Jason Interpreter, Enterprise Edition

Dejan Mitrović and Mirjana Ivanović
Department of Mathematics and Informatics, Faculty of Sciences
University of Novi Sad, Novi Sad, Serbia
E-mail: {dejan, mira}@dmi.uns.ac.rs

Rafael H. Bordini
Postgraduate Programme in Computer Science – School of Informatics (FACIN)
Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, RS, Brazil
E-mail: r.bordini@pucrs.br

Costin Bădică
Computer and Information Technology Department
Faculty of Automatics, Computers and Electronics, University of Craiova, Romania
E-mail: cbadica@software.ucv.ro

Keywords: intelligent agents, Java EE, Jason, AgentSpeak

Received: July 24, 2015

The Enterprise edition of the Java platform has been endorsed by both small and large enterprises, as it enables the development of large-scale, reliable, and secure software solutions. In the world of agent development, on the other hand, AgentSpeak, and its practical interpreter Jason, represent one of the most popular tools for writing complex, reasoning agents. This paper presents a framework that integrates the two approaches to distributed software development, and supports a seamless deployment of Jason agents in enterprise environments. The proposed framework offers many technical advantages, including automated agent load-balancing and fault-tolerance. The end-goal of this research, however, is to try and bridge the gap between the agent technology and modern enterprise applications.

Povzetek: Predstavljena je izpopolnjena platforma za agente v Jasonu z namenom izdelave agentnih aplikacij.

1 Introduction

Java platform, Enterprise Edition (Java EE), is designed to support the development of scalable, secure, and reliable software products [11]. It is built around the idea of code reuse, and incorporates many libraries, frameworks, and technical solutions. As such, Java EE is often utilized as the main software development platform by small and large enterprises.

When it comes to agent development, most existing multiagent frameworks are written using the *Standard Edition* of Java (Java SE) [6]. On the other hand, as discussed in [14, 24], the use of Java EE can significantly reduce the effort needed to develop efficient multiagent frameworks. In addition, it can simplify the process of integrating agents into enterprise applications.

Extensible Java EE-based Agent Framework (XJAF) [16, 14, 24] is a multiagent framework built on top of Java EE. It utilizes technical solutions of Java EE in order to support scalable and reliable multiagent systems. More concretely, XJAF runs on top of computer clusters in order to provide *high-availability* of deployed applications, which is concerned with scalability and uninterrupted delivery of services, i.e. regardless of software or hardware failures [25].

Although the Java programming language is well-suited for many scenarios, the process of writing complex, reasoning agents often requires a special, *agent-oriented* programing language (AOPL) [6]. An AOPL provides programming constructs that enable developers to apply and use advanced multiagent concepts in practice. One of the most popular AOPLs is *AgentSpeak*, which directly supports the popular *Belief-Desire-Intention* agent architecture [21, 19]. To a great degree, the language owes its popularity to the *Jason* interpreter [4, 5]. Jason is a practical and efficient Java-based interpreter for an extended version of AgentSpeak, with a highly-customizable architecture.

This paper presents our most recent research efforts aimed at extending Jason with the support for enterprise environments. The new edition of Jason, named *Jason Interpreter, Enterprise Edition* (Jason EE), is integrated into XJAF, and uses its agent-oriented abstractions of Java EE technologies. This integration results in several advantages. First and foremost, being enterprise components themselves, Jason EE agents can interact with other parts of regular enterprise applications in a straightforward fashion. For example, a Jason EE agent can easily interact with web services or expose its capabilities in form of a web service, manage data in a remote (relational or NoSQL) database, etc. This integration could, therefore, help bridging the gap between agent technology and business, enterprise applications.

On the technical side, Jason EE provides agent load-balancing, thread pooling, and fault-tolerance. Load-balancing is concerned with automatic distribution of agents across the computer cluster [15]. It spreads the computational load, and enables Jason EE to run large number of agents. Thread pooling stems from the use of XJAF as the underlying multiagent framework. In XJAF, there is no thread-to-agent mapping [16]. An agent is assigned a thread whenever it needs to perform some processing. In the worst-case scenario, when all agents need to run simultaneously, there will be as many threads as there are agents, but the underlying enterprise application server tries to reduce resource consumption otherwise. As a result, XJAF and Jason EE can run many more agents on a single machine than a Java Virtual Machine (JVM) can support threads.

Finally, fault-tolerance is concerned with state replication and error recovery. It handles not only agents, but other Jason (EE) components as well, including *Execution control* and *Environment* [5]. Whenever a state of an object changes, it is replicated to a predefined number of nodes. In case the host node fails, the object is transparently restored on one of the remaining nodes and all calls to it are redirected there.

Along with these advantages comes one disadvantage. Since Java EE is more complicated than Java SE, the process of developing Jason EE applications is inherently more complicated, when compared to the process of developing standard Jason applications.

The idea of executing Jason agents in Java EE environments was originally presented in [13]. While it only dealt with mapping agents to *Enterprise JavaBeans*, this paper proposes Jason EE as a fully-featured redesign of Jason, suitable for enterprise environments.

The rest of the paper is organized as follows. Section 2 provides more details about AgentSpeak and Jason, as well as XJAF and *Enterprise JavaBeans*. Detailed insight into the Jason EE architecture and its components is given in Section 3. Section 4 presents a case-study that demonstrates one important technical advantage of Jason EE: state replication and failover. Related work is presented in Section 5, while the final conclusions and future research directions are given in Section 6.

2 Technology overview

In order to fully understand the architecture of Jason EE, some basic understanding of its underlying technologies is needed. This section describes AgentSpeak and Jason, as well as *Enterprise JavaBeans*, one of the core Java EE technologies, and XJAF, a multiagent framework which provides the necessary infrastructure for Jason agents.

2.1 AgentSpeak and Jason

The syntax of AgentSpeak is strongly inspired by Prolog. Its main data type is the *term*, which can be a constant (an atom, a number, or a string), a variable, a structure, or a list [7, 5]. However, AgentSpeak also includes a variety of new syntactical (and semantical) elements, in order to simplify the development of goal-oriented reasoning agents.

AgentSpeak agents are defined in terms of their beliefs, goals, and plans [4, 5]. The agent's belief base consists of *predicates* and *rules*. For example, the predicate `ball(8,32)` might represent a belief of a football playing agent that the ball is at position (8,32). The rule `canKick :- me(X,Y) & ball(A,B) & dist(X,Y,A,B) < 1` might indicate that the agent can kick the ball if the distance between itself and the ball is less than some predefined value.

More information about a belief can be provided using *annotations*. Annotation is a user-defined or a built-in structure attached to the predicate. For example, `ball(8,32)[source(percept)]` indicates that the agent's belief about the ball's current position stems from the perceptual information (i.e. the agent has directly observed the ball).

AgentSpeak supports two types of negations: *strong negation*, and *negation as failure* [5]. The first type, denoted by ~, indicates that the agent explicitly believes something no to be true. In the second type, a belief preceded by the `not` operator is true if it cannot be deduced from the agent's belief base.

The language supports two types of goals: *achievement* and *test* [4, 5]. Achievement goals are expressed as logical formulae describing the state of affairs the agent would like to reach. Test goals, on the other hand, are typically used to query the belief base, and determine if certain beliefs exist.

Beliefs and goals together describe the agent's mental state. Changes in the mental state, i.e. additions or removals of beliefs and goals, trigger the execution of *plans*. A plan definition consists of a *triggering event*, a *context*, and the plan *body* [5]. Whenever the agent's mental state changes, all plans with the corresponding triggering event are marked as *relevant*. The context of each relevant plan is then evaluated in order to determine if the plan is *applicable*. The context is a logical expression which describes beliefs, e.g. about the environment, that the agent must hold, and is especially useful in dynamic environments. The final element of a plan, its body, is defined as a sequence of simple logical expressions, internal or environment actions, test or achievement goals, as well as mental notes, which add new beliefs to the belief base [5].

Jason interpreter operates in *reasoning cycles*, where each cycle consists of ten steps [5]. First, the agent perceives its environment, processes a single message received from another agent, filtering out "socially unacceptable" messages along the way, and updates its belief base accordingly. The remaining six steps represent the core of agent's reasoning and acting:

– A single event is selected to be processed;

– A set of relevant plans, i.e. plans corresponding to that event, is selected;

– Of those, a set of applicable plans (or, *options*) is determined;

– Committing to an applicable plan, creating an *intention*;

– Selecting an intention from a stack of pending intentions; and

– Executing one step of the selected intention.

Users can modify the interpreter's behavior in many of these steps. For example, the applicable plan will be selected (for the agent to commit to executing it) based on its order in the source file (similarly as in Prolog). This behavior can be changed by modifying the corresponding selection function.

The work presented in this paper deals with a different aspect of modifying Jason. The goal is to enable AgentSpeak/Jason agents to operate in enterprise applications, by providing customized multiagent infrastructure, agent architecture, execution control, and environment, as discussed in Section 3.

2.2 Enterprise JavaBeans

Enterprise JavaBeans (*EJBs*, or simply, *beans*) represent one of the core Java EE technologies. They are server-side components that encompass the business logic of an application. An EJB is described as a *managed* component, as its life-cycle, concurrent access, transactional integrity, etc. is controlled by an enterprise application server.

There exist two main categories of EJBs: *message-driven* and *session*. Message-driven beans act as message receivers in the context of the *Java Message Service* (JMS)[1], an additional Java EE technology for asynchronous communication. Session beans are further classified as *singleton*, *stateless*, and *stateful*.

As its name suggests, there is only a single instance of a singleton session bean per Java Virtual Machine (JVM). Stateless session beans do not preserve conversational state with the client, and are best-suited for operations that can be performed in a single method call. Stateful sessions beans, on the other hand, are used when the client requires an ongoing, more complex conversation.

From the point of view of runtime efficiency, the best performance is achieved with stateless beans. When a client request is received, the enterprise application server can freely construct a new stateless bean on any cluster node. Once the request is handled, the instance is destroyed. Alternatively, instead of constructing and destroying stateless bean instances with each request, the server can be configured to recycle them from a *pool*. In any case, the advantage of using stateless beans is that it becomes relatively easy to implement load-balancing techniques in order to efficiently handle large numbers concurrent requests.

Although load-balancing of client requests that target stateful beans is also applicable, here, the more important focus is on *state replication and failover*. The server maintains multiple copies of a stateful bean across the cluster. In case of a node failure, client requests are transparently redirected to an instance residing in one of the remaining nodes. This functionality enables the development of highly-available systems, i.e. systems resilient to software and hardware failures.

As discussed next, session EJBs, mainly stateless and stateful, can be used to represent agents in enterprise applications. Stateless beans have a more restricted applicability, and are well-suited for "one-off" agents – agents that perform one task, and then terminate. More complex behaviors, like those exhibited by Jason agents, can be achieved by mapping agents to stateful beans.

2.3 XJAF

XJAF is an enterprise-scale multiagent framework [16, 24, 14]. One of the goals behind the development of XJAF was to show how Java EE technologies can be utilized to easily provide many functional requirements of multiagent platforms. By integrating enterprise technologies and agents, XJAF could assist in bridging the gap between the two approaches to distributed software development.

It is designed as an customizable architecture, in the sense that its core components, called *managers*, are recognized and used solely by their interfaces, while the implementation details can change. Each manager is in charge of handling a distinct part of the overall architecture. Over time, several managers have been in use, but the latest version [16] includes three: agent, message, and connection manager.

The agent manager acts as an agent directory and controls the agents' life-cycles. It represents each agent as an EJB component, and passes it to the underlying enterprise application server. The server is then in charge of managing the concurrent access, maintaining the state integrity of agents, fault-tolerance, etc. Internally, the agent directory is managed using the *Java Naming and Directory Interface* (JNDI)[2].

The message manager is in charge of transporting and delivering messages. By default, messages are processed asynchronously. Several utility functions, such as blocking the receiving of a message, are provided as well, in order to simplify the agent development. For most of its functionality, the message manager relies on the Java Message Service technology described earlier.

The latest version of XJAF, available at the XJAF home-

[1]`http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html`, retrieved on October 15, 2014.

[2]`http://docs.oracle.com/javase/tutorial/jndi/`, retrieved on October 15, 2014.

page[3], is focused on clustered computing and exploiting its many benefits [16]. It operates in *symmetric* clusters, where each node is connected to and aware of every other node. A single node is recognized as the *master*, while the others are called *slaves*. The only difference between the master and a slave is that the master can be used to remotely control the cluster: start or stop slaves, deploy applications, etc.

Finally, the connection manager is in charge of connecting physically distributed XJAF clusters in a single computational framework. Basically, it creates another virtual cluster formed of master nodes. As all parts of the XJAF, the connection manager relies on another Java EE technology for its functioning, *JGroups*[4].

In the context of this paper, XJAF acts as an underlying infrastructure for Jason EE agents. The details of integrating Jason and XJAF/Java EE concepts are described in details in the next section.

3 Jason EE

Jason is designed as a highly-customizable system. Users can modify not only the selection functions mentioned earlier, but also some of the interpreter's core components. The most important customizable components include *Agent architecture*, *Execution control*, and *Environment* [5]. Agent architecture represents the agent's "physical body" [5]. It enables the agent to perceive its environment and act upon it, and also to send and receive messages. The main function of Execution control is to synchronize the reasoning cycles of individual agents, while the Environment component serves as the basis for simulating real-world or artificial environments.

Another important concept in Jason is the *infrastructure* [5]. Infrastructure refers to a multiagent platform that actually hosts the agents, carries out the transmission and delivery of messages, etc. By default, two infrastructures are provided: *Centralised* and *JADE*. Figure 1 depicts main components of Jason and shows how the Centralised infrastructure binds them together[5].

Jason EE provides new versions of Jason's core components, and integrates them with an XJAF-based infrastructure, as described in the rest of this section.

3.1 Jason EE components

Unfortunately, Jason EE and Jason are not (yet) fully compatible, due to various technical and "philosophical" differences between Java EE and Java SE. For example, Java EE components should never directly create and use threads,

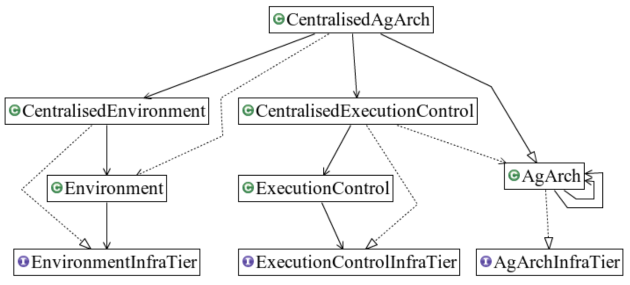

Figure 1: An overview of the main Jason components and the Centralised infrastructure that binds them together. The tight coupling and interdependencies of components might provide some difficulties in understanding the interpreter at first.

since these cannot be managed by the enterprise application server. In addition, the distributed nature of Java EE applications and the use of computer clusters pose additional requirements (e.g. component serialization).

With these differences in mind, the work of developing Jason EE consisted of three main tasks:

- Providing a new set of base classes for Agent architecture, Execution control, and Environment;
- Introducing a new set of components that support these base classes; and
- Integrating the developed architecture with XJAF, which acts as the underlying infrastructure.

The main components of Jason EE and their distribution are shown in Figure 2. The first noticeable difference from the (standard) Jason approach is in the way user-defined objects are loaded. By default, Jason relies on the Java *Reflection API* for user object construction. In case of Jason EE, however, this approach would not work. The enterprise application server views each user application as a distinct module, and loads it through a separate class-loader. This means that Jason EE components do not have direct access to classes defined in user applications. In order to resolve this issue, Jason EE introduces a new *Remote Object Factory* interface. The interface defines a set of methods that will be called whenever a user-defined object is to be created. Each user application needs to realize this interface in the form of a stateless session bean.

Agent architecture, Execution control and Environments are all deployed in the enterprise application server, and executed on top of a computer cluster. Since XJAF is used as the underlying infrastructure, the same approach of mapping agents to EJBs is used in Jason EE. This means that Jason EE agents exhibit the two important features described previously: automatic load-balancing, and state replication and failover.

[3]`https://github.com/gcvt/siebog`, retrieved on October 15, 2014.

[4]`http://www.jgroups.org/`, retrieved on October 15, 2014.

[5]All class diagrams in this paper have been generated from the source code of Jason 1.4.1 using *ObjectAid UML Explorer for Eclipse*: `http://www.objectaid.com/`, retrieved on September 19, 2014.

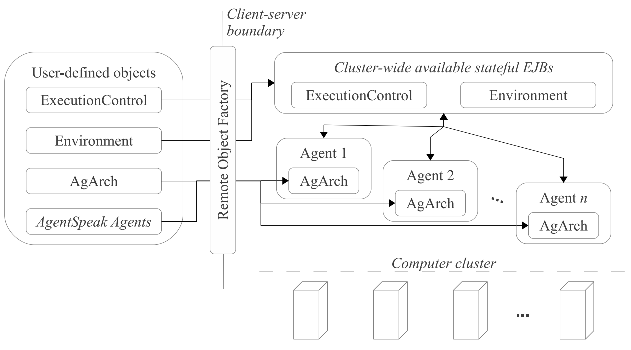

Figure 2: Organization and distribution of components in Jason EE-based applications.

3.2 Custom agent architectures

In Jason, users provide custom implementations of the Agent architecture by inheriting the base `AgArch` class (Figure 1). Some of its more important modifiable methods include:

- `perceive`: Perceives the environment, and returns the list of percepts;
- `act`: Performs the given action. The action does not have to be completed in the same method call, and the feedback (i.e. success or failure information) can be provided in one of the future reasoning cycles.
- `checkMail`, `sendMsg` and `broadcast`: Used for inter-agent communication.
- `sleep` and `wake`: Suspend and resume the agent's execution.

Jason EE provides a slightly different, intermediary class for the Agent architecture, in order to accommodate for the enterprise setting. The new class, `JasonEEAgArch`, modifies parts of the base `AgArch` class that are in charge of interaction with other components. For example, Execution control is an EJB in Jason EE, and the process of constructing and using EJBs is slightly different than constructing and using regular objects.

In order to connect the new Agent architecture class with the XJAF-based infrastructure, an additional component is provided. As shown Figure 2, `JasonEEAgArch` is embedded within an XJAF agent. The agent controls the architecture's life-cycle and also acts as a layer between the architecture and the remaining parts of the system. For example, it translates XJAF's FIPA ACL message format into Jason's KQML, but also controls the architecture's reasoning cycles, either directly in asynchronous, or indirectly in synchronous execution mode, as described later.

Jason EE agents are represented as EJB components, and are created through JNDI lookups. During the lookup phase, the enterprise application server will choose a node in the cluster to host the agent instance. From then on, the agent will have an affinity to that node, meaning that all invocations will be executed on it. However, whenever the agent's internal state changes, it will be copied to a predefined number of other nodes in the cluster. In case the current host fails, the agent will be restored on one of the remaining nodes, and continue its execution there.

Unfortunately, the base `AgArch` class in Jason, which is also used in Jason EE, is not fully serializable. This means that the agent's internal state, including, for example, the *transition system*, cannot be fully replicated across cluster nodes. If the agent's host fails, the agent will be transparently restored on one of the remaining nodes, but some of its parts will need to be re-initialized. The support for full state replication would require changes in the Jason interpreter itself. More details on this issue are given in Section 4.

3.3 Execution control

Jason supports two execution modes [5]: asynchronous and synchronous. In asynchronous mode, the agent executes its reasoning cycles continuously, regardless of the behavior of other agents. In synchronous mode, the agent can continue to the next reasoning cycle only after all other agents have completed the *current* reasoning cycles as well.

The execution control component is used in synchronous mode only. It maintains the list of active agents, and instructs them when to advance to the next reasoning cycle. This happens either when all agents complete the current reasoning cycle, or after a specified amount of time has passed (e.g. in case an agent has died unexpectedly). Users can provide their own, custom execution modes by inheriting the appropriate base class.

Having one central component that manages the execution of other components might not seem as a good design approach for distributed systems, such as Jason EE. However, the Execution control component can be thought of as a *synchronization barrier*. Barriers represent efficient synchronization approach when distributed or parallel processes need to operate in global computational steps [22]. In synchronous execution mode, Jason (EE) agent reaches the barrier after completing one reasoning cycle, and then waits until other agents have reached the barrier too.

The sequence diagram shown in Figure 3 outlines the execution of agent's reasoning cycles in Jason EE. Once the agent is created, it registers itself with the Execution control component. Subsequently, it will receive a signal to advance to the next reasoning cycle. When finished, the agents reports back to the Execution control. Once the condition is met (i.e. all agents have reported back, or the timeout has expired), the process is repeated.

The main technical difficulty in developing this Execution control for Jason EE is to properly design it for for clustered environments. That is, there should be a single instance of the component for the entire cluster, it should be easily accessible from any node, and should preferably exhibit state replication and failover. The singleton session

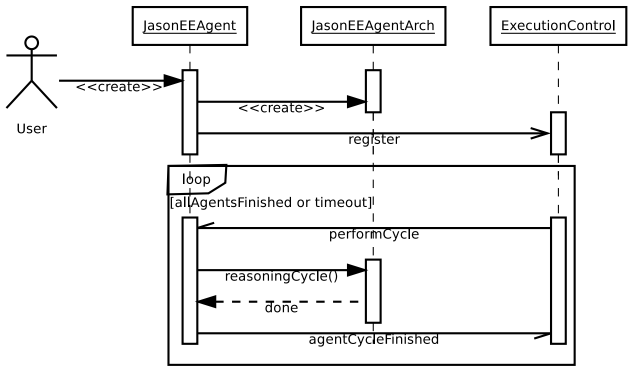

Figure 3: Synchronous execution of an agent managed by the Execution control.

EJB cannot be used, as there exists a single instance of this object per JVM.

The solution used in Jason EE is to create a single stateful EJB component and store it in a global, cluster-wide *Infinispan* cache [12] included in the *WildFly* enterprise application server[6]. As shown earlier in Figure 2, this Execution controls runs, figuratively speaking, on top of the entire cluster, supporting the aforementioned features.

End-users are offered an additional class, named `UserExecutionControl`, for customization. The Jason's standard class for this purpose, `ExecutionControl` (Figure 1), cannot be used in Jason EE, for two main reasons. First, the class is not serializable, and thus cannot be used in the state replication process. Secondly, the class creates and manages its own threads. Java EE applications are managed by the enterprise application server. The server needs the full control over the application's resources in order to secure scalability and high-availability. If an application creates its own threads, the server looses this control and the whole concept is undermined[7]. For these reasons, Jason EE provides the new base class, i.e. one that satisfies all the requirements and recommendations for Java EE applications.

3.4 Environment

The Environment component in Jason provides a model of a real-world or artificial environment [5]. It is strongly related to the Agent architecture, in the sense that the architecture can delegate perception and action execution to the Environment component. In addition, the Environment can exhibit "individualized perception" [5], and provide only a subset of percepts to an agent. This can be useful, for example, in evaluating how the agent performs under varying degrees of available information.

[6]`http://wildfly.org`, retrieved on October 15, 2014.

[7]The Java EE 7 specification defines so-called *managed executor services* which should be used by applications that need to spawn their own threads.

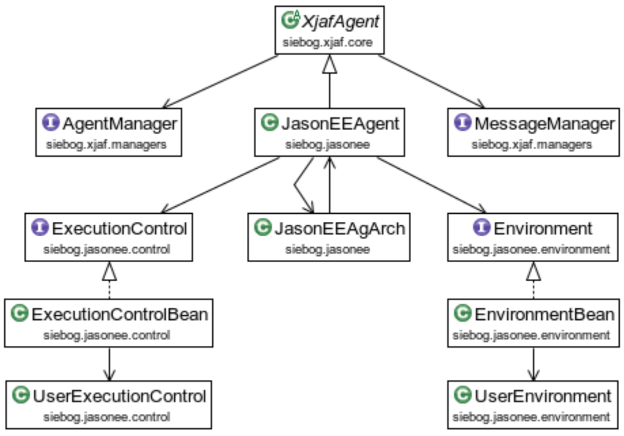

Figure 4: Diagram of the core classes in Jason EE. The connection to standard Jason is available through `JasonEEAgArch`, `ExecutionControl` and `Environment`, which are sub-types of, respectively, Jason's `AgArch`, `ExecutionControlInfraTier`, and `EnvironmentInfraTier` (not shown here for clarity).

Users provide custom environments by redefining methods of the base `Environment` class (Figure 1). All of its methods are dedicated to perception management (retrieval, removal, etc.), as well as action execution. For example:

- `getPercepts`: Returns the list of percepts for the given agent. The list will include only new percepts, i.e. percepts that have been obtained since the previous invocation.

- `scheduleExecution`: Performs an asynchronous execution of the provided action in the environment. The agent will be notified of the result later on, once the execution is completed, either successfully or unsuccessfully.

From the technical side, the Jason EE Environment is realized in a similar way as the Execution control component: in the form of a cluster-wide stateful EJB, with an additional `UserEnvironment` class for user customizations. The final class diagram of Jason EE is shown in Figure 4. Each deployed Jason EE application includes a single Environment instance, which is given a cluster-wide unique identifier and kept in a global cache. All agents of that particular application use the identifier in order to interact with the Environment. The same is true for Execution control.

The connection with XJAF is made through the `JasonEEAgent` component. Being a special XJAF agent, this component relies on the agent manager for creating and destroying other (Jason) agents, and on the message manager for sending and receiving messages.

The effectiveness of the proposed Jason EE architecture is demonstrated using two case-studies, described in details in the following section.

4 A case study

The case study presented in this section demonstrates state replication and failover in practice. Its purpose is to outline one of the technical advatages brought by Jason EE. The benefits of using AgentSpeak and Jason for complex agent development are presented in e.g. [23]. For XJAF agent load-balancing in computer clusters, see [16]. The full source of this case study is available at the XJAF homepage[8].

The case-study includes two highly-available agents. Each agent has a single belief: a list of strictly monotonically increasing numbers. The agent periodically prepends a number $n = h + 1$ to the list, where h is the current head. Listing 1 shows the AgentSpeak source code of the agent.

Listing 1: AgentSpeak source code of the highly-available agent used in the case study.

```
numbers([0]). // inital belief
!addNextNum. // initial goal

+!addNextNum : true <-
 ?numbers([OldHead | Tail]);
 NewHead = OldHead + 1;
 NewList = [NewHead, OldHead | Tail];
 -+numbers(NewList);
 printList(NewList);
 !!addNextNum.
```

The case-study also includes a user-defined Agent architecture and a user-defined Execution control. The architecture is capable of executing the `printList` action shown in Listing 1, while the user-defined Execution control simply outputs the current reasoning cycle. These components were developed primarily in order to show how the state replication and failover work with other Jason EE components as well (and not just agents).

A cluster with two virtual nodes, a master and a slave, was setup, and the load-balancer was configured to put both agents and the Execution control component on the slave node. The project was then executed, yielding in the following possible output (filtered for clarity):

```
Cycle 0 on node slave@192.168.213.129
...
agent0 on slave@192.168.213.129: [1,0]
agent1 on slave@192.168.213.129: [1,0]
Cycle 7 on node slave@192.168.213.129
...
agent0 on slave@192.168.213.129: [2,1,0]
agent1 on slave@192.168.213.129: [2,1,0]
Cycle 15 on node slave@192.168.213.129
...
```

The slave node was then forcibly terminated. In response, all components from the slave have been automatically restored on the master node, and continued to operate as follows:

```
Cycle 63 on node master:xjaf-master
...
agent0 on master:xjaf-master: [1,0]
agent1 on master:xjaf-master: [1,0]
Cycle 68 on node master:xjaf-master
...
```

Here, it can be seen that both the agents and the Execution control have successfully continued their execution on the remaining node, confirming that the state replication and failover in Jason EE works as expected.

However, while the Execution control's internal state was successfully restored, the belief base of each agent has been reset. As noted earlier, some of the important Jason components, such as the Agent architecture and its transition system are not serializable. Since these are used in Jason EE as well, the agent's internal state cannot be fully replicated. Currently, Jason EE detects this issue and emits a warning, but the full support for the agent state replication requires changes in the Jason implementation.

5 Related work

In general, there are two main approaches for writing software agents. The first one is to use an existing programming language, such as Java. For example, in JADE the process of writing agents consists of inheriting the proper base classes [1]. More recently, source code annotations have been proposed as a convenient approach for developing BDI agents [18]. Extensions to the Java programming language have been proposed as well [26]. Main advantages of these approaches are a flatter learning curve and the availability of existing programming libraries and tools. The main disadvantage is that the agent source code is "cluttered" with object-oriented programming constructs.

The second approach is to use dedicated, agent-oriented programming languages. These languages offer programming abstractions that enable straightforward implementations of advanced multiagent concepts, and hide the overall complexity of writing intelligent agents. As a result, agent developers can focus on solving the problem in question, rather than dealing with class inheritance and method overriding.

Over time, a plethora of agent-oriented programming languages has been developed [6, 3]. Among the most recent is ASTRA[9], which combines AgentSpeak and *Teleo-Reactive* functions [9, 17]. Among the well-established languages, besides AgentSpeak, the two notable examples are *Goal-Oriented Agent Language* (GOAL)[10], and *A Practical Agent Programming Language* (2APL) [8].

In addition to achievement, GOAL adds support for *maintenance* goals [10]. An agent uses maintenance goals to refrain itself from acting, and to keep the current state of affairs. Agents generally follow the *blind commitment*

[8]`https://github.com/gcvt/siebog`, retrieved on October 15, 2014.

[9]`http://www.astralanguage.com/`, retrieved on October 15, 2014.

strategy [10, 20]: an active goal will not be dropped until it is fully completed. Actions are mostly user-provided, and are guarded by preconditions and postconditions. Action execution strategy is determined by *action rules*. An action rule defines a mental state that has to hold before the corresponding action can be considered as a candidate for execution.

2APL combines declarative and imperative programming styles [8]. It offers several important agent-oriented programming concepts, including beliefs, goals, events, actions, and plans. Belief and goals are declarative constructs that describe the agent's mental state. Events carry information about some change in the environment, and can trigger the plan execution. Actions describe agent's capabilities, and are divided into six categories, including *belief updates*, *goal updates*, and *abstract actions* which act as procedure calls. Finally, a plan consists of a sequence of actions, with the addition of conditional statements, loops, and non-interleaving operators for building atomic plans.

Each of these languages represents a powerful tool for developing intelligent agents. One of the main reasons AgentSpeak was selected for the work presented in this paper is its practical interpreter Jason. As shown, Jason is highly customizable and portable, allowing AgentSpeak agents to be executed on different multiagent platforms and environments. It is worth noting that Jason is not the only interpreter for AgentSpeak. For example, *AF-AgentSpeak* is an implementation of the language for the versatile *Agent Factory* framework[10].

In addition to Jason EE, two Jason infrastructures are available. The Centralised infrastructure is a lightweight and an efficient (performance-wise) solution, but designed for single-machine deployments only. The JADE infrastructure uses JADE as the underlying multiagent platform. Therefore, it provides all the features available in JADE, including distributed execution and platform fault-tolerance [1]. Jason EE implementation is to a certain degree based on these solutions. Its main advantages include state replication and failover demonstrated in Section 4, which is more advanced than the one offered by JADE, and automated agent clustering and load-balancing shown in [16].

6 Conclusions and future work

Jason EE represents an enterprise-scale agent development framework. It combines Java EE, one of the most widely-used software development platform, with AgentSpeak and Jason, a popular agent-oriented programming language and its interpreter, designed for writing complex, reasoning agents.

As discussed throughout the paper, Jason EE brings several important benefits over standard Jason, as well as other similar solutions. On the technical side, the underlying enterprise application server manages the application resources, and provides advanced programming features, such as agent load-balancing, scalability, and fault-tolerance. This enables agent developers to focus on solving the problem at hand, without having to bother with technical difficulties. The end-goal of the presented research, however, is to enable seamless integration of intelligent agents in modern enterprise applications, and to bridge the gap between the two approaches to distributed software development.

For the future, several research and development directions of Jason EE are planned. First of all, as discussed in Section 3, changes in the Jason implementation itself are required. These would allow not only the complete state replication and failover of agents, but also the full portability of Jason and Jason EE applications. In the longer run, the remaining two components of the *JaCaMo* project will be re-designed for enterprise environments: the *CArtAgO* artifacts modeling framework, and the *Moise* framework for virtual multiagent organizations [2].

[10]`http://www.agentfactory.com/`, retrieved on October 15, 2014.

Acknowledgement

This work was partially supported by the Ministry of Education, Science and Technological Development of the Republic of Serbia, through project no. OI174023: "Intelligent techniques and their integration into wide-spectrum decision support."

References

[1] F. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, 2007.

[2] O. Boissier, R. H. Bordini, J. F. Hubner, A. Ricci, and A. Santi. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 78(6):747–761, 2013.

[3] R. H. Bordini, L. Braubach, M. Dastani, A. E. Fallah-Seghrouchni, J. J. Gomez-Sanz, J. Leite, G. M. P. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica (Slovenia)*, 30(1):33–44, 2006.

[4] R. H. Bordini and J. F. Hubner. BDI agent programming in AgentSpeak using Jason. In F. Toni and P. Torroni, editors, *Computational Logic in Multi-Agent Systems*, volume 3900 of *Lecture Notes in Computer Science*, pages 143–164. Springer Berlin Heidelberg, 2006.

[5] R. H. Bordini, J. F. Hubner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley Series in Agent Technology. John Wiley & Sons Ltd, 2007.

[6] C. Bădică, Z. Budimac, H.-D. Burkhard, and M. Ivanović. Software agents: Languages, tools, platforms. *Computer Science and Information Systems*, 8(2):255–298, 2011.

[7] W. F. Clocksin and C. S. Mellish. *Programming in Prolog: using the ISO standard*. Springer, 5 edition, 2003.

[8] M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[9] A. Dhaon and R. Collier. Multiple inheritance in AgentSpeak(L)-style programming languages. In *Proceedings of the 4th International Workshop on Programming based on Actors, Agents and Decentralized Control*. 2014.

[10] K. V. Hindriks. Programming rational agents in GOAL. In A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini, editors, *Multi-Agent Programming: Languages, Tools and Applications*, pages 119–157. Springer US, 2009.

[11] Java EE at a glance. `http://www.oracle.com/technetwork/java/javaee/overview/index.html`. Retrieved on October 15, 2014.

[12] F. Marchioni and M. Surtani. *Infinispan data grid platform*. Packt Publishing Ltd., 2012.

[13] D. Mitrović, M. Ivanović, and C. Bădică. Jason agents in Java EE environments. In E. Petre and M. Brezovan, editors, *3rd Workshop on Applications of Software Agents (WASA 2013), held within 17th International Conference on System Theory, Control and Computing (ICSTCC 2013)*, pages 768–771, Sinaia, Romania, October 2013.

[14] D. Mitrović, M. Ivanović, Z. Budimac, and M. Vidaković. Supporting heterogeneous agent mobility with ALAS. *Computer Science and Information Systems*, 9(3):1203–1229, 2012.

[15] D. Mitrović, M. Ivanović, and Z. Geler. Agent-based distributed computing for dynamic networks. *Information Technology and Control*, 43(1):88–97, 2014.

[16] D. Mitrović, M. Ivanović, M. Vidaković, and Z. Budimac. Extensible Java EE-based agent framework in clustered environments. In J. Mueller, M. Weyrich, and A. L. C. Bazzan, editors, *12th German Conference on Multiagent System Technologies*, volume 8732 of *Lecture Notes in Computer Science*, pages 202–215. Springer International Publishing, 2014.

[17] N. J. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.

[18] A. Pokahr, L. Braubach, C. Haubeck, and J. Ladiges. Programming BDI agents with pure Java. In J. P. Müller, M. Weyrich, and A. L. C. Bazzan, editors, *Multiagent System Technologies*, volume 8732 of *Lecture Notes in Computer Science*, pages 216–233. Springer International Publishing, 2014.

[19] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. de Velde and J. Perrame, editors, *Agents Breaking Away: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world (MAAMAW '96)*, volume 1038 of *Lecture Notes in Artificial Intelligence*, pages 42–55. Springer-Verlag, 1996.

[20] A. S. Rao and M. P. Georgeff. Intentions and rational commitment. Technical Report 8, Australian Artificial Intelligence Institute, 1993.

[21] A. S. Rao and M. P. Georgeff. BDI agents: from theory to practice. In V. Lesser and L. Gasser, editors, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*, pages 312–319, 1995.

[22] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[23] S. Vester, N. S. Boss, A. S. Jensen, and J. Villadsen. Improving multi-agent systems using Jason. *Annals of Mathematics and Artificial Intelligence*, 61(4):297–307, April 2011.

[24] M. Vidaković, M. Ivanović, D. Mitrović, and Z. Budimac. Extensible Java EE-based agent framework – past, present, future. In M. Ganzha and L. C. Jain, editors, *Multiagent Systems and Applications*, volume 45 of *Intelligent Systems Reference Library*, pages 55–88. Springer Berlin Heidelberg, 2013.

[25] WildFly 8 high availability guide. `https://docs.jboss.org/author/display/WFLY8/High+Availability+Guide`. Retrieved on October 15, 2014.

[26] M. Winikoff. Jack intelligent agents: An industrial strength platform. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 175–193. Springer US, 2005.