?discover_fs(FSId);
   getCurrentDir(Dir);
   .concat("file://localhost/",Dir,"/onts/room.owl",OntUri);
```

```
   storeManual("room.Room",OntUri).

+?discover_fs(FSId) : true
<- lookupArtifact("fs",FSId).

-?discover_fs(FSId) : true
<- .wait(100);
   ?discover_fs(FSId).

+?discover_smart_m3(SSId) : true
<- lookupArtifact("smart_m3",SSId).

-?discover_smart_m3(SSId) : true
<- .wait(100);
   ?discover_smart_m3(SSId).
```

This is an example of JaSA agent, too, and its aim is to observe the artifact Room in order to sense its current temperature, and adjust it according to the target temperature desired by the human user. Since that information is stored into the Smart-M3 SIB by the `user` agent, the `thermostat` agent has to consult the corresponding artifact manual, as well as the manual of the artifact Room, that is created accomplishing the first goal of the plan triggered by the initial goal `!start`.
After that, the `thermostat` agent joins the Smart-M3 SIB, and then looks for a temperature controller (in our case the `Thermostat GUI` artifact), since it is a device with its own independent location and it can't be known a priori; this is done executing the plan triggered by the `!findControllerAndSubscribe` achievement goal. Here is interesting noting that the `thermostat` agent takes advantage of the information acquired through `SmartM3Bridged` artifact manual consultation to find the artifact he needs to know to gather the target temperature information.
Once the controller ID is found, the `thermostat` agent subscribes to the triples which template is `(target_temp,ControllerId,null)` in order to be notified about every change in the target temperature set through the controller GUI. From now on, the `thermostat` agent is ready to react to changes of the current temperature in the room and of target temperature set by an human user thanks to the two corresponding plans (triggered respectively by the ontological beliefs corresponding to `current_temp` and `target_temp` properties), so that the current temperature of the room is always equal to the one set through the GUI.

6 Conclusions & future work

In this paper an innovating programming model for Smart Environment Applications has been proposed, together with a software platform that implements it. The innovating push innate in JaSA project constitutes a step forward in two directions: on one side it shows how the agent programming paradigm is suitable to handle informations described through ontologies, on the other how ontologies can be exploited to improve context awareness and situatedness of components of software systems. These characteristics, as windedly discussed in this paper, have fundamental importance in Smart Environment Applications where the interaction between users and computational intelligence leads to ambient intelligence.
The realization of JaSA technology proves that extending the Jason agent programming language with the power of description logic has a significant impact in the way agent-oriented programming works in general, and in particular for the developement of Smart Environment Applications; the usefulness of combining ontological reasoning within an agent-oriented programming language was already clear, as explained in this paper and in the referenced literature: this work implements a practical application of those pretentions, so concretely supports them.
Future work aims at using JaSA in a real world practical application within the SOFIA project: a pilot at the Bologna municipality building that adresses to show how the maintenance process of a building can benefit from the SOFIA interoperability platform (which is presented in this paper in section 2.3.1). This application involves interesting cross domain interactions, such as mobile devices which receive notifications if a fault is detected in the environment, that communicate with onboard car systems that can guide operators to the building where the fault has to be repaired.
Another aim of future work is to raise the abstraction level from which JaSA agents perceive the MAS environment through the `SmartM3Bridged` artifact, so that the environment planning of a JaSA MAS implementing a Smart Environment System will consist solely in defining artifacts of higher level of perceptions and actions on the environment properties; those artifacts will encapsulate the Smart-M3 SIB(s) primitives, so that the space will be used to create interoperability, remaining hidden to agents. This way the developement of Smart Environment Applications using the agent-oriented platform JaSA will be even more simple, immediate and efficient than it is now as the system state will be handled in the smart space, and the MAS will be the easy way to access and modify it.

7 Acknowledgements

I gratefully express thanks for the support lent, fundamental to follow out the JaSA technology, by Alessandro Ricci, researcher affiliate at DEIS, University of Bologna, at Second Engineering Faculty in Cesena, Italy (e-mail:

a.ricci@unibo.it).

References

[1] E. Aarts, *Ambient intelligence: a multimedia perspective*. Multimedia, IEEE, vol. 11, no. 1, pp. 12-19, 2004.

[2] G. Abowd and J. Sterbenz. *Final report on the inter-agency workshop on research issues for smart environments*. Personal Communicationc, IEEE, 7(5):36-40, October 2000.

[3] R.H. Bordini and J.F. Hübner. *Jason - A Java-based interpreter for an extended version of AgentSpeak*. Release version 0.9.5, February 2007.

[4] J. Coutaz, J.L. Crowley, S. Dobson and G. Garlan. *Context is key*. Communications of the ACM, 48(3):49-53, March 2005.

[5] J. Honkola, H. Laine, R. Brown and O. Tyrkkö. *Smart-M3 Information Sharing Platform*.

[6] T. Klapiscak and R.H. Bordini. *JASDL: A Practical Programming Approach Combining Agent and Semantic Web Technologies*. In M.Baldoni et al. (Eds.): DALT 2008, LNAI 5397, pp. 91-110, 2009. Springer-Verlag Berlin Heidelberg 2009.

[7] A.F. Moreira, R. Vieira, R.H. Bordini and J.F. Hübner. *Agent-Oriented Programming with Underlying Ontological Reasoning*. In M.Baldoni et al. (Eds.): DALT 2005, LNAI 3904, pp. 155-170, 2006. Springer-Verlag Berlin Heidelberg 2006.

[8] G. Niezen, B.J.J. van der Vlist, J. Hu and Loe M.G. Feijs. *From Events to Goals: Supporting Semantic Interaction in Smart Environments*.

[9] A. Omicini, *Agents: From Premises to Definition*, slides from Multi-Agent Systems LM course, Second Engineering Faculty, Cesena (Italy) 2010.

[10] A. Omicini and M. Piunti, *Agents as Intentional Systems*, slides from Multi-Agent Systems LM course, Second Engineering Faculty, Cesena (Italy) 2010.

[11] A.S. Rao and M.P. Georgeff, *An abstract architecture for rational agents*. In B. Nebel, C. Rich and W.R. Swartout, editors, 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR '92), pp. 439-449, Cambridge, MA, USA. Morgan Kaufmann. Proceedings.

[12] A.S. Rao and M.P. Georgeff, *BDI agents: From theory to practice*. In V.R. Lesser and L. Gasser, editors, 1st International Conference on Multi Agent Systems (ICMAS 1995), pp. 312-319, San Francisco, CA, USA. The MIT Press.

[13] A. Ricci and A. Santi, *CArtAgO by examples*. Version 2.0.1, DEIS, Università di Bologna, Italy.

[14] M. van Welie, G.C. van der Veer and A. Eliëns. *An ontology for task world models*. Proceedings of DSV-IS98, Abingdon, pp. 3-5, 1998.

[15] A. Zagnoli. *Designing and Developing Smart Environments using Intelligent Agents and Smart-M3 technology*.

[16] F. Zambonelli and H.V.D. Parunak. *Towards a paradigm change in computer science and software engineering: A synthesis*. The Knowledge Engineering Review, 18, 2004.