tATAmI-2 – a Flexible Framework for Modular Agents[1]

Andrei Olaru
Department of Computers
University Politehnica of Bucharest
313 Splaiul Independentei
060042 Bucharest, Romania
Email: cs@andreiolaru.ro

***Abstract*—The paper introduces tATAmI-2, an agent development framework that allows the creation of modular agents and permits a great deal of flexibility with respect to the manner in which various functionality, such as agent communication, is implemented. The framework strikes a good balance between flexibility and ease of use, by offering several pre-implemented agent components and communication platforms. The architecture of the framework relies on three elements: the ability to simultaneously start multiple platforms for agent management and communication; the ability to load agents in a number of manners; and, in the case of composite agents, the ability to customize the component set of an agent, including the possibility to add application-specific components, or to use components recommended by the platform for certain functionalities (such as communication).**

I. Introduction

There are currently a reasonably large number of frameworks that allow developers to design, create, deploy and monitor multi-agent systems. Their focus varies from research to industrial applications. They allow deployment of agents programmed in various languages, from Java to specific AOP languages such as Jason.

Throughout our work, we have found that existing platforms lack flexibility and ease in deployment and management. They either enforce a specific means of communication between agents, or a specific programming language, or specific requirements for the host system.

This paper introduces the tATAmI-2 framework – towards Agent Technologies for Ambient Intelligence, 2^{nd} generation.

The main goal of tATAmI-2 is flexibility. It does not enforce on the agent developer a platform, a means of communication, a model for the agent, a means of reasoning or a representation for knowledge (although some basic restrictions exist, such as messages being translatable to character strings). At the same time, it offers various pre-existing implementations for each of these elements, as well as the possibility for the developers to implement or extend their own. tATAmI-2 offers a means to connect various implementations that offer these functionalities and make them work together as an agent system.

Another important goal of tATAmI-2 is to offer the agent developer a framework which is easy to understand and use, and quick to deploy. Repeatable, one-click deployment is a central issue.

Although the features offered by tATAmI-2 are one way or another offered by existing frameworks, we believe that tATAmI-2 strikes a great balance between the range of features that it offers and the easiness of using those features.

The architecture of tATAmI-2 relies on three main elements: the platform, the agent, and the agent component. The platform is what handles agent communication and possibly other cross-machine features (such as agent mobility). The agent is a hub for agent components, which in turn implement agent functionalities, as independently as possible from other agent components, and regardless of the concrete implementation of other components (for instance, regardless of how inter-agent messaging happens).

This paper describes in detail the mechanisms that have been implemented in order to achieve this level of flexibility in the framework, through interfaces that specify a minimal number of functionalities, and a bootstrap process that loads all the elements in a deployment in the appropriate order.

After discussing other existing frameworks in the next section, we give more details on the motivation for creating tATAmI-2 in Section III. Section IV describes the architecture and components of the framework. Deployment of tATAmI-2 applications is discussed in Section V. The last section draws the conclusions and introduces the aspects of future work.

II. Related Work

In the domain of agent development frameworks, we see two types of frameworks. Some, such as JADE, JIAC, JACK and Jadex rely on Java agent implementations and offer various functionalities and APIs for agent management, communication and security. These are the frameworks most used for business and industrial applications. Other frameworks, such as Jason, Agent Factory and JaCaMo, rely on some agent-oriented programming (AOP) language, usually Jason /

[1]The final publication can be found on IEEE Xplore
Olaru, A., *tATAmI-2 – a Flexible Framework for Modular Agents* in Dumitrache, I., Florea, A. M., Pop, F. and Dumitrascu, Alexandru (eds.), Proceedings of CSCS 20, the 20th International Conference on Control Systems and Computer Science, May 27-29, Bucharest, Romania, IEEE Xplore, 2015, pages 703-710
http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7168503

AgentSpeak for the implementation of agents, reasoning, and planning.

JADE [1] was the base of tATAmI-1 and it was an inspiration while creating tATAmI-2. It is a powerful and easy to use agent development and deployment framework that offers agent management, mobility, and communication for agents implemented in Java. One of the drives to move tATAmI-2 away from Jade was a desire to increase the performance of the agent system, to increase its flexibility, and to be able to implement other means of communication between agents (such as through WebSockets).

Jadex [2] is a reasoning engine that extends Jade agents with reasoning capabilities (a BDI model), agent components, and business-oriented features. However, developing agents in Jadex is requires combining Java and XML in advanced specifications that may be difficult to grasp by the agent developer.

JACK Intelligent Agents [3] is a production-grade framework for building BDI agents, building on the experience of PRS [4] and dMARS [5]. It is very well suited for building agent systems quickly and with relative ease, however it does not feature the flexibility we are trying to offer in choosing the messaging components, platforms, and concrete agent implementations.

JIAC [6] is another production-grade framework for developing complex agent systems, on both workstations (JIAC-V) and mobile/embedded devices (microJIAC). Its focus on industrial applications, by offering features for security, management and scalability, is what differentiates it from tATAmI, which is focused on lightweight, repeatable testing and execution of AmI applications.

Jason [7] is based on AgentSpeak and is an AOP language with its own special syntax to define goals, conditions, and plans. Jason agents can be run on top of Jade when a distributed setup is required. Jason is however constraining the developer into a particular way of programming agents, which some may find difficult to grasp, especially for developers not familiar with agent theory.

Agent Factory [8], together with its implementation for mobile devices – Agent Factory Micro Edition [9], includes flexibility for implementing agents in various programming languages, but currently contains development kits only for Jason-based implementations, therefore the developer must be familiar with Jason.

JaCaMo [10] is a framework that combines three components: the Jason programming language, Moise organizations features, and the CArtAgO artifact and environment infrastructure. JaCaMo is probably one of the most suited frameworks for deploying MAS, however creating a JaCaMo deployment involves considerable work and requires a great deal of background theoretical knowledge. Performing even common agent-related tasks is not easy to do.

Repast Simphony [11] is a great open source and free environment for agent-based modeling of complex adaptive systems. It offers a varied range of tools for modeling and analysis. It's goal is, however, agent simulation. It does not support deployment over networks of heterogeneous devices and platforms, nor is it focused on heterogeneous agent systems.

III. CONTEXT AND MOTIVATION

The development of tATAmI-2 started with the modularization and enhancement of tATAmI-1, a Jade-based framework for AmI-oriented applications implemented in S-CLAIM [12]. However, the process of modularization involved a great deal of changes, resulting in a framework which is almost completely new.

We consider the tATAmI-1 and tATAmI-2 frameworks as being suitable for AmI applications (such as person detection and tracking [13]) thanks to the focus on performance and distribution, agents of variable size and functionality, and focus on context-awareness.

A. Features in tATAmI-1

After the successful implementation of Ambient Intelligence scenarios using the CLAIM AOP language and the SyMPA platform [14], the tATAmI-1 framework was implemented from the ground up, with new code, to replicate the old functionality, but to be easier to use, and allow implementation of agents in the new and improved S-CLAIM language [12].

tATAmI-1 agents relied on a layered architecture, having a layer for each main functionality of the system: visualization, web services interface, hierarchical agent mobility, and S-CLAIM behavior execution. This layer stack consisted of classes extending the Jade `Agent` class.

S-CLAIM is a behavior-centric agent programming language that focuses on agent-specific primitives, such as for sending and receiving messages, knowledge management, mobility, and input and output to a visual interface. The language has a LISP-like syntax. While the language contains some algorithmic features – such as the `if` primitive and a loop over knowledge records – arithmetic operations and complex algorithmic functionality are implemented in external Java libraries of pure functions (see Figure 6 for an example of source code).

One of the drawbacks of tATAmI-1 is the reliance on Jade [1] for agent management, agent communication, agent mobility, and even S-CLAIM behavior management. This poses problems in some contexts involving mobile devices and also implies some performance issues due to Jade. Another drawback is that the layers of the agent cannot be easily removed or replaced, even with recompilation of the source, as the implementations are strongly interconnected. Basically, developers of tATAmI-1 applications were constrained by the framework's architecture, much like in other frameworks described in related work.

B. Requirements for tATAmI-2

The requirements for tATAmI-2 were primarily to solve the drawbacks of tATAmI-1 in order to obtain a maximum of flexibility and performance. We can split these requirements in two sets: one consisting of features that already exist in

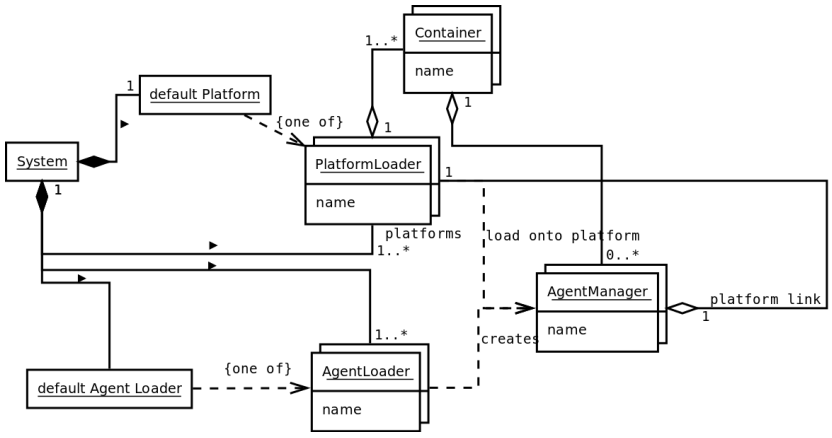

Figure 1. The various instances existing in an execution of the system: there are several platform loaders, of which one is default; there are several agent loaders, of which one is default; each platform has a number of containers, which in turn have a number of agents (viewed as agent managers). Agents are created (loaded) by an appropriate agent loader, and then loaded onto a platform, to which they have a platform link reference.

tATAmI-1, and one consisted of features that did not. The requirements solved in tATAmI-1 and that must be maintained in tATAmI-2 are:

- have a framework in which it takes little time and effort to go from agent design to implementation and deployment;
- have the possibility to implement agents in an AOP language that is easy to read, understand and learn by developers who are less familiar with various general-purpose programming languages;
- provide a means to deploy a MAS at one click even across multiple machines and platforms, with minimal intervention on machines other than the launching workstation;
- offer interoperability with other platforms and services;
- use agents that are autonomous and mobile in the system;

The additional requirements for tATAmI-2 are:

- work independently from Jade and the Jade library, while offering all of the features offered by tATAmI-1;
- have the ability to specify for each agent which functionality to include and how it should be implemented, without recompiling the source code, by means of the deployment scenario specification (and potentially not have the code for the component at compile time);
- be able to use various means of communication between agents, such as TCP/IP, WebSockets[1], web services, or the Jade platform.

This paper presents the architecture of the tATAmI-2 framework, which fulfills the requirements above (except for, at the present moment, web service interoperation and execution-time agent mobility).

IV. tATAmI-2 Architecture

The tATAmI-2 framework, implemented[2] in Java, is built around 3 main elements: *platforms*, *agents*, and *agent components* (see Figure 1). Agent components are normally used only by *Composite Agents*. While it is not required that all agents loaded in the system are composite (made of agent components), we will only discuss in this paper the composite type, as they are a central feature of tATAmI-2.

A. System Structure

A **platform** is an infrastructure that enables various features for agents, such as communication. One such platform can be Jade. Another may be underpinned by WebSocket communication. And so on. More platforms can be started simultaneously during the same execution. The system manages a platform through an instance of the `PlatformLoader` interface. A platform has a *name* and has methods to *start* and *stop* the platform, *create a container* and *load an agent*. It also can be queried what implementation it recommends for a particular component type. For instance, if an agent must contain a "messaging" component, but does not specify the exact implementation (via class path), the system loads the component recommended by, and therefore specific to, the platform (as in the example in Figure 5).

An **agent** is characterized by its name. An agent resides in a container, on a platform. A container must belong to a platform, and only to one. An agent must reside in a container, and only in one. Multiple agents may reside in the same container, and multiple containers may reside on the same machine. A container is bound to a specific machine, while an agent may move between containers (and therefore machines), if the platform supports agent mobility.

Any agent that is loaded on a platform must implement the `AgentManager` interface, which has methods for inquiring the agent's name, for *starting* and *stopping* the agent, and for passing a link to the platform to the agent, as an implementation of the marker interface `PlatformLink` (which does not specify any methods). Setting the *platform link* can be done by the platform, when the agent is loaded, so that the platform-specific components can access functionality in the platform.

An agent of a particular type (e.g. composite) is loaded by an implementation of the `AgentLoader` interface. An agent

[1]http://en.wikipedia.org/wiki/WebSocket

[2]The tATAmI-2 source code is open source, licensed under a GNU LGPLv3 license, and is available at https://github.com/tATAmI-Project/tATAmI-PC/tree/tATAmI-2/master.

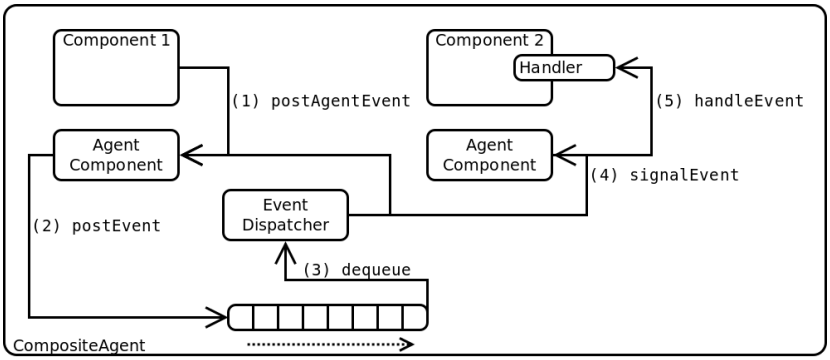

Figure 2. An agent component uses the methods in the class AgentComponent, which it extends, to post events and to register handlers for events (by type). An event dispatcher dequeues events from the event queue of the agent and signals the event to the components, each component calling the handler registered for that event (if any).

loader is able to *pre-load* the agent creation data for an agent, and later, using this creation data, to *load* the agent and return an `AgentManager` instance (see Section IV-D).

A composite agent contains various **components** that offer specific functionalities. Components have access to other components and to the platform, identifying them by type or name. For instance, a visualization component is able to contact the messaging component of the agent, if any, to send and receive messages. The messaging component, which is specific to the platform (e.g. agents load a specific class for using messaging on Jade, according to the recommendations of the Jade platform loader), is able to contact the platform in a specific way so as to actually send a message to another agent. Components are also able to post agent events, that the agent disseminates to all other components (see Figure 2).

One execution of the system (also called a *simulation*) is overseen by the **Simulation Manager**, which provides a GUI that contains means of visualizing the state of the system and controls that enable the user to start, pause and stop the system (see Figure 7). The Simulation Manager creates and maintains references to Simulation Link Agents, one per platform, that enable it to send and receive control messages to and from the agents on each platform.

B. Composite Agent and Agent Components

A composite agent contains a list of components, an event queue, an event processing thread, and the agent state (see Figure 2). Any component can post an event in the event queue. When an event is picked from the event queue, it is broadcast to all components.

A composite agent is in one of the following states:

- *pre-loaded* – the agent instance has not yet been created, but all creation information about it has been loaded and checked by the agent loader; the components have been created, but not added to the agent;
- *initializing* – the agent has been created, and components are being added to it;
- *starting* – the event processing thread has been started and a START event has been posted; the components are starting as a result; by now all components have been added to the agent and the agent has been loaded on a platform, therefore the *platform link* is active;
- *running* – all components have started and the agent is processing normally.
- *stopping* – a STOP event has been posted; no more events are accepted and the components are stopping;
- *stopped* – all components have stopped. The agent can be started again at this point, or it can be instructed to destroy all components and exit.

Component implementations must extend the AgentComponent class, which offers to extending classes several functionalities, such as access to scenario component data, access to composite agent functionality (agent name, posting agent events, event handler registration, platform link) and to functionality of some other, usual, components, such as the messaging component (sending messages and registering message handlers) and the visualization component (access to the agent log).

Components are able to register event handlers for any of the standard events in the agent lifecycle (agent start and stop, simulation start and pause, agent movement), or for custom events the developer knows may happen (such as GUI input).

Any component has both direct access (by requesting a reference from the parent agent) and indirect access (through the event mechanism) to any other component in the agent.

Usual agent components are the Parametric Component – managing agent parameters; the Visualizable Component – managing the agent's log, reporting to the Simulation Manager, and handling of control messages (such as starting the simulation or stopping the agent); the Messaging Component – sending messages and managing message handlers for the agent; the Mobility Component – managing the movement of the agent from one machine to another; the Cognitive Component – managing the agent's knowledge base through standard functions such as adding, removing, and retrieving pieces of knowledge matching a certain pattern; and the S-CLAIM Component – executing S-CLAIM behaviors as specified by the agent definition file (such as the one in Figure 6).

In a way, components, as processes activated by agent

events, are similar to Jade's behaviors, activated (many times) by received messages. However, components address a broader range of functionality, such as different implementations of agent messaging and mobility. Platforms may be developed together with special components so that, by means of the *platform link*, the component is able to access specific functionalities that other components are unaware, and therefore independent of.

C. Messaging

An interesting aspect of the internal organization of composite agents is messaging. While no specific means of sending messages from one agent to another is enforced, in order to ensure interoperability of components, there are some internal standards.

Any message sent between tATAmI-2 agents contains three elements: a source endpoint, a destination (or target) endpoint, and the content. All three must be strings of characters. An endpoint (or path) is formed of two parts: the external path, designating the agent, and the internal path, designating the component, functionality, etc. (see Figure 3).

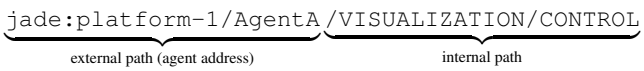

Figure 3. Endpoint structure. The format of the external path depends on the communication platform.

The format of the external path depends exclusively on the communication platform that is used. Components are able to register with the messaging component handlers for specific internal path prefixes (e.g. the VisualizableComponent may register a handler for all messages with the internal patch beginning with `/VISUALIZATION`).

For the developer of an agent component, working with messages is simple, as the base `AgentComponent` class offers the methods `registerMessageReceiver(handler, prefix)` and `sendMessage(content, sourceEndpoint, targetEndpoint)`, as well as methods for facilitating the construction and deconstruction of agent paths.

For the developer of a communication platform, the handling of messages once they are received is done by the abstract `MessagingComponent` class, that any messaging component implementation must extend. What is left to do is:

- create a class implementing the `PlatformLoader` interface, that handles communication between agents;
- implement a platform-specific messaging component (that extends `MessagingComponent`), which is able to convert the platform-specific format of messages into the source-endpoint/target-endpoint/content format;
- when the agent starts (and is therefore already loaded on the platform), the messaging component must register with the platform (via the agent's *platform link* reference), as the message handler for its parent agent;
- when a message must be sent, the messaging component may use the platform link reference to send the message.

What the platform link actually points to depends on the platform. Components using this reference must be aware of this (e.g. be platform specific). In some cases, the reference may point to the `PlatformLoader` instance. In other cases, for instance in the implementation for Jade, the platform link points to a Jade `Agent` implementation, that wraps the `AgentManager` instance.

D. Platform Bootstrap and Control System

All the information about what platforms, agents and components should be created, on how to do that and with what parameters, is contained in the *scenario file*. Each execution is completely specified by one such file, which is in XML format. The scenario file (see an example in Figure 5) contains information on:

- general configuration that all platforms can use, such as IP and port of the local machine, and IP and port of a remote 'central' machine.
- names and parameters of the platforms to start; if a platform is one of the standard pre-implemented platforms, the name suffices to identify the corresponding `PlatformLoader` class; otherwise, the class is searched in the package with the same name as the platform; for other setups, the exact class path must be specified.
- names and parameters of the agent loaders to use; if the agent loader is not standard, the exact class path of the AgentLoader implementation must be specified.
- packages that contain various elements that are necessary to agents, such as agent definition files, code attachments, etc.
- names of containers, platforms to create the containers on, and indications whether the containers are created locally or will be created remotely.
- names and parameters of agents to be created in each container.
- 'timeline' information: moments of time and content of messages to send, from the Simulation Link Agents, to specified recipients, at those moments of time.

The lifecycle of the framework is divided among the phases visible by the user, which are:

- start of execution (*Boot*) – all necessary classes, as well as the scenario file, exist on the local machine;
- start of the Simulation Manager GUI – by now, all elements of the scenario have been checked; agent data has been preloaded; components have been created, but not initialized; platforms have been started; Simulation Link Agents have been started on all successfully started platforms;
- start of the system on other machines – systems on other machines connect to the central machine and receive data for agents to be created locally;
- "Create Agents" button is pressed – on each machine, agents are created, components are loaded, and initial communication takes place;
- "Start" button is pressed – all agents receive a message to initialize simulation;

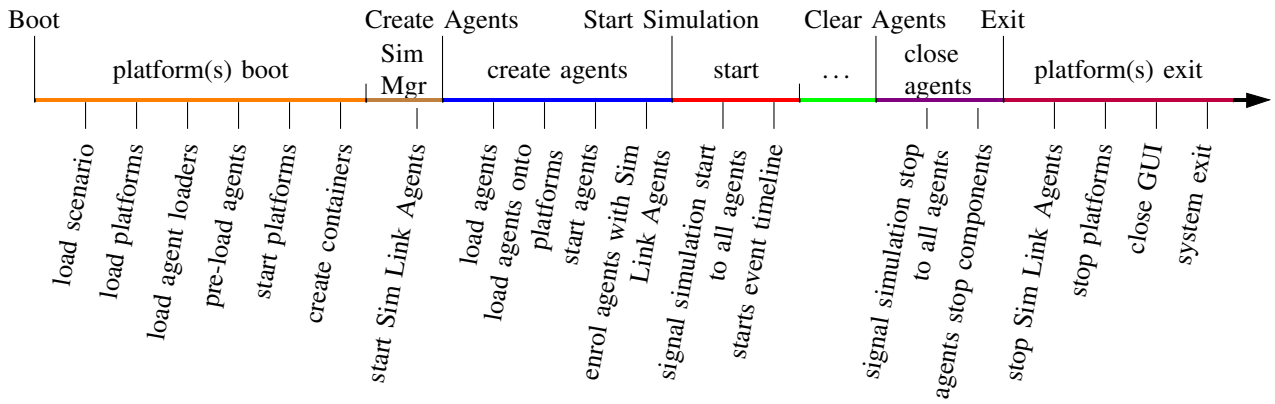

Figure 4. Timeline of an execution of tATAmI-2. Marked above the timeline are events and phases as observed by a user. Below the timeline are processes that take place inside the framework.

```
1  <scen:platform><scen:parameter name="name" value="local" /></scen:platform>
2
3  <scen:initial><scen:container name="Container">
4    <scen:agent>
5      <scen:component name="parametric" />
6      <scen:component name="visualizable" />
7      <scen:component name="messaging" />
8      <scen:component name="testing" classpath="scenario.examples.PingBackTestComponent">
9        <scen:parameter name="other_agent" value="AgentB" />
10       <scen:parameter name="initiator" value="true" />
11     </scen:component>
12     <scen:parameter name="loader" value="composite" />
13     <scen:parameter name="name" value="AgentA" />
14   </scen:agent>
15   <scen:agent>
16     <scen:component name="parametric" />
17     <scen:component name="visualizable" />
18     <scen:component name="messaging" />
19     <scen:component name="testing" classpath="scenario.examples.PingBackTestComponent" />
20     <scen:parameter name="loader" value="composite" />
21     <scen:parameter name="name" value="AgentB" />
22   </scen:agent>
23 </scen:container></scen:initial>
```

Figure 5. Snippet of a scenario file, specifying a simulation using the local communication platform, in which each of the two agents contains a parametric, a visualizable and a messaging component, together with a custom component which may have some parameters. The loader for each agent, as well as agent names, are also specified.

- "Clear agents" button is pressed – all agents receive a message to exit;
- "Exit" button is pressed – Simulation Link Agents exit and all platforms are closed; the Simulation Manager exits and closes the JVM.

In the first phase, the scenario file is loaded and parsed. Based on the information in it, the classes for platform loaders and agent loaders are identified and instantiated. For each agent agent data is preloaded and components are created. Platforms are started and local containers are created.

After all platforms have been started, the Simulation Manager is started, which in turn loads, on each platform, a Simulation Link Agent.

When the user presses the "Create Agents" button, the Simulation Manager uses the appropriate Agent Loader to load each agent, based on the agent's creation data, starts it, then invokes the appropriate Platform Loader to load the resulting Agent Manager instance onto the platform. Each agent is then enrolled with the Simulation Link Agent in its platform, to which it will report visualization data.

When the user presses the "Start" button, if the agents have not yet been created, they are created. Then, it signals a START_SIMULATION event to all agents, and starts processing events on the scenario timeline (if any).

The simulation continues normally until the "Clear Agents" button is pressed. At this point the Simulation Link Agents broadcast a STOP event to all agents. Then the "Exit" button stops the platforms and closes the Simulation Manager. If the agents had not been cleared, agents are cleared and then the platforms stops.

To facilitate monitoring the system, every agent, by means of its Visualization Component, reports all of its logging

```
(agent ChatAgent ?otherAgent
(behavior
    ...
  (reactive snd
    (input messageinput text ?sendMessage)              // the agent is notified there is a message to send
    (readK (struct knowledge sequence ?sequence))       // read the sequence number from the knowledge base
    (removeK (struct knowledge sequence ?sequence))     // remove the stored sequence
    (addK (struct knoweldge msg sent ?sequence ?sendMessage))          // store the message in KB
    (increment ?sequence ?newsequence)                                 // increment the sequence
    (addK (struct knowledge sequence ?newsequence))                    // store the sequence

    (send ?otherAgent (struct message newchat ?sendMessage))           // send the message

    (initOutput ??output)                                              // will init output to an empty string
    (forAllK (struct knowledge msg ??direction ??sequence ??message)   // go through all messages
        (assembleOutput ??direction ??sequence ??message ??output) )   // put them in the ??output variable
    (output chatlog ??output)                                          // display
))
```

Figure 6. S-CLAIM code for one of the two reactive behaviors in a chat agent. The behavior is triggered by message input. It then increments the sequence stored in the knowledge base, stores the message in the KB and sends it, then reads the messages in the KB and assembles them for output. See how Java functions (such as `increment`, `initOutput` and `assembleOutput`) are called in the same manner as standard primitives.

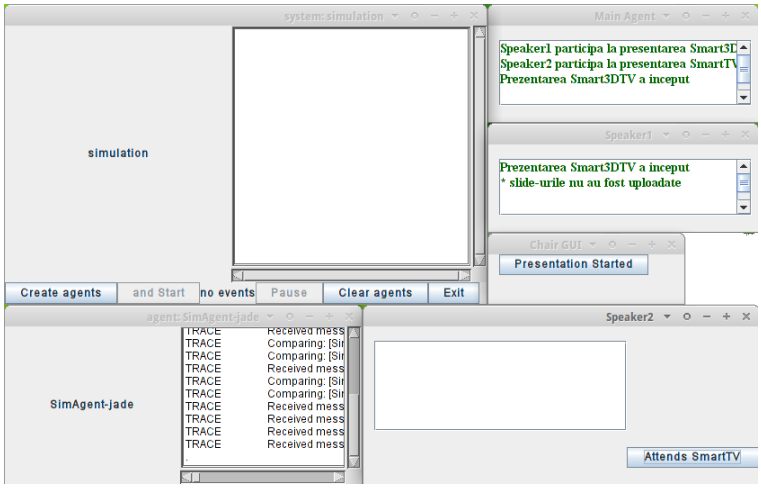

Figure 7. Various GUIs of the Smart Conference S-CLAIM application, deployed using tATAmI-2. One can also see the GUI of the Simulation Manager and the Simulation Link Agent (SimAgent-jade).

messages to the Simulation Link Agent on the same platform. These messages are gathered, sorted according to their timestamp, and displayed in order, so that the user can follow the evolution of all the agents in the system, even if they are on different machines.

To facilitate visualization, the `WindowLayout` class is used to automatically layout the windows corresponding to various agents on the screen, so the user does not have to arrange windows by hand.

V. Deploying MAS with tATAmI-2

Deploying multi-agent systems with tATAmI-2 does not require knowledge of the details of the framework's implementation.

If agents are implemented in S-CLAIM, the developer only needs to modify a scenario file that is provided in the examples in the project, and then write the S-CLAIM code. If algorithmic processing is needed (functions such as `increment` in Figure 6) and not yet implemented, only basic Java knowledge is required[3].

For developers wishing to implement functionality as components for the Composite Agent, it is only required to be familiar with two things: the functionality offered by standard tATAmI-2 components (such as the Visualization Component) and the functionality offered by the `AgentComponent` class. All standard components are thoroughly documented.

tATAmI-2 already contains default Parametric, Visualization, Cognitive and S-CLAIM components, as well as messaging components and platforms for local messaging and for Jade. A WebSocket messaging platform is under implementation.

Once all necessary classes exist and the scenario file has been created, the tATAmI-2 system can be started by passing the scenario name to the tATAmI-2 application. Except for the Java Runtime Environment, no other requirements for the host system exist.

[3]The reader may view an example at https://github.com/tATAmI-Project/tATAmI-PC/blob/tATAmI-2/master/src-scenario/scenario/examples/sclaim_tatami2/simpleScenarioE/ChatFunctions.java

For scenarios involving multiple machines, it is sufficient that tATAmI-2 finds the scenario file on the 'central' (or 'main') machine, much like in Jade. On other machines, tATAmI-2 can be started by specifying in the program arguments the IP of the central machine and the name(s) of the local container(s).

One of the prominent features of tATAmI-1 is the possibility to deploy agents on the Android platform, by using Jade for Android. While tATAmI-2 has not yet been deployed on Android, all the core functionality is platform-independent, and it is compatible with the way tATAmI-1 was ported, so tATAmI-2 portability to Android is just around the corner.

A. Deployment Case

In the spring semester of 2014, during her bachelor thesis, Emma Sevastian has developed a relatively large, distributed application using S-CLAIM over tATAmI-2 [15]. The application related to an initiative to build an AmI application for Smart Conferences. In the scenario, there is an agent for each conference participant – participants can be speakers and chairs of conference sessions. The application handles the lifecycle of conference sessions and notifications for speakers when sessions they are interested in begin. The GUIs shown in Figure 7 belong to various agents in the system.

VI. Conclusion

In a world of complexity, we have created a modular, flexible agent development framework that allows the developer to change and customize every aspect in the implementation of agents and platforms. At the same time, standard components and platforms, already implemented and ready to use, allow the developer to boot an agent simulation in minutes (having already implemented any application-specific functionality).

The framework provides tools to deploy and monitor multi-agent systems across multiple machines, and will soon be available for Android devices as well.

Future work consists, in the near future, of including more functionality in tATAmI-2, primarily communication using WebSockets and agent mobility at execution time. Other features that will be implemented involve the integration of context-awareness as a first-class notion for agents, and more powerful knowledge representation, using context graphs and patterns.

Acknowledgment

The authors would like to thank Thi Thuy Nga Nguyen and Diego Salomone-Bruno for their participation in the first demos of AmI applications using CLAIM; Marius-Tudor Benea for his great implication in tATAmI-1, precursor of tATAmI-2; Amal El Fallah Seghrouchni and Cédric Herpson for their help in coordinating tATAmI-1 progress; and finally Emma Sevastian for her dedication in improving early versions of tATAmI-2 and building a complex application deployed using tATAmI-2. And finally professor Adina Magda Florea for coordinating the research efforts of the AI-MAS Laboratory.

The work has been funded by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Ministry of European Funds through the Financial Agreement POSDRU/159/1.5/S/134398.

References

[1] F. Bellifemine, A. Poggi, and G. Rimassa, "Developing multi-agent systems with JADE," *Intelligent Agents VII Agent Theories Architectures and Languages*, pp. 42–47, 2001.

[2] L. Braubach and A. Pokahr, "Jadex active components framework-BDI agents for disaster rescue coordination." *Software Agents, Agent Systems and Their Applications*, vol. 32, pp. 57–84, 2012.

[3] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas, "Jack intelligent agents-summary of an agent infrastructure," in *5th International conference on autonomous agents*, 2001.

[4] M. P. Georgeff and A. L. Lansky, "Reactive reasoning and planning." in *AAAI*, vol. 87, 1987, pp. 677–682.

[5] M. d'Inverno, M. Luck, M. Georgeff, D. Kinny, and M. Wooldridge, "The dMARS architecture: A specification of the distributed multi-agent reasoning system," *Autonomous Agents and Multi-Agent Systems*, vol. 9, no. 1-2, pp. 5–53, 2004.

[6] M. Lützenberger, T. Küster, T. Konnerth, A. Thiele, N. Masuch, A. Heßler, J. Keiser, M. Burkhardt, S. Kaiser, and S. Albayrak, "JIAC V: A MAS framework for industrial applications," in *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*. International Foundation for Autonomous Agents and Multiagent Systems, 2013, pp. 1189–1190.

[7] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons, 2007, vol. 8.

[8] S. Russell, H. Jordan, G. M. O'Hare, and R. W. Collier, "Agent factory: a framework for prototyping logic-based AOP languages," in *Multiagent System Technologies*. Springer, 2011, pp. 125–136.

[9] C. Muldoon, G. M. P. O'Hare, R. W. Collier, and M. J. O'Grady, "Agent factory micro edition: A framework for ambient applications," in *Proceedings of ICCS 2006, 6th International Conference on Computational Science, Reading, UK, May 28-31*, ser. Lecture Notes in Computer Science, V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds., vol. 3993. Springer, 2006, pp. 727–734.

[10] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi, "Multi-agent oriented programming with JaCaMo," *Science of Computer Programming*, vol. 78, no. 6, pp. 747–761, 2013.

[11] M. J. North, N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko, "Complex adaptive systems modeling with repast simphony," *Complex Adaptive Systems Modeling*, vol. 1, no. 1, pp. 1–26, 2013.

[12] V. Baljak, M. T. Benea, A. El Fallah Seghrouchni, C. Herpson, S. Honiden, T. T. N. Nguyen, A. Olaru, R. Shimizu, K. Tei, and S. Toriumi, "S-CLAIM: An agent-based programming language for AmI, a smart-room case study," in *Proceedings of ANT 2012, The 3rd International Conference on Ambient Systems, Networks and Technologies, August 27-29, Niagara Falls, Ontario, Canada*, ser. Procedia Computer Science, vol. 10. Elsevier, 2012, pp. 30–37. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050912003651

[13] M. Chiperi, M. Trascau, I. Mocanu, and A. M. Florea, "Data fusion in a multi agent system for person detection and tracking in an intelligent room," in *Intelligent Distributed Computing VIII*. Springer, 2015, pp. 385–394.

[14] A. Suna and A. El Fallah-Seghrouchni, "A mobile agents platform: architecture, mobility and security elements," in *Programming Multi-Agent Systems*. Springer, 2005, pp. 126–146.

[15] I.-E. Sevastian, "Agentbased android application for conference participants," University Politehnica of Bucharest, Bachelor Thesis, September 2014. [Online]. Available: http://aimas.cs.pub.ro/amicity/doc/EmmaSevastian-document.pdf