tATAmI 1: Towards Agent Technologies for AmI

— Technical Report —

Andrei Olaru [b], Marius-Tudor Benea [a,b], Amal El Fallah Seghrouchni[a], and Adina Magda Florea [b]

a) LIP6 - University Pierre and Marie Curie, France,
b) Computer Science Department - University Politehnica of Bucharest, Romania
`cs@andreiolaru.ro`,
`{marius-tudor.benea,amal.elfallah}@lip6.fr`,
`adina.florea@cs.pub.ro`

Abstract. In this report we describe the tATAmI S-CLAIM framework for the development and deployment of AmI applications. The framework provides a simple development language and a platform allowing cross-platform deployment and simulation / visualization tools and it allows an agent designer to design and deploy agents quickly and repeatable. The programming language has few, agent-oriented, primitives and the platform offers the possibility to deploy agents in various local or distributed setups, including on the Android mobile OS. We also provide some scenario implementation examples.

1 Overview

In this document we describe the tATAmI platform, which is part of the unified framework tATAmI – S-CLAIM (comprising an AOP language and a platform), used in order to develop and to deploy agent-based applications for AmI. The framework is composed of the S-CLAIM (Smart Computational Language for Autonomous, Intelligent and Mobile Agents) programming language, described in [1], and, as we already mentioned, of the tATAmI (towards Agent Technologies for Ambient Intelligence) platform as well. tATAmI supports the execution of S-CLAIM agents in an environment based on Jade, [2], Java and Android, integrates web-services in order to assure interoperability with other platforms and, together with S-CLAIM, it offers good solutions to problems such as context-awareness, mobility or the quick and easy development and execution of scenario-based simulations.

S-CLAIM and tATAmI are the solution to our vision for AmI. However, they can be successfully used to develop a wide range of other agent-based applications, especially in situations where there is a need for MAS characterized by mobile, cognitive, agents and continuously-changing topologies, that need to be deployed on hybrid networks of workstations and mobile devices.

We designed this framework to be easy to use (this being one of our main goals), while still having a high expressiveness. This is a great advantage, as it can considerably reduce the gap between the design and the implementation phases when developing multi-agent based applications. Hoping to prove this advantage, we provide in this document, beside a series of technical aspects about tATAmI, some implementation examples using S-CLAIM and tATAmI, the most detailed one being based on the SmartRoom scenario that was introduced in [1].

2 Requirements

In this section we describe the requirements we wanted our framework to satisfy. They are:

– a declarative high-level language to describe agents, that can be easily used, considering that good AmI solutions to real world problems can come even from developers without a strong technical background;
– a flexible koowledge representation allowing translation to and from XML or other serialization formats, because of the need to store or to load knowledge, to display knowledge in a user-readable form, or to use different representations of knowledge;
– a way to represent knowledge patterns and a powerful matching mechanism, for context-awareness purposes;
– further context awareness capabilities, through agent hierarchies and ambient calculus, [3], insipred hierarchical mobility and agent management features;
– support for scenario-based simulations (the scenarios being described in XML format) and a method for the quick and repeatable execution of simulations;
– agent communication based on standard formats and protocols (i.e. compliance with FIPA);
– interoperability with other platforms, provided by the integration of web services and by the possibility to expose agents as web services as well;
– a good tracking and visualizing system;
– the possibility to deploy agents on hybrid networks of devices, comprising mobile devices as well (i.e. Android devices), very important from an AmI point of view, as a MAS application that steps out of the classical domain into AmI has to integrate many technologies and to consider several aspects. The system must be endowed with sensors, GPS, cameras, smart screens, etc. tATAmI is able to assure this, at a certain level, by exploiting these characteristics of the Android devices.

3 Context-awareness

Context-awareness is an advanced feature able to make a system more intelligent and it is an important aspect to consider for the applications developed using tATAmI. It is a capacity of an application to understand the situation the user is in, in order to be able to change its behaviour depending on, and according to, the recognized context, so that it could provide a better experience for the user, adapting to changes quickly. Moreover, to understand the user's intentions, it is necessary to understand the reason that has led to, or that has changed the user's course of actions. This reason can be inferred from the contextual information focused on that user. Context-awareness is considered as a core feature of any user-centered application.

One definition for context is *the set of environmental states and settings that either determines an application's behavior or in which an application event occurs and is interesting to the user*, [4]. There are several ways to classify the contexts. However, for our framework we divided the context into four different types, that correspond to four different sources of data: *computational context*, *physical context*, *user context* and *temporal context.*

The computational context and the physical context comprise all the information about the devices in the system like their characteristics, their communication capabilities, their location, etc. and also information about the environment like the temperature, the state of the weather, etc. The user context includes all the information available about the user: his profile, preferences, actual actions, and also his location. The temporal context is based on information like the current time, the duration of the events, etc. For the agents in our framework the contextual information is stored in the agent's knowledge base.

4 Structure of the platform

The tATAmI platform is structured on three main components, or parts: the *Agent*, the *Simulation*, and the *Visualization.*

The *Agent* component is the central component of the platform, and it deals with everything that is related to a normal tATAmI agent. An agent contains several aspects (Figure 1), and is based on the (i.e. extends the class) Jade agent. We will present this component in Section 4.1.

The *Simulation* component contains additional classes and processes that allow a user to easily deploy agents on multiple machines, run simulations according to scenarios, and control the agents' lifecycle (Figure 3(a)).

The *Visualization* component allows the user to obtain various information about the activity of all of the agents in the system (like logs, hierarchical information, etc.) and view it on a single machine. It also manages the automatic positioning of the agents' graphical interface on the machine's screen, in such a way that the layout allows a user to view them optimally without the need to reposition them manually (Figure 3(b)).

Also related to this section is the fact that the tATAmI is able to run on multiple platforms – not only on Linux or Windows, but also on the Android platform for mobile devices. In order to achieve this, the Agent component has been implemented so that it is compatible with all platforms that support Jade and Java. Details on how this was done are presented in Section 5.2.

The tATAmI project is open source and has been made public on GitHub[1], together with documentation, code style rules, guide, etc. We are putting an

[1] `https://github.com/tATAmI-Project`

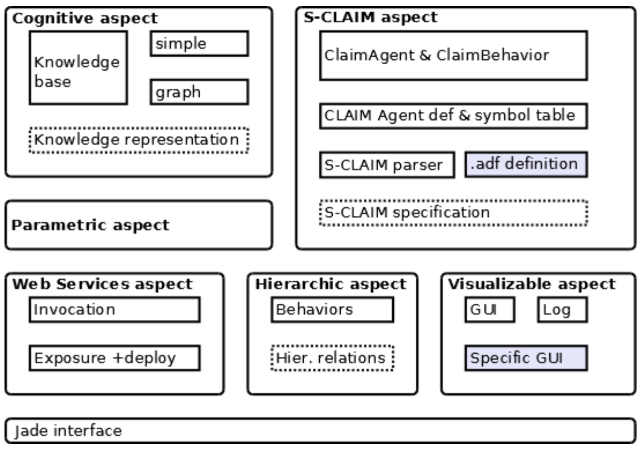


Fig. 1. The structure of the tATAmI agent. The dotted frames indicate specifications, not actually implemented parts. Highlighted frames indicate parts that need to be defined by the user for each specific scenario.

important effort in making the project usable by other members of the MAS community, so maintenance and documentation are current priorities.

4.1 The tATAmI Agent

The tATAmI agent is an extension of the Jade agent. Therefore it inherits all the features offered by Jade: the possibility to run behaviors, the communication features, and the mobility.

Moreover, the tATAmI agent adds several aspects (Figure 1) that add important features over the usual Jade agent, making it a powerful tool in the hands of the agent designer. The features added are improved, cross-platform, visualization, centralized logging, hierarchical movement, web service access and exposure, a knowledge base and last, but not least, a simple way to program the agents using the S-CLAIM language.

The **link with Jade** is assured by the fact that the tATAmI agent class extends Jade's `GuiAgent`. Some other Jade-related functions are accessed through an interface that abstracts Jade itself, so that the project can switch to a different underlying agent platform if wanted.

The **Parametric aspect** assures the user of the platform that all that can be parameterized in the agent can be done so from the scenario file.

The **Visualizable aspect** of the agent covers several features related to the easiness of tracing the agent's activity. First, it manages the agent's log, which is a classic logger. However, the log is configured so that it prints its output in an area of the agent's GUI, and also sends the logging information to the central Visualization agent, that gathers all logging information and sorts it according to timestamps. Second, this aspect of the agent also automatically starts the appropriate GUI for the agent, depending on the settings in the scenario file. The programmer is able to specify the name of the agent's GUI and the correct platform-dependent implementation will be instantiated when the agent starts on or moves to a specify platform.

The **Web Services aspect** has two roles, both assured by the Jade add-ons WSIG and WSDC. First, it allows the agent to access web services and use their output, in a way that is almost identical to communicating with other agents. Second, it exposes all of the S-CLAIM agent's behaviors as web services, so that they can be activated by, and receive messages from, other components that are not implemented using Jade.

The **Hierarchical aspect** enables the creation of logical hierarchies between agents, executing on the same machine or spread across several ones. These hierarchies are primarily used to create a partial representation of context and to allow agents to communicate only within their context. Hierarchical movement is also an important feature: except for agents that are explicitly fixed to their machines (through a parameter in the scenario), when a parent agent moves to a different machine, all its sub-hierarchy of child agents follows it to that machine, allowing agents that offer sub-functionalities to remain together (see Figure 2).

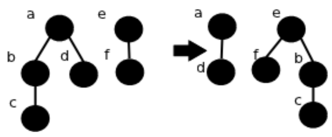


Fig. 2. Example of hierarchical migration: agents *b* migrates as a child of *e*, and its child *c* follows (potentially to another machine). As agents are hierarchized by context, it makes sense for a sub-context to move together with its parent context.

The **Cognitive aspect** of the agent means that the agent holds a knowledge base. The type of the knowledge base need not be the same for all agents in a scenario, as any knowledge base implementation must be accessible through a standard interface offering three functionalities: adding knowledge, removing knowledge, and searching for knowledge that matches a certain pattern (see Figure 5).

The **S-CLAIM aspect** of the tATAmI agent is its most important aspect, and the top-most one, as it uses all of the other aspects of the agent. An agent implemented using the S-CLAIM language is based on an `.adf2` – agent definition – file that is given to the agent as a parameter, and that is parsed into an agent definition containing behavior definitions. The latter are transformed into Jade behaviors. Besides the algorithmic part, the S-CLAIM aspect also interacts (if the `adf` code specifies it) with the Web Services aspect (the code can specify the invocation of web services or the response to web service invocations), with

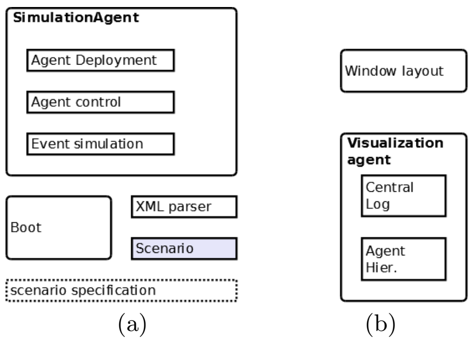


Fig. 3. The structure of the Simulation (a) and Visualization (b) components. The dotted frame indicates specifications, not actually implemented parts. Highlighted frames indicate parts that need to be defined by the user for each specific scenario.

the Hierarchical aspect (by allowing an agent to move or to become a child of another), and with the Visualizable aspect (by allowing the code to use components of the GUI for input and output).

4.2 Simulation

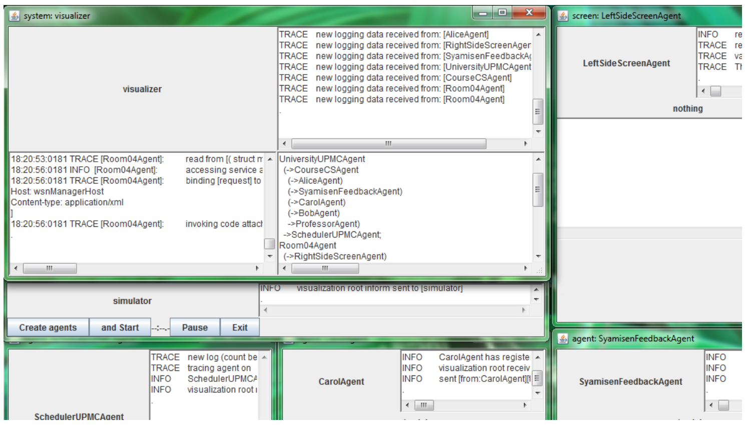
The simulation component of the platform is meant to allow the user of the platform or the agent designer to perform a series of operations quickly and effortlessly:

– start the platform and deploy the agents with one click;
– run the same predefined scenario multiple times easily;
– close the platform with one click.

In order to do this, there are several parts of the Simulation component. A simulation is based on an XML scenario file. That file contains all the information for configuring the Jade platform, for creating containers and agents, and on the parameters of each agent. It also contains a timeline of "events", messages which are sent by the Simulation Agent to other agents in the scenario, to simulate external perceptions. An example is presented in Section 6.

The tATAmI project provides a `Boot` class that can be called by a user who wants to run the platform. The class takes arguments related to scenario and other parameters. The class then loads the scenario and creates the Simulation and Visualization agents.

The **Simulation agent** creates (on user command) all agents in the scenario and informs them on whom to send the logging messages (i.e. to the Visualization agent). It also handles the events on the simulation timeline. Moreover, it is able to inform all agents about the fact that the platform is shutting down. This assures a clean shut down of the platform.



```
TRACE   new logging data received from: [AliceAgent]
TRACE   new logging data received from: [RightSideScreenAgen
TRACE   new logging data received from: [SyamisenFeedbackA
TRACE   new logging data received from: [UniversityUPMCAgent
TRACE   new logging data received from: [CourseCSAgent]
TRACE   new logging data received from: [Room04Agent]
TRACE   new logging data received from: [Room04Agent]
.
```

```
18:20:53:0181 TRACE [Room04Agent]:      read from [( struct m
18:20:56:0181 INFO  [Room04Agent]:      accessing service a
18:20:56:0181 TRACE [Room04Agent]:      binding [request] to
Host: wsnManagerHost
Content-type: application/xml
]
18:20:56:0181 TRACE [Room04Agent]:      invoking code attach
.
```

```
UniversityUPMCAgent
 (->CourseCSAgent
  (->AliceAgent)
  (->SyamisenFeedbackAgent)
  (->CarolAgent)
  (->BobAgent)
  ->ProfessorAgent)
 ->SchedulerUPMCAgent;
Room04Agent
 (->RightSideScreenAgent)
```

```
INFO    visualization root inform sent to [simulator]
.
```

```
INFO    re
TRACE   re
TRACE   va
TRACE   Th
.
```

```
TRACE   new log (count be
TRACE   tracing agent on
INFO    SchedulerUPMCA
INFO    visualization root i
.
```

```
INFO    CarolAgent has registe
INFO    visualization root receiv
INFO    sent [from:CarolAgent][t
.
```

```
INFO
INFO
INFO
.
```

Fig. 4. A screenshot of the platform's execution. The Visualizer agent gathers logging messages and displays the agent hierarchy in text format; the Simulation agent allows the control of agents; all agent GUIs are automatically arranged for easy access.

4.3 Visualization

While working with a multi-agent system deployed across multiple machines, it is important to be able to visualize how the system works, from a central point. This is what the Visualization component does, together with the Visualizable aspect of the tATAmI agent.

Besides the functionalities described in Section 4.1, under the Visualizable aspect, the visualization process uses a Visualization agent (one per platform, but extendable to multiple agents per platform) and the Window Layout.

The **Visualization agent** receives logging messages from all agents in the platform (including special agents like the Simulation agent), sorts them according to their timestamp, and displays them in its window. This is essential for following the activity of the system from a single point. The agent also displays information about the logical hierarchy of the agents in the system (see the agent's Hierarchical aspect) (see Figure 4).

The **Window Layout** class is a small utility class that, based on some simple layout indications, gives to every new window a space on the screen so that it does not overlap other windows. Windows have sizes that are also related to the indications, for instance making the GUI of the Visualization agent larger than other agents'. This way, when the user starts the simulation, there is no need to reposition GUI windows by hand, making working with the platform even simpler.

5 Implementation details

Implementing the tATAmI platform meant solving some challenges that are common to many agent-based platforms. Here we give details on some of those parts of the implementation.

5.1 S-CLAIM

It all begins with the tATAmI platform parsing both an XML file (using a parser derived from SAX - Simple API for XML) containing the description of the scenario that needs to be executed (this scenario file is received as argument when the platform starts) and the agent definition files (which are specified in the scenario), that are translated into some structures (using a LALR(1) parser generated using JFlex and BYACC/J, based on the vocabulary and syntax of S-CLAIM) that contain complete descriptions of the agents and of their behaviors. These structures are then passed to some newly created Jade agents, together with the values of their parameters (extracted from the scenario), so that they could instatiate themselves. From now on we can talk about S-CLAIM agents.

In what concerns the behavior component of an S-CLAIM agent, we used the Jade *CyclicBehaviour*s for the *reactive* behaviors (which always wait for new messages to trigger them), and *OneShotBehaviour*s for *initial* behaviors (which are executed only one time, when the agent is created). Similar to the reactive behaviors are the *proactive* behavior and the *cyclic* one, who also have a permanent character. So, they are based on Jade's *CyclicBehavior* too.

For the messages, we implemented a new class that extends the Jade message class. The agents can exchange Strings or Objects, such as a structure described by the *struct* keyword in S-CLAIM. The S-CLAIM messages have a specific ontology enabling agents to do pattern matching in order to verify if the messages they receive match the structures of the messages the reactive behaviors wait for.

We implemented the mobility of the agents with respect to the hierarchical structure of the systems implemented in S-CLAIM. The mobility primitives - *in, out* - and the agent management primitives - *open, acid, new* - are implemented by using some available Jade libraries plus some hierarchical actions, as described above.

5.2 Making tATAmI Cross-Platform

As the tATAmI interacts with various features like visual interfaces (for providing the agent GUI), web services, and logging tools, a tATAmI agent works seamlessly between Linux and Windows platforms, but cannot be directly deployed on, or move to, an Android device. However, our design choice was to make tATAmI as portable as possible, that is, have as much code as possible portable to Android.

The implementation uses several interfaces for components that need to be implemented in a different way on various platforms. The most important are the agent's GUI and the logging tool. The agent GUI is obviously not portable, as the paradigms for the GUI are different on PCs and Android devices. The logging tool needed to be abstractized by an interface because for the PC we have chosen a tool that is not supported on Android, so we'll use another logging tool while the agent is on an Android device. Both logging and the GUI are part of the Visualizable aspect of the agent.

Parts of the agent are initialized at creation and reinitialized after every move to another machine. When the Visualizable aspect of the agent is reinitialized, an external class is invoked that tells the agent what logging tool and what GUI paradigm to use on the current platform. Then, the appropriate class is instantiated dynamically. All classes implementing a certain functionality implement the same interface, so that they will be invoked in the same way regardless of the platform.

5.3 tATAmI Agents on the Android Platform

As we saw in Section 5.2, we wanted to have a core of tATAmI, common to both computers supporting Java and Android devices and we wanted this core to contain as much of the tATAmI code as possible. However, it wasn't possible to deploy the whole tATAmI platform to Android devices without having to replace some parts or even to renounce to some others. First of all, we needed to give a role to the Android devices and we decided that they will only be used as clients, as there was no reason of wanting to host a tATAmI platform on such a device, considering its limited resources. This also came together with the strong link between our platform and the Jade framework. There was already a very useful add-on for Jade, named *JadeLeapAndroid*, that allowed users to create containers on the Android devices, that, connected to a Jade platform on a PC, made the execution of Jade agents possible in the same manner as for PCs. Thus, Jade offered us much of the features we needed, so we decided to use them as they were. So, there was no need for the simulation and the visualization components and all the aspects related to them (see Section 4) on Android. All that we needed was a *Connection* component, enabling the device to connect to a running tATAmI platform, when desired. After the connection with the platform was established, the only component needed was the *Agent* one (4.1).

With a few exceptions, the agents could run on the Android devices just as they could on a PC. The exceptions are, as shown in Section 5.2, the agent's GUI and the logging. One other exception is the *web services aspect* (Section 4.1), because the WSIG and WSDC add-ons of Jade are not supported for Android. However, as web services are an important aspect of the tATAmI agent, one of our highest priorities is to enable them for the agents running on Android too.

Concerning the logging aspect, the main problem was that we wanted to use a powerful tool as *Apache log4j*, which is not supported for Android. Therefore, based on a common interface, we created, first, a wrapper for the Jade logger. We soon realized there were some compatibility issues between the Android and PC versions of the Jade logger, so, currently, agents running on Android use a

Java logger wrapper and when they pass to a PC or the agents running on such devices use a logger based on Apache log4j, that offers some extra features, of which we mostly use the formatting ones.

The last particularity we had to consider in order to adapt tATAmI for Android were the different concepts the two platforms use with respect to the graphical user interfaces. So we needed to do a complete separation of the tATAmI agent and of the GUI (together with the mechanism behind it). So, the details about how to create the GUI and how to offer it different functionalities are left to programmers specialized in doing this, who have to implement a Java interface that is connected to the tATAmI agent by means of the S-CLAIM *input* and *output* primitives. All the S-CLAIM programmer has to have in mind are the names of the GUI components and the values they receive as input or they wish to output. If the programmer wants to define his own GUIs for some of the agents (as opposed to using the default one - or of some other predefined GUI types, in the future) that are mobile and supposed to execute on both platforms, he has to define a GUI for each of the two platforms, respecting the same naming conventions for their components.

6 Scenario implementation

In this section we provide a detailed description of how to implement a scenario in tATAmI, using S-CLAIM.

6.1 The SmartRoom scenario

An example scenario to be implemented in order to prove the ease of use of tATAmI is *SmartRoom*, also described in [1] as follows:

Alice is a student at the university. Today, the Multi-Agent Systems (MAS) course is held in a room other than usual. All the students of this class are notified automatically via their smartphones about this change and receive an indication on how to get to the new room. Alice is the first one who arrives. While she enters, the lights are automatically turned on and the main screen shows a welcome message. When it is time to start the course, observing that the professor and all the students are in the room, the lights dim and the main screen shows the first slide of the presentation. When the professor indicates that the presentation has finished, the lights turn on again to start the second section of the course: brainstorming. The class is divided into several groups. Each group has a large smart screen to display their opinions. Students write their opinions on their smartphone or laptop. The opinions appear right away on the screen associated to the group, so that the others could see them. When Alice moves to another group to discuss, her opinions are automatically displayed on the screen of the new group, and removed from the other one.

6.2 Implementation of the SmartRoom scenario

We would like to use the *SmartRoom* scenario described in Section 6.1 as a starting point in order to show, in this section, what implementing a scenario using tATAmI means.

We decided to simplify SmartRoom in order to offer readers a glimpse of how to implement and execute a short scenario. Let us consider that there are three students enrolled in the MAS course, *Alice*, *Bob* and *Carol.*Each one of them is supposed to have his/her own device, running the SmartRoom application. Each student has an agent associated, running on his/her behalf. The agents are named *AliceAgent*, *BobAgent* and *CarolAgent* and they are of the same class, named *StudentAgent.* StudentAgent describes a class of agents that for our example are as simple as the definition in Figure 5.

```
(agent Student ?userName
  (behavior
    (initial register
      (send parent (struct message assistsUser ?userName))
    )

    (reactive courseScheduling
      (receive (message scheduling ?courseName ?roomName ?roomAgentName))
      (addK (struct knowledge scheduling ?courseName ?roomName))
      (addK (struct knowledge roomAgent ?roomName ?roomAgentName))
    )
  )
  ...
)
```

Fig. 5. *StudentAgent* class, defined in the file *StudentAgent.adf2*

```
<scen:agent>
    <scen:parameter name="loader" value="adf2" />
    <scen:parameter name="class" value="StudentAgent" />
    <scen:parameter name="name" value="AliceAgent" />
    <scen:parameter name="parent" value="MASCourseAgent" />
    <scen:parameter name="userName" value="Alice" />
    <scen:parameter name="fixed" value="true" />
    <scen:parameter name="GUI" value="UserAgentGUI" />
</scen:agent>
```

Fig. 6. Declaration of AliceAgent in the scenario XML file

An agent of the class *StudentAgent* registers with its parent, *MASCourseAgent*, when it is created (using the *register* initial behavior), then it waits for messages informing it about changes in the scheduling of the courses and, when it receives them, it updates its knowledge base (for reasons of keeping the example simple, one update consists only of the agent adding the new knowledge to its knowledge base. In a real case, however, the programmer might want to verify first if the new

```
1 <scen:jadeConfig mainContainerName="Administration"
2       platformID="SmartRoom" />
3 <scen:adfPath>scenario/SmartRoom</scen:adfPath>
4 <scen:agentPackage>agent_packages.smartRoom</scen:agentPackage>
```

Fig. 7. Jade configuration, path to adf2 files and Java package used by the scenario

knowledge is in conflict with other knowledge records and he might also want to solve this conflict by, for example, deleting the old knowledge record from the knowledge base, in order to leave only the new one, which is supposed to be better) according to the information contained in these messages, so that further behaviors could use the knowledge to adapt to the changes and, this way, to help the student. The name of the student such an agent assists is given as parameter in the scenario's XML file and stored as the value of the *?userName* variable. Let's look at the declaration of *AliceAgent* in the scenario file (Figure 6). Beside the name of the user it assists, Alice, and its name, we can see that *AliceAgent* also receives some other parameters. One is the name of the class, which is used, together with the loader type (currently only *adf2* is supported) and with the path to the agent definition files (Figure 7, line 3), to find the agent class definition in order to parse it into a *tatami.core.claim.parser.ClaimAgentDefinition* Java object. Another parameter is the name of its parent. It is stored as value of an implicit variable *?parent* that can also be referred using the S-CLAIM keyword *parent*. Then, we have an optional parameter named *GUI*, specifying the name of the Java class implementing the *tatami.core.interfaces.AgentGui* interface, located in the agent package associated with the scenario (Figure 7, line 4). This class is defined for each different platform, in our case for both PC and Android (in this case, however, it's not very different from the default agent GUI, a simple window with the name of the agent and which prints the log of the agent). And the last parameter, *fixed* set by default to *false* and, in our case, set explicitly to *true* (Figure 6, line 7) specifies wether the agent will physically move together with its parent (namely MASCourseAgent) or if it stays in its container when the parent moves. The declarations of *BobAgent* and of *CarolAgent* are similar to the one of *AliceAgent* except for the fact that, for this example, we chose not to make them "fixed", although, normally, for such a scenario these agents should be sticked to the devices used by the users they represent.

Furthermore, our example scenario contains two more agents, *UniversityUPMCAgent* and *SchedulerUPMCAgent*, that, together with MASCourseAgent (referred above and defined by the agent class in Figure 8), are deployed to a container named *Administration*, which is also the main container of the Jade platform associated to the scenario (Figure 9). Another container is associated to *Room04*. It is supposed to be running on a machine available in this room, the new room for the MAS course (Figure 10), and it contains, at the beginning of the execution of the scenario, the *Room04Agent* agent. The roles of SchedulerUPMCAgent, UniversityUPMCAgent and Room04Agent are simple in this example. The first is supposed to forward messages of the type *scheduledTo*

```
1  (agent Course ?courseName
2    (behavior
3      ... //"register" (initial) and "registerUser" (reactive) behaviors
4
5      (reactive changeRoom
6        (receive (message scheduling ?courseName ?roomName))
7        (addK (struct knowledge scheduling ?courseName ?roomName))
8        (if (readK (struct knowledge roomAgent ?roomName ?roomAgentName))
9        then
10         (forAllK (struct knowledge userAgent ??userName ??userAgentName)
11             (send ??userAgentName (struct message scheduling ?courseName
12               ?roomName ?roomAgentName))
13         )
14         (in ?roomAgentName)
15       else
16         (send parent (struct message whoManagesRoom this ?roomName))
17       )
18     )
19
20     (reactive changeRoom2
21       (receive (message managesRoom ?roomAgentName ?roomName))
22       (condition (readK (struct knowledge scheduling ?courseName
23                  ?roomName)))
24       (addK (struct knowledge roomAgent ?roomName ?roomAgentName))
25       ... //HERE, the code from the lines 10-14, copied!
26     )
27   )
28 )
```

Fig. 8. *CourseAgent* class, defined in the file *CourseAgent.adf2*

?courseName ?roomName to its parent, UniversityUPMCAgent, the second is supposed to act as the manager of the whole university and its main role in our example is to inform MASCourseAgent about the change of the room, when he receives this information from the scheduler agent, and the third is supposed to manage the room it represents.

We saw two declarations of containers in Figure 9 and 10, which are containers that, in our example, run on the same machine as the tATAmI platform. However, there are cases in which we would prefer not to automatically create the containers on the same machine as the platform, like the cases of the containers supposed to host the agents assisting users, who should execute on their machines. For this particular case, we can specify that the container should not be automatically created, like in the example below, which corresponds to the container that will run on Alice's Android phone:

```
<scen:container name="AliceContainer" create="false">
...
</scen:container>
```

In this case, we can specify that the container should not be created automatically, by setting the value of the *create* argument in a container declaration to "false".

It's time, now, to run our scenario. As we saw before, we have the students, Alice, Bob and Carol, attending the MAS course. Thus, their agents are children of MASCourseAgent. At a certain moment the room allocated by the university

```
<scen:container name="Administration">
    <scen:agent>
        <scen:parameter name="loader" value="adf2" />
        <scen:parameter name="class" value="UniversityAgent" />
        <scen:parameter name="name" value="UniversityUPMCAgent" />
    </scen:agent>
    <scen:agent>
        <scen:parameter name="loader" value="adf2" />
        <scen:parameter name="class" value="SchedulerAgent" />
        <scen:parameter name="name" value="SchedulerUPMCAgent" />
        <scen:parameter name="parent" value="UniversityUPMCAgent" />
    </scen:agent>
    <scen:agent>
        <scen:parameter name="loader" value="adf2" />
        <scen:parameter name="class" value="CourseAgent" />
        <scen:parameter name="name" value="MASCourseAgent" />
        <scen:parameter name="parent" value="UniversityUPMCAgent" />
        <scen:parameter name="courseName" value="CSCourse" />
    </scen:agent>
</scen:container>
```

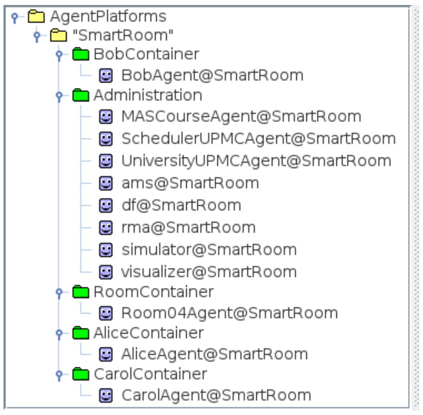
Fig. 9. Declaration of *Administration* container

```
<scen:container name="RoomContainer">
    <scen:agent>
        <scen:parameter name="loader" value="adf2" />
        <scen:parameter name="class" value="RoomAgent" />
        <scen:parameter name="name" value="Room04Agent" />
        <scen:parameter name="parent" value="UniversityUPMCAgent" />
        <scen:parameter name="roomName" value="Room04" />
    </scen:agent>
</scen:container>
```

Fig. 10. Declaration of *RoomContainer* container

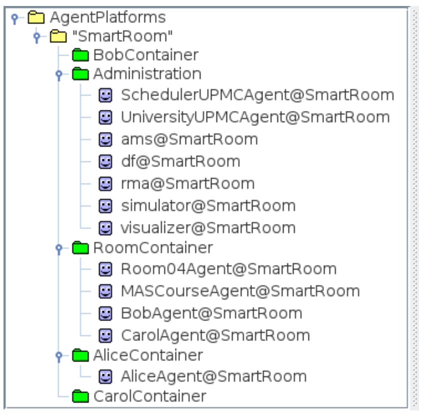
```
  <scen:timeline>
    <scen:event time="2000" >
      <scen:CLAIMMessage>
        <scen:to>SchedulerUPMCAgent</scen:to>
        <scen:protocol>newSchedule</scen:protocol>
        <scen:content>
            ( struct message newSchedule ( struct knowledge
                scheduledTo CSCourse Room04 ) )
        </scen:content>
      </scen:CLAIMMessage></scen:event>
  </scen:timeline>
```

Fig. 11. Timeline of the scenario, generating a message at the time moment 2000



```
AgentPlatforms
  "SmartRoom"
    BobContainer
      BobAgent@SmartRoom
    Administration
      MASCourseAgent@SmartRoom
      SchedulerUPMCAgent@SmartRoom
      UniversityUPMCAgent@SmartRoom
      ams@SmartRoom
      df@SmartRoom
      rma@SmartRoom
      simulator@SmartRoom
      visualizer@SmartRoom
    RoomContainer
      Room04Agent@SmartRoom
    AliceContainer
      AliceAgent@SmartRoom
    CarolContainer
      CarolAgent@SmartRoom
```

(a) Jade RMA - before



```
AgentPlatforms
  "SmartRoom"
    BobContainer
    Administration
      SchedulerUPMCAgent@SmartRoom
      UniversityUPMCAgent@SmartRoom
      ams@SmartRoom
      df@SmartRoom
      rma@SmartRoom
      simulator@SmartRoom
      visualizer@SmartRoom
    RoomContainer
      Room04Agent@SmartRoom
      MASCourseAgent@SmartRoom
      BobAgent@SmartRoom
      CarolAgent@SmartRoom
    AliceContainer
      AliceAgent@SmartRoom
    CarolContainer
```

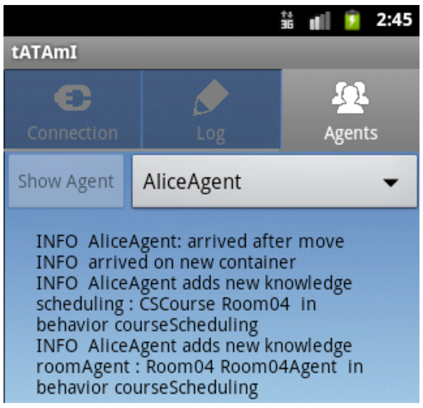
(b) Jade RMA - after

```
MASCourseAgent->AliceAgent
UniversityUPMCAgent->SchedulerUPMCAgent
MASCourseAgent->CarolAgent
MASCourseAgent->BobAgent
UniversityUPMCAgent->MASCourseAgent
UniversityUPMCAgent->Room04Agent
```

(c) tATAmI Visualizer - before

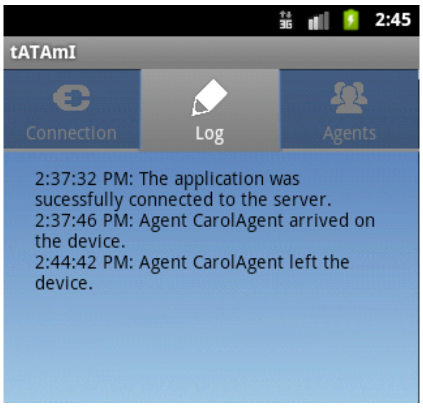
```
MASCourseAgent->AliceAgent
UniversityUPMCAgent->SchedulerUPMCAgent
MASCourseAgent->CarolAgent
MASCourseAgent->BobAgent
Room04Agent->MASCourseAgent
UniversityUPMCAgent->Room04Agent
```

(d) tATAmI Visualizer - after



```
INFO  AliceAgent: arrived after move
INFO  arrived on new container
INFO  AliceAgent adds new knowledge
scheduling : CSCourse Room04  in
behavior courseScheduling
INFO  AliceAgent adds new knowledge
roomAgent : Room04 Room04Agent  in
behavior courseScheduling
```

(e) Alice's device - after



```
2:37:32 PM: The application was
sucessfully connected to the server.
2:37:46 PM: Agent CarolAgent arrived on
the device.
2:44:42 PM: Agent CarolAgent left the
device.
```

(f) Carol's device - after

Fig. 12. Scenario execution (before = before execution, after = after execution)

for this course changes. In the real world this could be seen as an administrator of the university changing a record in the university's schedule, that is connected to a behavior of the SchedulerUPMCAgent. But, simpler, it is possible model this by means of a time event, defined in the timeline of the scenario's XML file, as in Figure 11. So, at the time moment 2000 (considered in milliseconds) a message is sent to the scheduler agent, informing it about the change. Further, SchedulerUPMCAgent informs its parent, UniversityUPMCAgent about this, who, then, announces MASCourseAgent that there is a new room for the course. As we can see in Figure 8, the behavior *changeRoom* is triggered by this message and MASCourseAgent informs all its children about this modification (children who registered with their parent at creation so that MASCourseAgent stored information about them in its KB, using the *registerUser* reactive behavior) then it moves with all its subhierarchy, becoming a child of the agent managing the room (the statement *in ?roomAgentName*), if it already knows its name. Otherwise, it will ask its parent about the name of the agent managing the room and will do the same thing after receiving an answer, by means of the *changeRoom2* behavior. MASCourseAgent and the agents assisting students, except for AliceAgent, will also move physically to the container of Room04Agent, because we wanted them not to be fixed.

In order to illustrate the example above, we provide Figure 12. Here we can see in the subfigures 12(a) and 12(c) an overview of the agents in the system, both physically (as seen by Jade's RMA agent) and logically, before the execution of the scenario, just after the creation of the agents. Then, after the execution, we can see how MASCourseAgent, BobAgent and CarolAgent all moved in RoomContainer (Figure 12(b)) and how MASCourseAgent changed its position in the logical hierarchy of agents, becoming a child of Room04Agent instead of UniversityUPMCAgent. After this, the subfigures 12(e) and 12(f) show us a comparative view of the Android devices of Alice and Carol after the execution of the scenario. We remember that AliceAgent was the only StudentAgent who had the "fixed" parameter set to "true", so we can see that AliceAgent is still on her Android device, offering assistance to her.

6.3 Other examples

PDAagent - Assigning a screen Another code example is the one from Figure 13, also presented in [1]. In this example the user has to display some opinions, from his/her PDA, on a screen. The agent that assists the user, running on the PDA, has the class PDAagent. The agent has only *initial* and *reactive* behaviors. The initial behavior *register* informs the parent of the agent, at creation, about the fact that it assists the user denoted by the variable *?userName*. The parent's name, as well as the name of the user to assist, are received as arguments, after they were read from the scenario's XML file (not presented here). The reactive behavior *assignScreen* is triggered when receiving a message of type *screenAssigned*, containing a variable. When triggered, it is verified if the agent already has a screen assigned. If *true*, it is verified if the old screen and the new one are different. If positive, the agent managing the old screen is informed to

```
1  (agent PDAagent ?userName ?parent
2    (behavior
3      (initial register
4        (send ?parent (struct message assistsUser this ?userName))
5      )
6      ....
7      (reactive assignScreen
8        (receive screenAssigned ?screenAgentName)
9        (if (readK (struct knowledge useScreen ?oldscreenAgentName))
10       then
11         (if (isDifferent ?oldscreenAgentName ?screenAgentName)
12         then
13           (send ?oldscreenAgentName (struct message removeUser ?userName))
14           (removeK (struct knowledge useScreen ?oldScreenAgentName))
15           (in ?screenAgentName)
16           (addK (struct knowledge useScreen ?screenAgentName))
17           (readK (struct knowledge opinionType ?type))
18           (forAllK (struct knowledge opinion ?opinion)
19             (send ?screenAgentName (struct message opinionList ?type
20               ?userName ?opinion))
21           )
22         )
23       else
24         (in ?screenAgentName)
25         (addK (struct knowledge useScreen ?screenAgentName))
26         (readK (struct knowledge opinionType ?type))
27         (forAllK (struct knowledge opinion ?opinion)
28           (send ?screenAgentName (struct message opinionList
29             ?type ?userName ?opinion))
30         )
31       )
32     )
33   )
34 )
```

Fig. 13. Another example of an agent class definition, for a class called PDAagent

remove the rights of the user to display information on the screen. The old screen is also removed from the knowledge base of the agent. Then it moves to the sub-hierarchy of the agent managing the new screen. It also stores the information about the new screen in the knowledge base. All the opinions, together with their type, are sent to the agent managing the new screen, so they would be displayed. If no screen was previously assigned, a sequence of code identical with the one found at the lines 15-20 is executed.

Note that in S-CLAIM, once variables are bound inside a context (agent, behavior, block), they keep their value until the end of their context. Any further test on the variable will consider it as bound, therefore as a restriction to a pattern. Moreover, the *isDifferent* function, that is used at line 11, is a Java function from an external library. Note that it is used exactly the same as with any S-CLAIM construct.

Ant colony optimization This example concerns an ant colony optimization (ACO) example, in which the ants have to find all the food locations from a given map (i.e. graph) and to transport as much food as possible to the ant hill, using coordination strategies as well. An illustration of this scenario is offered in Figure 14 and the agentification (for the same problem, but for a different graph) in Figure 15.

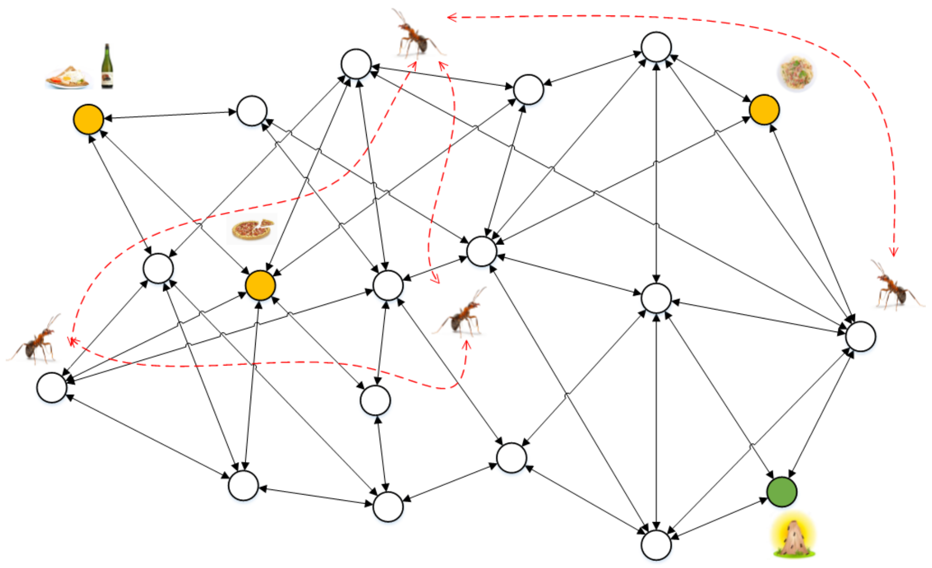


Fig. 14. Illustration of the ACO scenario, with coordination; circles – locations; yellow – food locations; green – ant hill location; dashed line – coordination through messages; continuous lines – paths between locations.

For the agentification from Figure 15 we see that the ants are first created by the ant hill (i.e. the agent managing the ant hill, but, as there is no risk of confusions, we simply call it "the ant hill" – similar references are used for the other agents too), at the beginning of the simulation, so they become children of this agent. Then, the ants exit the subhierarchy of the ant hill, using the "out" mobility primitive, and start an iteration, exploring the available locations according to the graph received as input at the beginning of the simulation. Then, when they find food locations, they enter their hierarchies and collect the food, leaving them after this in order to return to the ant hill. All the exploration, food collection and food deposit at the ant hill are made possible by the *in* and *out* primitives. At the end and at the beginning of each iteration all the ants are in the subhierarchy of the ant hill, just like at the beginning of the simulation.

Considering the discussion above, we can see in Figure 16 a snippet of the code of the agent class *AntHill*. The initial behavior is used in order to register with its parent and with the Manager agent (whose name is received as parameter). Later, when the manager agent processes the registration message, it will send an message to the ant hill informing it about some important parameters, that Manager read from a file at the beginning of a simulation with the name received as parameter as well. Then we see the *receiveParamsAndCreateAnts*

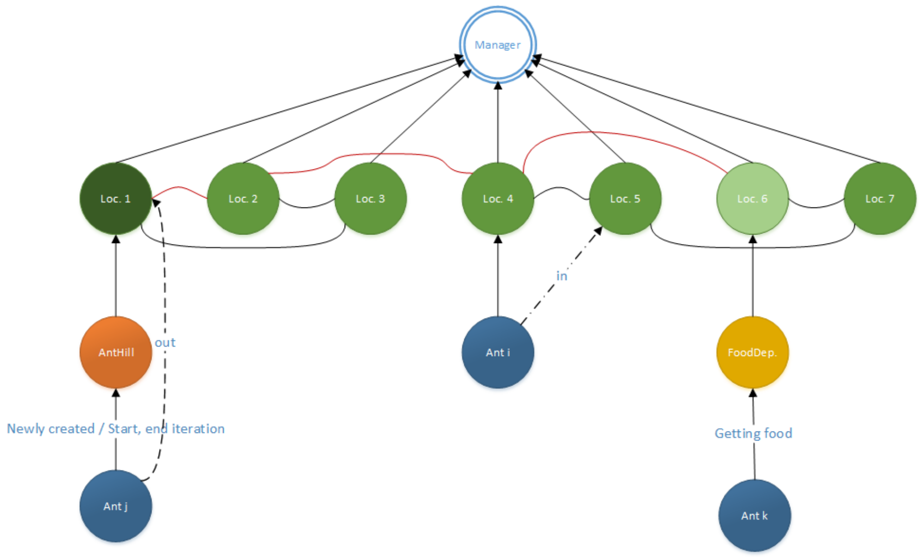


Fig. 15. Agentification of the ACO scenario; circles – agents; yellow – food deposit manager; green – location managers; blue – ants; orange – ant hill manager; uppermost agent – the manager of the whole scenario; arrows – hierarchical relation; curved lines - the paths between the locations.

```
(agent AntHill ?managerAgent

  (behavior
    (initial register
      (send parent (message managesAntHill))
      (send ?managerAgent (message managesAntHill))
    )
    /*...*/
    (reactive receiveParamsAndCreateAnts
      (receive (message informParams ??noAnts ?alpha ?beta))
      (addK (struct knowledge noAnts ??noAnts))
      (addK (struct knowledge alpha ?alpha))
      (addK (struct knowledge beta ?beta))

      //Creating ??noAnts ants
      (while (greater ??noAnts 0)
        (concatenate Ant ??noAnts ??currentAntName)
        (new Ant ??currentAntName)
        (decrement ??noAnts ??noAnts)
      )
    )
    /*...*/
  )
)
```

Fig. 16. *AntHill* class, defined in the file *AntHill.adf2*

behavior, in which AntHillAgent receives three parameters, stores them in the knowledge base, then it creates, based on one of them (noAnts), a certain number of Ant agents (agents of class Ant). The call to the *new* agent management primitive receives as arguments the name of the class (i.e. the constant *Ant*) and a variable storing the name the new ant will have (computed using two Java functions from the agent's function library, *concatenate* – which stores in the third argument the result of the concatenation of the first two – and, later, *decrement* – which decrements *noAnts* by one). We observe here the notation *??*, which was later introduced in S-CLAIM in order to denote "affectable variables", which are variables that can change their values in the same scope life cycle, once bounded, being a useful construct in the case of the loops, like the *while* loop in our example.

In order for an agent of the type AntHill to be able to create an agent of class Ant, the Ant class has to be parsed as well, even if no agent of this class is available in the scenario in its initial state. For this purpose we use, in the scenario XML file, a construct called "subsequent", in which we declare all the agent classes of the agents that will be created during the simulation (as oposed to the agents created at the beginning of the simulation). This way, the intermediate representations (i.e. the Java objects returned by the parser) of the classes declared here can be atta ched to the agent classes, from the initial configuration, creating such agents, making use of the parametric aspect (see Section 4.1). For our scenario this is illustrated in Figure 17, where the containers and the agents declared inside the *initial* tag are part of the initial configuration of the scenario, while the *Ant* class is declared inside the *subsequent* tag.

In Figure 18 we see a part of the code of the Ant class, namely the one that illustrates how an ant chooses the following locations in search for food (and before returning). All starts with a message asking the ant to start seeking food, at the beginning of the iteration, in the *startSeekingFood* behavior. After receiving this message, the ant exists the subhierarchy of the ant hill, being now a child of the *Location01* agent, as stated in the scenario XML file (see Figure 17). Subsequently, the agent initializes the list of the visited locations, during this iteration. Then, a cyclic behavior controlled by two condition primitives that return a true value now, makes the ant ask its parent (being a location, as we saw later) about the neighbouring locations and about so other information, like the values of the pheromone trails or the distances to its neighbours. After this, in the *receiveNeighborsAndInfo* reactive behavior, the ant receives the information requested earlier and it is able to compute the next location, that will be stored in the *?nextLocation* variable. The code stops here in our figure, but, following this step, the ant asks its parent about the name of the agent managing the computed next location, so that it can, when receiving it, move to its subhierarchy using the *in* primitive. If no valid location can be computed from the current location of the ant, it will enter into a "returning mode", in which the ant follows the same path to return to the ant hill and offers the locations it visits information about how to update the pheromone trails, based on the result it obtained during

```xml
<?xml version="1.0" encoding="UTF-8"?>
<scen:scenario xmlns:pr="http://www.example.org/parameterSchema"
  xmlns:kb="http://www.example.org/kbSchema"
    xmlns:scen="http://www.example.org/scenarioSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.example.org/scenarioSchema
␣␣␣␣␣../../../config/scenarioSchema2.xsd">

  <scen:jadeConfig isMain="true" />
  <scen:adfPath>scenario</scen:adfPath>
  <scen:agentPackage>agent\_packages.aco</scen:agentPackage>

  <scen:initial>
      <!-- Manager container and agent -->
      <scen:container name="ManagerContainer">
        <scen:agent>
        <scen:parameter name="loader" value="adf2" />
        <scen:parameter name="class" value="Manager" />
        <scen:parameter name="name" value="Manager" />
        <scen:parameter name="GUI" value="ManagerGui" />
        <scen:parameter name="java-code" value="ManagerFunctions" />
        <scen:parameter name="resourcePath" value="resources/" />
        <scen:parameter name="parametersFile" value="parameters.txt" />
        <scen:parameter name="graphFile" value="graph.txt" />
        <scen:parameter name="distancesFile" value="distances.txt" />
        <scen:parameter name="poiFile" value="pointsOfInterest.txt" />
      </scen:agent>
    </scen:container>

      <!-- Location containers and agents -->
      <scen:container name="C01">
          <!-- Location 1, containing the ant hill -->
        <scen:agent>
        <scen:parameter name="loader" value="adf2" />
        <scen:parameter name="class" value="Location" />
        <scen:parameter name="name" value="Location01" />
        <scen:parameter name="GUI" value="LocationGui" />
        <scen:parameter name="java-code" value="LocationFunctions" />
        <scen:parameter name="location" value="C01" />
        <scen:parameter name="fixed" value="true" />
        <scen:parameter name="parent" value="Manager" />
      </scen:agent>

        <!-- The agent representing the ant hill -->
        <scen:agent>
        <scen:parameter name="loader" value="adf2" />
        <scen:parameter name="class" value="AntHill" />
        <scen:parameter name="name" value="AntHillAgent" />
        <scen:parameter name="java-code" value="AntHillFunctions" />
        <scen:parameter name="fixed" value="true" />
        <scen:parameter name="parent" value="Location01" />
        <scen:parameter name="managerAgent" value="Manager" />
      </scen:agent>
    </scen:container>

      <scen:container name="C02">
        <scen:agent>
        <scen:parameter name="loader" value="adf2" />
        <scen:parameter name="class" value="Location" />
        <scen:parameter name="name" value="Location02" />
        <scen:parameter name="GUI" value="LocationGui" />
        <scen:parameter name="java-code" value="LocationFunctions" />
        <scen:parameter name="location" value="C02" />
        <scen:parameter name="fixed" value="true" />
        <scen:parameter name="parent" value="Manager" />
      </scen:agent>
    </scen:container>
    <!-- ... -->
   </scen:initial>

   <scen:subsequent>
      <scen:agent>
      <scen:parameter name="loader" value="adf2" />
      <scen:parameter name="class" value="Ant" />
      <scen:parameter name="java-code" value="AntFunctions" />
      <scen:parameter name="GUI" value="null" />
    </scen:agent>
   </scen:subsequent>
   <!-- timeline following -->
</scen:scenario>
```

Fig. 17. Part of the scenario XML file for the ACO scenario

```
(agent Ant
  (behavior
    /*...*/
    (reactive startSeekingFood
      (receive (message findFood))

      //Changing the mental state in order to reflect the fact that
        the iteration was started.
      (addK (struct knowledge iterationStarted))
      (addK (struct knowledge stepRequired))//used when a new step is required

      (out) //getting out of the ant hill

      //Initializing the list of visited locations
      (add_visited_location parent ??visitedLocationsList)
      (addK (struct knowledge visitedLocationsList ??visitedLocationsList))
    )
    /*...*/
    (cyclic playGame
      (condition (readK (struct knowledge iterationStarted)))
      (condition (readK (struct knowledge stepRequired)))

      // Ask parent for neighboring locations, the distances towards
        them and the pheromone trails:
      (send parent (message neighborsAndInfoRequest))
      (removeK (struct knowledge stepRequired))
    )

    (reactive receiveNeighborsAndInfo
      (receive (message neighborsAndInfo ?neighborsList
        ?pherList ?distancesList))

      (readK (struct knowledge visitedLocationsList ?visitedLocationsList))
      (readK (struct knowledge alpha ?alpha))
      (readK (struct knowledge beta ?beta))
      (if (getNextLocation ?neighborsList ?pherList ?visitedLocationsList
        ?distancesList ?alpha ?beta ?nextLocation)
      then
        (send parent (message whoManagesLocation ?nextLocation))
      else
        (addK (struct knowledge returning))
      )
    )
    /*...*/
  )
)
```

Fig. 18. *Ant* class, defined in the file *Ant.adf2*

the iteration. In this mode the ant enters when it finds and collects food as well, case in which the pheromone trails will be positively influenced.

7 Conclusion

In this document we presented the architecture and a series of important technical aspects related to the tATAmI platform, starting with the requirements that we considered before designing and implementing it. tATAmI allows the quick and easy implementation of mobile MAS to be deployed on hybrid networks of Java-enabled computers and Android mobile devices, that can interoperate with other systems by means of web-services. All these, together with a powerful context-awareness capability makes it, together with the S-CLAIM language, a suitable platform for the development of agent-based AmI applications. Furthermore, we offered some implementation examples based on several simple scenarios.

References

1. Baljak, V., Benea, M.T., Seghrouchni, A.E.F., Herpson, C., Honiden, S., Nguyen, T.T.N., Olaru, A., Shimizu, R., Tei, K., Toriumi, S.: S-claim: An agent-based programming language for ami, a smart-room case study. Procedia Computer Science **10**(0) (2012) 30 – 37 ANT 2012 and MobiWIS 2012.
2. Bellifemine, F., Poggi, A., Rimassa, G.: Developing multi-agent systems with JADE. Intelligent Agents VII Agent Theories Architectures and Languages (2001) 42–47
3. Cardelli, L., Gordon, A.D.: Mobile ambients. Theor. Comput. Sci. **240**(1) (2000) 177–213
4. Chen, G., Kotz, D., et al.: A survey of context-aware mobile computing research. Technical report, Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College (2000)