A Reference Implementation of ADF (Agent Developing Framework): Semantic Web-based Agent Communication

Cătălin Hriţcu
"A. I. Cuza University of Iaşi
Faculty of Computer Science
Berthelot, 16 – Iaşi, Romania
Catalin.Hritcu@gmail.com

Sabin C. Buraga
"A. I. Cuza University of Iaşi
Faculty of Computer Science
Berthelot, 16 – Iaşi, Romania
busaco@infoiasi.ro

Abstract

The purpose of this paper is to present the main characteristics of ADF, an open source agent developing platform with a focus on agent collaboration. The basic architecture of the platform is investigated, and a reference implementation – based on Java 2 Enterprise Edition – is presented. Two short case studies (a reverse auction system and meeting scheduding multi-agent system), that use current semantic Web technologies (such as RDF and OWL), illustrate the main features of the framework.

1. Introduction

Collaborative software agents provide developers with a way of structuring an application around autonomous entities, that communicate in order to achieve their internal goals. They offer a new and convenient design metaphor for the development of complex software systems, that operate in open and dynamic environments.

Agent frameworks provide a foundation for developing complex systems using the agent paradigm. Instead of executing low-level tasks like building naming, location or directory services, inventing communication protocols, mobility mechanisms or cryptographic algorithms, agent developers can concentrate on their particular problems and on the logic of the agent-oriented applications solving them. There have been many attempts to build multi-agent frameworks, in both the research and business communities. Some of them were quite successful and are still actively maintained – e.g., Cougaar [21], DIET Agents [23], JADE [28], Voyager [38] or ZEUS [42] –, while other did not stand the test of time, but nevertheless had an important contribution to the agents field – for example, Aglets [19], Ajanta [20], D'Agents [22], FIPA-OS [24], or Omega [1].

Our paper presents the *Agent Developing Framework* (*ADF*) [11], an open source agent platform, with a stated emphasis on collaboration. What distinguishes ADF from many other agent frameworks is, also, the pragmatic engineering approach, starting by the focus on technology and component reuse rather than reinvention, and emphasis on real-world scenarios and on working code.

There are many difficult problems to be solved when building multi-agent systems [14], and we have chosen to concentrate most of our work on a very basic one: the *communication problem*. What makes the communication problem tangible, is that many pieces of the puzzle seem to be present somewhere, in one form or another, and only need to be put in place.

For two agents to communicate, there is the need for a common transport protocol, a common communication language and a common understanding of the terms in use (for example, a common ontology).

Current Web technologies already offer a solution for transporting messages over any physical transport protocol by the means of SOAP (Simple Object Access Protocol) [35] – previous research work is presented in [2] and [5]. RDF (Resource Description Framework) [15] is a flexible knowlege representation language and – together with Web Ontology Language (OWL) [8] – it is quite close to solving the ontology problem, at least in the case of task- and domain-specific ontologies. An agent communication language (ACL) is already standardized by FIPA [37] and can be flexibly encoded, according to the particular needs of the application. Since all these technologies are XML-based (or XML-capable), the Extensible Markup Language (XML) [4] family is the ligand that makes all the parts fit together in the ADF communication. However, we will see that ADF is much more flexible in the case of communication. Older, but still valuable technologies, can work hand in hand with the state of the art ones.

This paper is structured into five sections. In the next section, we introduce the main goals of ADF and we present its overall architecture and the technologies that allowed us to achieve them. Section 3 is dedicated to agent commu-

nication and discusses the transport protocols, message encodings, content languages and interaction protocols used within ADF. Section 4 describes two simple semantic Web-based agent systems built with ADF: a reverse auction system and a meeting scheduling system. The final section briefly summarizes the discussed issues and presents several directions of future improvement for the ADF framework.

2. Agent Developing Framework

The ADF project was started more than one year ago, with the design of an abstract architecture for the framework [13]. The next logical step was to build a concrete reference implementation of that abstract design. However, as progress was made on this implementation, the architecture was redefined in order to make it more flexible, scalable and easily extensible. Actually, ADF [11] can be freely obtained as open source code, under the terms of the GNU LGPL.

The following sections will describe the most important goals, the general architecture, and the current implementation of the ADF system.

2.1. Goals

The purpose of ADF is to build an interoperable, flexible, scalable and extensible framework that will allow real-world multi-agent systems to be developed with less effort. The goals of ADF were originally presented in [13] and then they were successively refined:

Interoperability We intend ADF agents to be able to easily communicate and interact with entities in other systems (even with those that were not foreseen during the original development), whether they are agents themselves, or more traditional applications. In our view, using open standards is the only possible way to achieve true interoperability in a large-scale, thus heterogeneous, distributed environment.

Flexibility and Extensibility We aim to make it easy for developers to add new features, or reimplement existing ones using different technologies, without impacting the operation of the system as a whole. When designing ADF, evolution was regarded as something positive and inevitable, that was planed for in advance. Programming to standard interfaces and not to implementations, separating policy from mechanism, orthogonality and loose coupling between interacting modules were several of the mechanisms we used to ensure the flexibility and extensibility of the ADF framework.

Scalability The number of ADF agents the multi-agent system can accommodate, while maintaining an acceptable performance, should not be limited by centralization, whether we refer to centralized data, services or algorithms. Additionally, agents that are very far from one another, should still are able to interact efficiently by using asynchronous communication.

Platform independence We decided to make ADF independent of the particular hardware configuration, operating system or application server and to model all the dependencies to third-party components through standard interfaces. Therefore, we save a great amount of developing time, while having excellent portability for the framework.

Easy to use No matter how complex the internals of ADF system are, this complexity is hidden form the users behind a simple and intuitive API. Programmers are able to build simple agents easily, even if they do not have much knowledge about the characteristics of framework, and then they are able to evolve their skills gradually, as they use new and more advanced features. Our programming model is similar to that adopted by JADE [28], which we find simple, yet powerful.

Pragmatism While the theoretical foundations of ADF are surely important, even more important for us is the practical applicability of the theoretical methods in the actual implementation. Because we want our framework to be really useful, we always try to ensure everything works well by writing tests. And although simple test cases are a good start, the actual implementation of "real-world" scenarios is even better. Another aspect, we find very important, is reusing mature existing technologies and synthesizing existing research, which is more effective and more honorable than reinventing the "wheel", or worse, "a flat tire". Only when the existing "wheels" were missing or broken, did we invent new ones, making the platform become greater than the sum of its parts. This pragmatic approach is similar to that of the Zeus Agent Toolkit [42].

2.2. Architecture Overview

ADF is built as a Service-Oriented Architecture (SOA) [9, 12], which assures loose coupling between the interacting software agents. Agents consume the services provided by other agents, in order to be able to provide their own specialized services. This allows every agent to specialize only in several tasks it does very well, while delegating the other tasks to other expert agents, thus leading to the separation of concerns. A flexible mechanism permits a provider to register its services, and a potential consumer to discover the providers that offer the

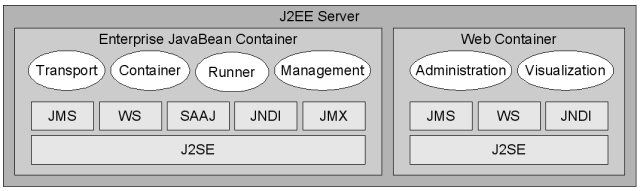

Figure 1. An Overview of the ADF Platform

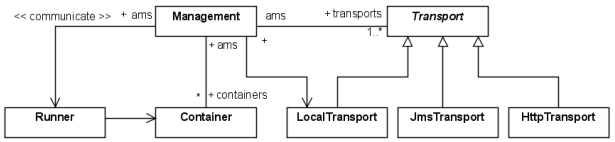

Figure 2. ADF Agent Management System

services it needs. Both registration and discovery happen dynamically, at runtime.

Loose coupling is obtained in a SOA by defining a small set of simple generic interfaces that are universally available to the participating software agents. The interaction between agents is done by exchanging descriptive messages through the standard interfaces. The messages are constrained by an extensible schema, thus allowing new versions of services to be introduced without breaking existing ones [9]. This approach reduces artificial dependencies and is different from that of object oriented programming, which suggests that data and its processing should be bound together.

Since large-scale distributed environments are inherently heterogeneous, there are few generic interfaces universally available. ADF uses the simple framework provided by SOAP [35] in order to transport descriptive ACL messages over any physical transport protocol, and particularly over the ubiquitous HyperText Transfer Protocol (HTTP). SOAP protocol allows for very rich conversational patterns, where the semantics are at the level of the sending and receiving agents.

The ADF architecture is also compatible with the FIPA abstract reference model [37], which ensures interoperability with other FIPA compliant agent platforms such as JADE [28] and FIPA-OS [24]. This reference model mandates the basic services provided by the agent platform itself, such as creating, registering and deleting agents, facilities to locate agents and services, and services for inter-agent message-oriented communication.

2.3. Actual Implementation

ADF is implemented as a Java 2 Platform, Enterprise Edition (J2EE) application [27], and makes extensive use of many proved technologies, services and standard APIs provided by the J2EE environment. ADF was implemented and tested using JBoss 4 [30].

The ADF application is a collection of enterprise components (Enterprise JavaBeans – such as *ManagementBean*, *ContainerBean*, *LocalTransportBean*, etc. – and Java Servlets) that work together in order to provide the high level features that make up a multi-agent platform. Figure 1 provides an overview of the most important components of the ADF system.

The agent management system is the central component of the every FIPA-compliant agent platform, that keeps track of the agents in the associated platform and provides services for agent creation and termination. Parameters can be provided when an agent is created, and the results of its work can be easily obtained once the agent is finished. The agent management system also offers a white pages service, that allows an agent to be looked up by name in order to obtain a reference to its container or a list of communication endpoints.

The agent management system is implemented is ADF as a stateless session bean (see Figure 2). This allows multiple concurrent requests to be services by several equivalent instances of this bean, a mechanism that ensures vertical scalability, and is intensively used in ADF. The agent management system is used not only internally by most other components of the platform, but also by application clients (e.g., Java Servlets) in order to perform administrative tasks on behalf of a user, so servicing multiple requests simultaneously was a stringent necessity. And although the management bean itself is stateless, it uses the Java Naming and Directory Interface (JNDI) to store and retrieve vital information about the agents registered with the platform or the different transport protocols that are simultaneously used by the platform.

The agent container is responsible for holding an agent and providing it controlled access to the basic services of the framework. The agent container is the only entity in the whole system to hold a reference to the agent it contains and it uses it to manage its life-cycle. This way, the agent's methods can only be called by the agent itself and its container, thus *guaranteeing the agent autonomy*. The agent is not just a mere object, because it is has complete control over its own life-cycle. The only exception to this general rule is when the agent management system decides to immediately terminate the agent. This may happen because the agent has not respected an important policy (e.g., the security policy) or when the owner of the agent has explicitly asked for its termination (for example, when the agent is no longer responding to messages). Please note, however, that this is just an exception, and the general rule is that the agent

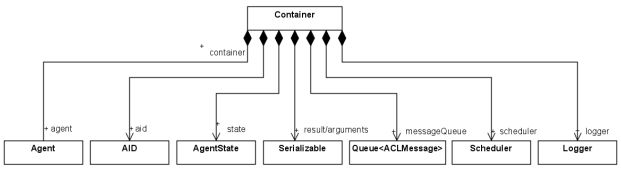

Figure 3. The ADF Agent Container

executes autonomously for as long as it takes for it to complete its tasks.

The agent container functions as a façade, that hides all the functionality of the framework from the agent, and offers a simple interface to the agent. This interface is generic and has no dependencies on other libraries or on the J2EE API, so that agents are not only simple to write, but the whole agent framework could be reimplemented using any other technologies without affecting the already implemented agent systems in any way. The agent container functions in fact as a façade the other way around too, thus it hides the agent from the other entities in the multi-agent system, in order to assure its autonomy.

The agent container is implemented as a stateful session bean, that holds the only reference to the agent and references to many other, very important objects (see Figure 3): the agent's identifier, state, arguments, results, message queue, task scheduler, and logger.

As previously stated, the agent container has a crucial role in managing the life-cycle of the agent (Figure 4), and one aspect of that is task scheduling. Dividing the work an agent has to complete into small tasks allows the agent code to run non-preemptively into a single thread and, at the same time, be responsive to asynchronous events, such as receiving a message or a notification of an expired timer. The task scheduling model in ADF is similar to that adopted by JADE [28].

The agent container also provides access for the agent to asynchronous messaging, by holding a message queue, where the agent can receive messages and also forwarding the messages sent by the agent to the corresponding transports. This mechanism is further examined in the next section.

Finally, one very important component of ADF is "the runner", a message-driven bean that is used internally by the framework, in order to allow agents to run concurrently. The bean is a simplification of the Service Activator EJB design pattern [3] and allows the invocation of EJB components in an asynchronous manner. And because the container is stateful and it cannot handle multiple concurrent invocations, it is never invoked directly but only through synchronization proxies created by the management bean.

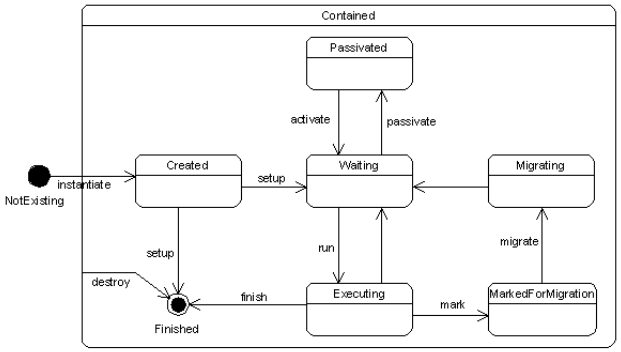

Figure 4. Lifecycle of an ADF Agent

3. Agent Communication

3.1. Overview

In most cases, agents have to collaborate with each other, in order to perform the tasks for which they are responsible. One important characteristic of software agents is that they communicate by means of an application-level communication protocol, which is referred to as an *agent communication language* (ACL), such as FIPA ACL or the Knowledge Query and Manipulation Language (KQML) [33].

ACLs rely on the speech act theory [18]. Agents perform *speech acts*, based on the relevance of their expected outcome (rational effect) in relation to their own goals. However, agents cannot assume that the rational effect will inevitably result from just sending a message, because it depends on other agents, which are themselves autonomous.

In an ACL, a strict separation is made between the contents of the message and its purpose, also known as the *performative* or the *communicative act*. The set of possible performatives is limited and their meaning is specified by the agent communication language and known by all the agents using the it. The content of the message is chosen by the participants to a conversation, and the use of task- or domain-oriented ontologies is very common in order to assure a common understanding of the symbols employed.

Agent communication in ADF is message-oriented and follows the standard model mandated by FIPA. ADF agents communicate by asynchronously exchanging messages by the means of a message transport service. When a message is sent, it is first encoded using a representation appropriate for the transport. A major concern when designing ADF was the orthogonality between four different aspects: the physical transport protocol, the ACL message encoding, the message contents, and the interaction protocol. This not only assures much flexibility for the moment, but also provides a maximum degree of extensibility.

3.2. Transport Protocols

Messages are physically transported from one agent platform to the other by using a transport protocol. The actual ADF implementation currently provides two transport techniques: one relying on HTTP, and the other using the Java Message Service (JMS) [16, 29] API.

HTTP is especially appealing, because it uses port 80 that is usually open, even in enterprises with a tight security policy, in order to allow Web browsing and, more recently, Web services. HTTP is not limited in any way to HTML, XML or SOAP, so we have actually abstracted from the encoding of the message itself, and built a generic HTTP transport. This component comprises a Java Servlet that receives messages and a stateless session bean that sends them.

JMS is a standard API that provides access to systems offering persistent asynchronous communication, traditionally named message oriented middleware (MOM). With persistent communication the transmitted messages are stored by the communication system as long as it takes, in order to deliver them to the receiver. Neither the sender, nor the receiver have to be up and running for message transmission to occur. Persistence also increases the reliability of the communication and enforces loose-coupling, by eliminating the timing dependencies between the communicating parties. The major disadvantage of JMS is that it is hard to use outside an administrative domain, because the underlying protocols are generally proprietary. The JMS transport in ADF uses this standard interface without knowledge of the implementation details of JMS provider, and relies on a message-driven bean that asynchronously receives messages and dispatches them to the corresponding agent containers.

Despite the many differences between them, in ADF all transports implement a common interface that hides their inner workings from the agents and their containers. It is the task of the agent management system to choose the right protocol for a given destination, by mapping URL prefixes to the corresponding transport. This means that a message sent to a destination with the address `http://adf.sf.net/agents` will be transported over HTTP, while one sent to `jnp://adf.sf.net/ADFQueue` will be transported over JMS. This flexible scheme can be extended to any other transport protocol. However, if the current platform is `adf.sf.net`, then a much simpler local transport will be used. In fact, when a message is intended for more than one agent, it might occur that different transports are used for different receivers.

Both HTTP and JMS offer "best-effort" services, so there are no strong guarantees that once a message is sent it will actually make its way to the destination. Even when communication is persistent (the case of JMS) and the chance that a message will be lost is usually small, it is still present. Reliable communication provides quality-of-service guarantees, under one of the following three delivery semantics: at-least-once, at-most-once and exactly-once. One additional reliability provision is in-order delivery. If the communication system is not capable of delivering a message under the given constraints, it will notify the sender of the failure.

Reliable asynchronous messaging is a key building block for any service-oriented architecture and there were some initiatives in making it a standard. HTTPR was an early attempt by IBM to provide a protocol on top of HTTP for the reliable transport of messages [26]. The two current divergent directions for reliable messaging are WS-Reliable Messaging [41] and WS-Reliability [40]. Presently, ADF does not support any of them, and – to authors' knowledge – neither does any other agent framework, so this could be an interesting subject for future exploration.

3.3. Message Encodings

The agent communication language used by ADF agents is fixed to FIPA ACL in order to achieve interoperability. The structure of a FIPA ACL messages is however extensible, and it allows custom properties to be added. Inside ADF agent platforms, the messages are represented as instances of a Java class (*ACLMessage*). However, when messages need to be exchanged between different agent platforms, they have to be encoded before they are actually send to the destination and then decoded when they are received.

FIPA has defined three standard encodings for ACL messages: String, XML, and bit efficient encoding. ADF already fully supports the first two, and additionally provides a SOAP encoding by using the SOAP with Attachments API for Java (SAAJ) [36]. A message codec factory can be used to instantiate the appropriate codec for a given Multipurpose Internet Mail Extensions (MIME) type. This facility is used together with the `Content-Type` HTTP header, or with a custom defined property in JMS, in order to assure that transports and encodings are entirely decoupled. Every transport has a list of encodings it can provide, and an encoding to be used by default. Later, when the message reaches its destination the value of the `Content-Type` header is used to instantiate the right decoder for the message.

3.4. Message Contents

The content language of the messages is chosen by the agents that communicate, and it is very common to use ontologies in order to assure the semantics of the sym-

bols used are the consistent. The languages commonly used to represent the knowledge exchanged by the agents are FIPA SL [37], Knowledge Interchange Format (KIF) [32] and, more interesting, Resource Description Framework (RDF) [15].

ADF allows any of these languages to be used for the contents or messages, but currently no special support is provided for any of them. As a further work, we aim to offer support for RDF and OWL by using the Jena semantic Web framework [31]. The use of the Jena's rule-based inference engine would also open the path to building intelligent agents capable of inference, in order to provide support for the semantic Web's superior layers [7, 39].

3.5. Interaction Protocol

ACL messages can be sequenced into more complex standard interaction protocols corresponding to common interaction patterns. Protocols range from simple request-reply ones, to complex interactions between multiple peers such as negotiations or auctions. For example, the Contract Net interaction protocol [17] allows simultaneous one-to-many negotiations between an initiator and several participants (see Figure 5), and it is used in both case studies from the next session.

ADF provides support for interaction protocols by defining predefined tasks for all the different roles an agent might play in a certain protocol. An agent uses a protocol by subclassing such a predefined task and redefining some of the callback methods, and then adding the task to its task list. Whenever an important event occurs the corresponding callback method is run, and the agent can process the information contained in the received messages and take decisions on the following steps of the protocol. Because ADF allows multiple tasks to be run in turns, an agent can perform multiple interaction protocols, independently, at the same time.

4. Case Studies

This section presents two example of agent systems built using ADF. Although they are not very complex, they still illustrate well enough the many capabilities of the framework, especially in the context of semantic Web-based e-business applications.

The source code of the presented agents is available as part of the ADF distribution [11].

4.1. Reversed Auction

The first ADF-based agent system models a reversed auction system. The developed application is comprised of agents buying products and agents offering products for sale. The seller agents register their service with the local

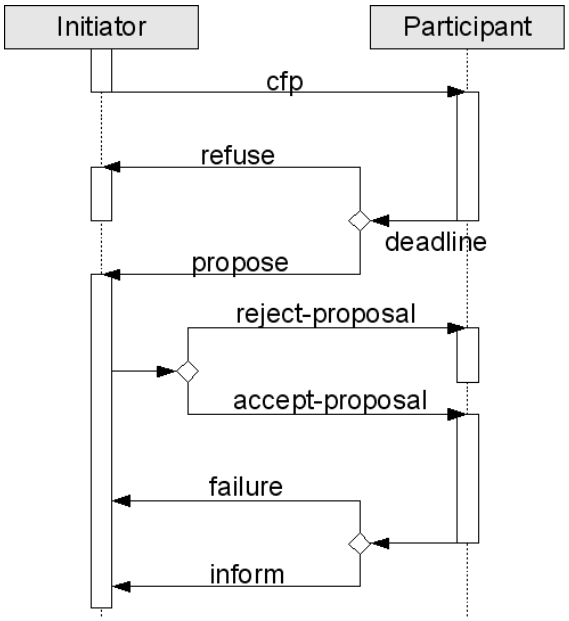

Figure 5. The Contract Net protocol [17]

service repository and then wait for requests to arrive. Every buyer agent receives the name of the product to buy as a parameter from its owner. It then locates all the seller agents, and sends them a call for proposal containing the name of the product it wants to buy.

The communication occurs by using the Contract Net interaction protocol, with the buyer playing the initiator role – Figure 5. Some of the sellers will respond with actual proposals to offer the product for a certain price (the propose act), other sellers may refuse to make a proposal, while others will not respond at all. Once all responses are received, or a deadline passes, the initiator evaluates the received proposals and selects the one offering the best price. The accepted seller receives an accept-proposal message and the other agents that made proposals receive a reject proposal message. Finally, the seller that won the auction removes the corresponding product from stock, and then informs the buyer the transaction was successful. However, when multiple negotiations happen at the same time, it is possible for several buyers to try to buy the same product, in which case only one receives the inform message, while all the others are sent a message indicating failure, and they will have to start a new negotiation in order to achieve their goal.

When a buyer agent completes its goal, that is it buys the product for its owner, it passes into the finished state. The owner can then request the results of its work, in this case the name of the agent that sold the product and the price the agent payed for it. All the data exchanged during the negotiation is encoded as plain strings, which is acceptable for such a small auction system, but inappropriate for larger

systems or when more complex content such as payment information would have to be used for the messages. We will see this problem fixed in the next example.

4.2. Meeting Scheduling

The second scenario involves agents collaborating in order to schedule a meeting on behalf of their owners. Each agent has access to relevant information about its owner's identity, schedule and the relationships between the owner and other people. The schedule is represented in RDF using an OWL ontologyof time [10], while the relationships between persons are represented using an extension – for details, consult [34] – of the Friend Of A Friend (FOAF) vocabulary [25]. The Jena [31] framework is used for processing RDF content and OWL ontologies.

When an agent is started, it receives a parameter representing an RDF description of the task to solve, in this case the details of the meeting it has to arrange. This description is composed of a set of participants, a duration, a time span, and the purpose of the meeting. The agent first finds all the time intervals that would suit its owner by inspecting her schedule, and then it contacts the agents of the given participants by issuing a call for proposal, and becoming the initiator of a Contract Net protocol. Each of the participants comes with proposals chosen from the list provided by the initiator, and in conformance with the schedule of its owner. When all the participants have responded, or a deadline passes, the initiator will chose the interval that was proposed by the most participants, and issues an accept-proposal communicative act, informing them of the choice. The remaining agents will receive a reject-proposal, and their owners will most likely not make it to the meeting.

5. Conclusion and Further Work

This paper has presented the development of a J2EE-based reference implementation of the Agent Developing Framework (ADF) and has investigated the methodology and technologies that made this possible. The results achieved so far on small agent systems – two of them presented in section 4 – demonstrate the soundness of our approach, and hopefully they will be easily generalizable on more complex systems.

Although the framework can already be used to build multi-agent systems, it is still in a fairly early stage of development, and many directions of improvement are still possible. For the near future, there is the need for a global naming service and a global service registry, build-in support for content expressed as RDF, for OWL ontologies and for inference using a rule-based inference engine, and even more transport protocols, message encodings and interaction protocols.

Easy-to-use tools – for installation and facilitating tasks like administration and debugging and, also, for visually designing agent systems – could make the ADF framework more attractive and accessible even to inexperienced developers.

On the long run, other directions of interest, such as agent mobility and security, will be the subject of our research.

Another important aspect is to study of the possibility of using ADF system within *ITW* [6], a semantic Web-based platform for multimedia resource discovery.

References

[1] Alboaie, S., Buraga, S., Alboaie, L., "An XML-based Object-Oriented Infrastructure for Developing Software Agents", *Scientific Annals of the "A. I. Cuza" University of Iaşi*, Computer Science Series, tome XII, 2002

[2] Alboaie, S., Buraga, S., Alboaie, L., "An XML-based Serialization of Information Exchanged by Software Agents", *International Informatica Journal*, 28 (1), 2004

[3] Alur, D. et al., *Core J2EE Patterns: Best Practices and Design Strategies (Second Edition)*, Prentice Hall PTR, 2003

[4] Bray, T. (ed.), *Extensible Markup Language (XML) – version 1.0 (Third Edition)*, W3C Recommendation, Boston, 2004: `http://www.w3.org/TR/REC-xml`

[5] Buraga, S., Alboaie, S., Alboaie, L., "An XML/RDF-based Proposal to Exchange Information within a Multi-Agent System", in Grigoraş, D., Nicolau, A. (eds.), *Concurrent Information Processing and Computing*, NATO Science Series, IOS Press, 2005

[6] Buraga, S., Găbureanu, P., "A Distributed Platform based on Web Services for Multimedia Resource Discovery", in *Proceedings of the Second International Symposium on Parallel and Distributed Computing – ISPDC'03*, IEEE Computer Society Press, 2003

[7] Daconta, M., Obrst, L., Smith, K., *The Semantic Web*, Wiley, 2003

[8] Dean, M., Schereiber, G. (eds.), *OWL Web Ontology Language Reference*, W3C Recommendation, Boston, 2004: `http://www.w3.org/TR/owl-ref/`

[9] He, H., "What is Service-Oriented Architecture?", *XML.com*, 2003: `http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html`

[10] Hobbs, J., Pan, F., "An Ontology of Time for the Semantic Web", *ACM Transactions on Asian Language Processing (TALIP): Special issue on Temporal Information Processing*, 3 (1), 2004

[11] Hriţcu, C., *ADF: Agent Developing Framework*, 2005: `http://adf.sourceforge.net/`

[12] Krafzig, D. et al., *Enterprise SOA: Service-Oriented Architecture Best Practices*, Prentice Hall PTR, 2004

[13] Nichifor, O., Buraga, S., "ADF – Abstract Framework for Developing Mobile Agents", in Petcu, D. et al. (eds.), *Proceedings of the 6th Internation Symposion on Symbolic and Numeric Algorithms for Scientific Computing – SYNASC'04*, Mirton, 2004

[14] Nwana, H., Ndumu, D., "A Perspective on Software Agents Research", *The Knowledge Engineering Review*, 16 (3), 1999

[15] Manola, F., Miller, E. (eds.), *RDF (Resource Description Framework) Primer*, W3C Recommendation, Boston, 2004: http://www.w3.org/TR/rdf-primer/

[16] Monson-Haefel, R., Chappell, D., *Java Message Service*, O'Reilly, 2001

[17] Smith, R., "The Contract Net Protocol: High-Level Communications and Control in a Distributed Problem Solver", *IEEE Transactions on Computers*, 29 (12), 1980

[18] Winograd, T., Flores, F., *Understanding Computers and Cognition – A New Foundation for Design*, Addison Wesley, 1987

[19] * * *, *Aglets*: http://aglets.sourceforge.net/

[20] * * *, *Ajanta: Mobile Agents Research Project*: http://www.cs.umn.edu/Ajanta/

[21] * * *, *Cougaar – Cognitive Agent Architecture*: http://cougaar.org/

[22] * * *, *D'Agents*: http://agent.cs.dartmouth.edu/

[23] * * *, *DIET Agents*: http://diet-agents.sourceforge.net/

[24] * * *, *FIPA-OS (Foundation for Intelligent Physical Agents – Open Source) Agent Toolkit*: http://fipa-os.sourceforge.net/

[25] * * *, *FOAF (Friend Of A Friend) Vocabulary Specification*: http://xmlns.com/foaf/0.1/

[26] * * *, *HTTP Reliable (HTTPR) Specification*: http://www.ibm.com/developerworks/library/ws-httprspec

[27] * * *, *J2EE (Java 2 Platform, Enterprise Edition)*: http://java.sun.com/j2ee/

[28] * * *, *JADE – Java Agent DEvelopment Framework*: http://jade.tilab.com/

[29] * * *, *JMS (Java Message Service)*: http://java.sun.com/products/jms/

[30] * * *, *JBoss Application Server*: http://www.jboss.org/

[31] * * *, *Jena – A Semantic Web Framework for Java*: http://jena.sourceforge.net/

[32] * * *, *KIF (Knowledge Interchange Format)*: http://logic.stanford.edu/kif/kif.html

[33] * * *, *KQML (Knowledge Query and Manipulation Language)*: http://www.cs.umbc.edu/kqml/

[34] * * *, *Relationship: A Vocabulary for Describing Relationships between People*: http://vocab.org/relationship/

[35] * * *, *Simple Object Access Protocol (SOAP) 1.2*: http://www.w3.org/TR/soap12/

[36] * * *, *SOAP with Attachements API for Java (SAAJ)*: http://java.sun.com/xml/saaj/

[37] * * *, *The Foundation for Intelligent Physical Agents (FIPA)*: http://www.fipa.org/

[38] * * *, *Voyager Java Development Platform*: http://www.recursionsw.com/voyager.htm

[39] * * *, *World-Wide Web Consortium (W3C)*: http://www.w3.org/

[40] * * *, *WS-Reliability – OASIS Web Services Reliable Messaging TC*: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm

[41] * * *, *WS-ReliableMessaging – Web Services Reliable Messaging*: http://www-128.ibm.com/developerworks/library/specification/ws-rm/

[42] * * *, *Zeus Agent Toolkit*: http://sourceforge.net/projects/zeusagent/