DOI: 10.2298/CSIS120102025M

Supporting heterogeneous agent mobility with ALAS

Dejan Mitrović[1], Mirjana Ivanović[1], Zoran Budimac[1], and Milan Vidaković[2]

[1] Faculty of Sciences, Department of Mathematics and Informatics
Trg Dositeja Obradovica 4, 21000 Novi Sad, Serbia
{dejan, mira, zjb}@dmi.uns.ac.rs
[2] Faculty of Technical Sciences
Trg Dositeja Obradovica 6, 21000 Novi Sad, Serbia
minja@uns.ac.rs

Abstract. Networks of multi-agent systems are considered to be heterogeneous if they include systems with different sets of *API*s, running on different virtual machines. Developing an agent that can operate in this kind of a setting is a difficult task, because the process requires regeneration of the agent's executable code, as well as modifications in the way it communicates with the environment. With the main goal of providing an effective solution to the heterogeneous agent mobility problem, a novel agent-oriented programming language, named *ALAS*, is proposed. The new language also provides a set of programming constructs that effectively hide the complexity of the overall agent development process. The design of the *ALAS* platform and an experiment presented in this paper will show that an agent written in *ALAS* is able to work in truly heterogeneous networks of multi-agent systems.

Keywords: agent-oriented programming languages, mobile agents, heterogeneous agent mobility, multi-agent systems

1. Introduction

According to the *weak notion of agency* [39], *software agents* can be defined as executable software entities characterized by autonomous behavior, social interaction with other agents, reactivity to environmental changes, and the ability to take the initiative and express goal-directed behavior. The *strong notion of agency* [40] extends this definition by including human-like behavior and mental categories, such as beliefs, desires, and intentions (the so-called *BDI* agents).

Agents usually don't exist on their own, but are rather situated inside an environment. This runtime agent environment is often referred to as a *multi-agent system* (*MAS*). Main tasks of a *MAS* are to control the agent life-cycle, provide the messaging infrastructure, and offer a service subsystem that effectively supports agents, giving them the possibility of accessing resources, executing complex algorithms, etc.

An agent, however, does not have to be confined to a single *MAS* instance. A *mobile* agent is able to physically leave its current *MAS* and continue pursuing its goals in another machine in a network.

EXtebsible Java EE-based Agent Framework (*XJAF*) [34, 35] is a multi-agent system developed by the authors of this paper. The system is designed as a modular architecture, comprised of a set of *managers*. Each manager is a relatively independent module in charge of handling a distinct part of the overall agent-management process. There are several benefits of the modular design. For example, the system's functionality can easily be extended by the addition of new managers. In addition, each manager is accessible only though its interface, which means that even the behavior of standard managers can be easily changed.

Over the years, *XJAF* has been successfully used in several software systems, such as the virtual central catalogue and a metadata harvesting system for library records [34]. Recently [16], it has also been proposed as an underlying platform for agent-based harvesting of learning resources needed by e-learning and tutoring systems. But, despite its successful practical usage, *XJAF* had a disadvantage of being "locked" into a particular development platform. Because it was implemented in Java, only Java-based external clients were able to use the system and interact with its agents. In order to increase the interoperability of the system and enable its wide-spread use, *XJAF* has been redesigned as a service-oriented architecture (*SOA*). The new system, named *SOA-based MAS* (*SOM*) [22], retains the manager-based design, but with managers re-defined in terms of web services. In this way, even regular web browsers can be used as clients of *SOM*, since the interaction relies on the standardized communication protocol (i.e. *SOAP* [36]).

The *SOA*-based design of *SOM*, however, poses another problem. The system is (only) an abstract specification of web services, their functionalities and interactions, and any modern implementation platform can be used. But, developing an agent that can run on any of these implementations becomes almost an impossible task. In the literature (e.g. [24]), this issue has been recognized as an *agent-regeneration* problem: if a mobile agent migrates across a network consisting of *MAS*s that offer the same *API*, but are based on different virtual machines, its executable code needs to be regenerated for each *MAS* it visits.

Unfortunately, the lack of *MAS* interoperability is not specific to different implementations of *SOM*. Currently, there exists a large number of *MAS*s offered by different vendors. And although significant efforts have been put into the standardization of the *MAS* development process (e.g. the *Foundation for Intelligent Physical Agents*, *FIPA* [11]), agents are often incapable of operating in these truly heterogeneous environments. This problem arises as a consequence of standards incompliance, usage of different implementation technologies, different sets of *API*s offered to agents, etc. The lack of interoperability is a severely limiting factor in the agent development, and in the wide-spread use of the agent technology.

In order to solve the *MAS* interoperability problem, a new agent-oriented programming language named *Agent LAnguage for SOM* (*ALAS*) is proposed. Besides providing developers with programming constructs that hide the overall complexity of the agent-development process, one of the main goals of *ALAS* is

to serve as a tool for writing agents that can execute their tasks regardless of the underlying *MAS*. Originally, in [22], *ALAS* has targeted the agent-regeneration problem. The idea has since been broadened to support the execution of agents in heterogeneous environments. The focus of the research presented in this paper will thus be to demonstrate how *ALAS* can be used to develop mobile agents that can migrate across a network consisting of Java EE-based *SOM*, Python-based *SOM*, and *JADE* [2, 17] instances.

As noted, *ALAS* belongs to the category of agent-oriented programming languages (*AOPL*s) which represent crucial tools of the *agent-oriented programming* (*AOP*). *AOP* is a software development paradigm aimed at efficient development of software agents and multi-agent systems. Its main goals are to identify, analyze, and offer solutions for the most important theoretical and practical issues associated with the design and construction of software agents.

The rest of the paper is organized as follows. Section 2 provides an overview of existing research efforts related to the work presented in this paper. Section 3 describes the architectures of *XJAF* and *SOM*, multi-agent systems that form the basis for this research. Main features of *ALAS*, its syntax and programming constructs are given in Section 4. A practical example of an *ALAS*-based mobile agent operating in a heterogeneous environment is given in Section 5. Finally, the overall conclusion and future research directions are outlined in Section 6.

2. Related work

Related research efforts presented in this section are divided into three parts. The first part includes a general overview of existing multi-agent systems. The second part outlines the state-of-the-art of *AOPL*s. The final part deals with the work dedicated to interoperability multi-agent systems.

2.1. Multi-agent systems

Java Agent DEvelopment Framework (*JADE*) [2, 17] is a Java-based, *FIPA*-compliant *MAS*. At runtime, the framework consists of one of more *agent containers*, runtime environments with full support for agent execution. Individual containers can be distributed across a network, in which case they are linked to a designated *main* container. Each *JADE* agent has it own thread of control and exposes its functionalities in terms of *behaviors*. That is, for each functionality offered by an agent, developers need to define a separate class which extends the *Behaviour* class, or one of its more specialized subclasses. A background *scheduler* is then used to schedule execution of each behavior.

The main advantage of *XJAF* and *SOM* over *JADE* is in the use of Java EE, the *de facto* standard development platform for building large-scale, scalable, secure, and reliable software. Java EE includes a large set of standardized libraries and technical solutions which simplify the process of *MAS* development. More importantly, the use of modern enterprise application servers incorporates effective runtime agent load-balancing techniques into *XJAF* and *SOM*. Finally,

the *SOA*-based design of *SOM* results in the system with greater interoperability.

The *SOA* design philosophy has been employed in the development of *FUSION@* [31, 32], a modular, *FIPA*-compliant *MAS*. Functionalities of the system are exposed as *services* that can be accessed locally, or remotely through web interfaces. The set of services is not fixed, which means that the system can easily be extended with new functionalities. Any programming language can be used for implementing new services, as long as it supports *SOAP*. *FUSION@* also employs several types of specialized, system-level *BDI* agents. Their main task is to maintain high quality of service (*QoS*), by performing runtime load distribution, monitoring all incoming and outgoing messages, etc.

Many properties of *FUSION@*, such as the extensible modular architecture, and the use of low-level services, have been used in *XJAF*, although several years earlier. Additionally, *XJAF* delegates some functionalities of *FUSION@*'s system-level agents, such as runtime load-balancing, to an enterprise application server, which simplifies the overall development process. Finally, it is not clear how and if agent mobility is supported in *FUSION@*, or whether there is a mechanism for organizing distributed instances of the environment. These techniques have been built into *XJAF* and *SOM* from the start.

NOMADS [4, 30] is one of the few *MAS*s that support strong agent mobility (e.g. all other systems mentioned in this paper support weak mobility only). In order to achieve this feature, *NOMADS* runs on top of a customized, Java-compatible virtual machine named *Aroma*. *Aroma* can transparently capture the execution state of a single or all running threads, at the fine granularity level, and in a cross-platform manner. Additionally, it can limit the agent's access to resources and enforce similar security-related restrictions.

The use of a custom virtual machine in *NOMADS*, however, has several major disadvantages. These include interoperability issues, as well as the large amount of work that needs to be conducted in order to maintain and update *Aroma* for different operating systems and in accordance to new Java virtual machine specifications. Nonetheless, the system does offer an interesting technical insight into requirements of strong and safe agent mobility.

Java is by far the most widely used platform for *MAS* development. However, there exist notable examples of systems implemented using different technologies. One such example is *Smart Python multi-Agent Development Environment* (*SPADE*) [1, 29] implemented in Python. *SPADE* is characterized by the usage of *XMPP/Jabber* [41] instant messaging protocol for agent communication. Benefits of *XMPP/Jabber* include using [1] "an existing communication channel, the concepts of users (agents) and servers (platforms) and an extensible communication protocol based on *XML*". The *presence* feature included in the protocol also enables the real-time detection of the agent's state.

SPADE platform is accompanied by a Python-based *agent library* of classes, functions, and data structures that simplify the agent development process. The *SPADE* agent development process is heavily inspired by that used in *JADE*: functionalities of an agent are expressed in terms of behaviors, there is a sup-

port for automatic, pattern-based matching of incoming messages, and so on. The successful usage of Python as the implementation platform for *SPADE* has inspired the development of Python-based *SOM* (discussed in more details in Section 5).

2.2. Agent-oriented programming languages

AGENT0 [28] was the first agent-oriented programming language and a direct implementation of the *AOP* paradigm. In *AGENT0*, agents are defined in terms of capabilities, beliefs, and commitment rules, which consist of message and mental preconditions and resulting actions. Agents communicate by exchanging *request*, *unrequest*, and *inform* messages. New features were added to the language over time, with the two most notable direct extensions being *PLACA* [33], which introduced support for agent planning, and *Agent-K* [7], which replaced the custom communication messages with the standardized *KQML*, improving the overall interoperability. These three languages, however, mostly served as prototypes and were not designed for practical use. Their importance lays in the influence they had on the development of many later *AOPL*s.

A large family of *AOPL*s includes languages that use first-order logical formulae for describing agent's mental state and behavior. Thus, they are specifically suited for *BDI* agent architectures. Influential representatives of this family are *3APL* [6], and *AgentSpeak(L)* [26]. *3APL* supports descriptions of goals and basic and composite plans, as well as "embedding" actions inside pre-condition and post-condition rules. These rules, respectively, describe agent's belief before and after the action is executed. Another important concept of *3APL* are *goal*, *interaction*, and *plan rules*, which are used generate new and update or drop existing goals and plans. *3APL* has inspired the development of many other programming languages, most notably *2APL* [5] and *GOAL* [15]. *2APL* increases the expressiveness of *3APL* and aims at developing a more practical language. It makes a clear distinction between declarative concepts for describing agent's beliefs and goals, and imperative concepts for developing plans (unlike *3APL*, which mixes both declarative and imperative concepts in defining goals). *GOAL*, among other things, introduces the *blind commitment strategy*, a built-in goal update mechanism that automatically drops goals that have been fully achieved, and *perception rules* which enable agents to respond to external environmental changes.

AgentSpeak(L) is gaining more and more popularity due to the development of *Jason* [3, 18], an interpreter for an extended version of the language. Along with programming constructs for describing "common" features of *BDI* agents, such as beliefs, goals, rules, and plans, the extended version of *AgentSpeak(L)* supports *belief annotations*. Annotations are programming constructs used to attach additional details to agent's beliefs. They do not increase the expressiveness of the language, but improve its readability and allow for (semi-)automatic management of the agent's belief base.

Although powerful and expressive, these languages suffer from several major drawbacks. First of all, their descriptive nature and logical foundation might

make them computationally expensive and complex. Secondly, the languages were specifically designed for developing *BDI*-style agents and so are inadequate for implementing other types of architectures (e.g. purely reactive agents). But, most importantly, the logical foundation and highly-abstracted programming constructs used by this family of languages might prove to be their greatest weakness. To avoid the faith of logic and functional programming languages that were never widely adopted by the software development industry, *AOPL*s should first and foremost be as simple to use as possible, i.e. without too many high-level abstractions and without requiring a degree in computational logic to understand their concepts. *ALAS* was designed to appear as a member of the OO family of languages – the most widely used programming paradigm of today – but with clear distinctions between objects and agents. This simplified approach might turn out to be its main advantage, allowing for a broader acceptance of the agent technology.

JACK Intelligent Agents [38] is a robust, light-weight framework for rapid development of multi-agent systems. It extends the Java programming language by introducing new keywords and language constructs. The accompanying compiler produces pure Java code, which allows for each *JACK* agent to be used simply as another Java object. Although powerful, this system suffers from the same drawbacks as the original *XJAF*: it is locked into a particular development platform (i.e. Java). The *ALAS* platform is designed to allow transformation of the original agent source code into a pure source code written in an arbitrary language (such as Java and Python). With the *SOA*-based infrastructure supporting their execution, these agents can work in truly heterogenous environments and cooperate with any external *SOA*-enabled entity. In addition, *JACK* is a commercial product, while *SOM* can be freely downloaded and used.

2.3. *MAS* interoperability

Two *MAS*s are said to be *interoperable* "if a mobile agent of one system can migrate to the second system, the agent can interact and communicate with other agents (local or even remote agents), the agent can leave this system, and it can resume its execution on the next interoperable system" [25]. Therefore, interoperability of *MAS*s can seriously affect the agent's performance, by limiting its ability to move across the network or to interact with other agents.

Based on the actual types of *MAS*s that appear in a network, several types of agent mobility can be distinguished [24]:

- *Homogeneous*: all *MAS*s in the network offer the same *API* and are based on the same virtual machine (*VM*). This is the easiest type of mobility to implement, since no modifications of the agent's code are needed.
- *Cross-platform*: *MAS*s in the network offer different sets of *API*s, but are based on the same *VM*. In this scenario, the agent's executable code remains the same, but the *API* calls it makes need to be adapted.
- *Agent-regeneration*: the agent moves across *MAS* instances that offer the same *API*, but are based on different *VM*s. Therefore, the agent's executable code needs to be regenerated for each instance it visits, although

its *API* calls remain the same. This is the problem that affects different *SOM* implementations.

– *Heterogeneous*: *MAS*s in the network offer different sets of *API*s and are based on different *VM*s. This is the most difficult type of mobility to achieve, as it includes both regeneration of the executable code, and the modifications in *API* calls.

Cross-platform agent mobility is usually achieved via one or more software *layers*, where each layer is responsible for transforming *API* calls from one form to another. For example, *Grid Mobile-Agent System* (*GMAS*) [13] includes a *Foreign2GMAS* layer, which transforms agent's native *API* calls into an intermediary *GMAS API*, and a *GMAS2Native*, which transforms *API* calls made to *GMAS API* into calls to the native platform. The first layer is required for a *MAS* that needs to be able to send its agents to other architectures. Similarly, the second layer is required for a *MAS* that needs to be able to accept agents from other architectures.

Java-based Interoperable Mobile Agent Framework (*JIMAF*) [12] operates on the principle of splitting the agent implementation into a platform-independent (called the *head*) and a platform-dependent part (called the *body*). The body is executed within *Platform-dependent Mobile Agent Layer* which is implemented for each supported platform. When compared to *GMAS*, *JIMAF* is reported [12] to introduce significantly less overhead to the agent migration process.

In *ALAS*, the task of adapting *API* calls is handled transparently by the compiler's *MAS selector* component (see Section 4 for more details). The component transforms, on-the-fly, the calls made to the *ALAS* standard library of functions into native *API* calls. The main advantage of this approach is that, once the agent's executable code is regenerated, all native *API* calls are made directly. That is, there is no additional overhead introduced by layering *API* calls.

Much more work, however, is needed for heterogeneous agent mobility, since both the executable code and *API* calls need to be regenerated. *Generative migration* [24] is one proposed solution for this problem. Rather than on software layering, it relies on a pool of agent *building blocks*, platform-independent descriptions of reusable software components. Each building block is characterized solely by a description of its interface, without any details regarding the implementation. An agent is defined (or, rather, designed) by assembling and interconnecting these building blocks into an *agent blueprint*. During the migration process, the agent's blueprint is transferred, along with its runtime state. Using the blueprint, an *agent factory* tool can rebuild the agent's executable code for a specific *MAS*.

ALAS solves the same problem as generative migration does. However, *ALAS* is a programming language, syntactically (and, to a certain degree, conceptually) similar to many popular OO programming languages. Generative migration, on the other hand, might be formalized as a *Model-Driven Architecture* [27]. For a common software developer, this means that *ALAS* has a flatter learning curve than generative migration. In addition, *ALAS* has a wider goal of

simplifying the whole agent-development process, and it's not focused just on enabling heterogeneous agent migration.

3. *XJAF* and *SOM*: design and functionalities

XJAF was originally designed as a modular architecture. Each module, called a *manager*, is responsible for handling a distinct part of the overall agent-management process. The architecture defines a set of standard managers, described in the following paragraphs. This set is not fixed, and new managers (that is, new functionalities) can be added as needed. Additionally, managers are defined and used solely by their interfaces, so even the standard behavior can be changed.

AgentManager maintains the directory of agents and controls the agent life-cycle. Its functionality matches the one defined for the *FIPA*'s *Agent Directory Service* [9]. The directory of agents consists of two lists: *local* and *remote*. The first list keeps a record of agents located in the manager's host *XJAF* instance. The second, remote list is used to support agent mobility. Once an agent leaves its current host and migrates to another machine in the network, it is removed from the local, and placed in the remote list (along with the address of its new host *XJAF*). So when a message needs to be delivered to the agent, *Agent-Manager* will:

- Check the local list and, if the agent is available there, deliver the message directly.
- Otherwise, check the remote list and, if the agent is available there, forward the message to *AgentManager* of the agent's new host.

These steps are repeated in each *XJAF* instance in the agent's migration path, until the agent is finally located (i.e. until it appears in the *AgentManager*'s local list). This simple, yet effective technique of agent location tracking is known as the *forwarding pointers* technique [23].

ConnectionManager is the manager in charge of maintaining a network of distributed *XJAF* instances and, in combination with the previously described agent tracking technique, serves as the support for agent migration. In the earliest implementation [34], each *XJAF* instance in a network had a single other *XJAF* instance to register with, forming a tree-like structure. This organization, however, was characterized by a single point of failure – if one instance breaks, the whole tree is divided into two sets of mutually unreachable instances. Recently [20, 21], this organizational structure has been replaced with a fully-connected graph. A new type of a mobile agent, named *ConnectionAgent* was added to the system, with the job of building and maintaining the graph in an efficient fashion. This new approach of organizing *XJAF* instances is shown [21] to be fault-tolerant to unexpected failures, enabling each remaining *XJAF* to have the correct overview of the network state, regardless of the number of failures.

MessageManager provides the messaging infrastructure. It supports inter-agent communication via the exchange of *KQML* messages [8]. A *KQML* message sent from one agent to another is embedded into a *JMS* message [14]. *MessageManager* then broadcasts the message to all *XJAF* instances that have previously subscribed to this service, but only the instance containing the target agent will process the message and extract *KQML* content from it. In the ongoing work of increasing the interoperability of *XJAF*, the *KQML*-based messaging system will be replaced by the *de facto* standard *FIPA ACL* [10].

An important aspect of each *MAS* is security. In terms of the agent technology, security features are used to protect both agents and the *MAS* itself from malicious attacks, to keep the confidentiality of exchanged messages, etc. In *XJAF*, these features are offered by *SecurityManager*. And since often there is a significant computational overhead associated with security (e.g. encryption/decryption of messages), the security features are not applied automatically, but can rather be included on-demand, through an *API* exposed by the manager.

XJAF includes a *service* sub-system, where the service is a reusable software component managed by *ServiceManager*. The basic idea is to expose common tasks, such as file management, in form of services that can be directly accessed and used by agents. This approach simplifies the agent development process and supports the development of lighter agents (in terms of size). The list of services is not fixed and can be expanded as needed.

XJAF agents expose their capabilities in form of *tasks*. A task includes a detailed description of a single functionality offered by the agent. It incorporates types and names of input parameters as well as of the returned value. The list of tasks offered by the agents is maintained by *TaskManager*. External clients of the system can ask for a task execution. In response, *TaskManager* will find the most suitable agent for the given task, and then send it an appropriate message. For interoperability reasons, the format of task descriptions is based on the standardized and widely-used *W3C XML Schema language* [37].

The main advantage of *XJAF* over other existing *MAS* implementation is in its use of the Java EE technology. Java EE has been endorsed by large business enterprises as the main tool for building large-scale, scalable, secure, and reliable software. As such, it represents an excellent platform for *MAS* development. Immediate direct benefits of this approach are shorter development time, standards compliance, and harnessing of advanced programming features. For example, each *XJAF* agent is a regular Java object (i.e. a *POJO*), but wrapped inside an *Enterprise JavaBean* (*EJB*) component. At runtime, the component is passed to an enterprise application server in order to employ runtime load-balancing and object pooling features.

The original *XJAF* had one serious disadvantage – it was ”locked” into a particular development platform. A consequence of this problems is the lack of interoperability, in the sense that only Java-based external clients could access the system and interact with its agents. In order to overcome this issue, a new system, named *SOA-based MAS* (*SOM*) has been developed. *SOM* follows the

manager-based design approach of *XJAF*, but with managers implemented as web services. The most important improvement introduced by the ”switch” to the *SOA*-based design is increased interoperability: external clients and third-party tools can interact with *SOM* and its agents through web service interfaces, i.e. in a familiar fashion, and using the standardized communication protocol.

Unfortunately, the *SOA*-based design of *SOM* introduced a new, major issue. Because the system is a specification of web services, it can be implemented using many modern programming languages. However, developing an agent that can run on any of these implementations becomes almost an impossible task. In order to solve this problem, a new agent-oriented programming language, named *Agent LAnguage for SOM* (*ALAS*) has been developed. Its main features and functionalities are described in the following section.

4. Main features of *ALAS*

Originally, in [22], *ALAS* and its accompanying set of tools (in further text, the *ALAS platform*) were primarily aimed at the development of agents for different *SOM* implementations. One of the main characteristics of the *ALAS* platform is *hot compilation*: when an agent arrives to an instance of *SOM* implemented in a certain programming language *X*, its *ALAS* source code is transformed on-the-fly into the source code written in *X*. The generated source code is then forwarded to the native compiler, if any, to produce the executable code for the target platform.

Since the original proposal, the functionality of *ALAS* platform has been broadened to include support for other *MAS* implementations, such as *JADE*. According to the classification of agent mobility presented in [24] and described earlier, this means that the *ALAS* platform has been upgraded from delivering agent-regeneration to supporting true heterogeneous agent mobility. The main goal of *ALAS* is, therefore, to create an agent-oriented programming that hides the complexity of the overall agent-development process from developers, and, at the same time, operates regardless of the underlying *MAS*.

The *ALAS* platform has been designed ground-up with the idea of heterogeneous agent mobility in mind. The entire process of transforming the *ALAS* source code into the executable code for the target platform is shown in Fig. 1. In the first step, the agent source code written in *ALAS* is parsed to produce an abstract syntax tree. The tree is then fed into the *VM selector* which associates it with the proper *ALAS standard library*. The standard library includes utility functions for common operations, such as string processing, file management, and network connections. To support the idea of heterogeneous agent mobility, the library was re-implemented for each of the supported target languages (currently, Java and Python). The output of this step is fed into the *MAS selector* which replaces *MAS*-specific *ALAS* instructions with native *API* calls. *MAS* selector produces a fully-functional source code of the agent for the target *MAS* which is, finally, sent to the native language compiler (if any) to produce the executable code.

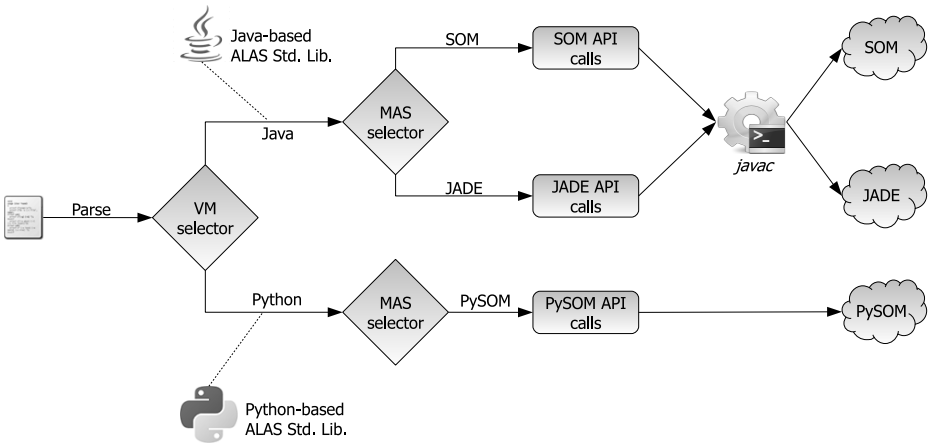

Fig. 1. The process of compiling *ALAS*-based source code of an agent into the executable code for the target platform

Protection from malicious attacks is an important issue, especially in systems that employ agent mobility. To protect the agent code during the migration process, *code certificates* can be used [35]. A certificate holds the *hash* of the *ALAS* source code, as well as the *digital signature* of the agent's internal state. The security check is performed before the parsing step, and if it fails, the agent is discarded for unauthorized modifications.

Common, non agent-specific programming constructs, such as *if-then-else* and *switch* control statements, *while*, *do-while*, and *for* loops, are also supported. Their syntax is, like in many modern programming languages, based on the syntax of the C programming language.

In the current stage, *ALAS* can be used for developing purely reactive agents. These assume agents that execute actions in response to some external events, such as messages received from other agents. The support for *BDI*-style architecture is planned for a latter stage.

ALAS compiler handles a single *compilation unit* at a time, which includes definition of one agent, and an optional package declaration:

```
CompilationUnit = [ Package ] AgentDefinition <EOF> ;
Package         = "package" Name ";" ;
Name            = Identifier { "." Identifier } ;
```

ALAS packages serve the same purpose as packages (or namespaces) in traditional procedural and *OO* programming languages. They provide the means for distinguishing between agents that have the same name, for logical grouping of related agents, etc.

4.1. Services

The main language construct for exposing agent's behavior is a *service*. In *ALAS*, a service is a functionality that the agent offers to others, and can be seen as a counterpart of a public method in *OOP*. External entities can ask for a service execution by sending an appropriate message to the agent. A *function*, on the other hand, is local and its primary use is to break large service implementations into smaller logical units. It cannot be accessed by external entities and is, therefore, a counterpart of a private method in *OOP*.

Agent definition in *ALAS* consists of the agent name and the agent body, which, in turn, is defined as a set of states, services, and functions:

```
AgentDefinition  = "agent" Identifier
                   "{" { AgentBodyDef } "}" ;
AgentBodyDef     = ( LookAhead(3) AgentState |
                     "service" Function |
                     "services" "{" { Function } "}" |
                     LookAhead(3) Function ) ;
Function         = ResultType Identifier ParamList Block ;
ParamList        = "(" Param { "," Param } ")" ;
Param            = Type Name ;
```

Agent service definition begins with the keyword *service*, followed by the return type, unique name of the service, formal parameter list, and a body. As a shortcut (i.e. to avoid typing the *service* keyword for each new service), several services can grouped under a single *services* block. When *SOM* is used as the target platform, a separate *XML*-based task description is produced for each defined service.

An important thing to note about agent services is that method overloading from *OOP* languages cannot be applied. That is, an agent cannot expose two or more services under the same name, even if formal parameters differ. Although the syntax of a service definition resembles the syntax of a method definition, services are actually message handlers. In standardized agent communication, the order of values passed as the message content does not (or, should not) matter. For example, the following code represents a *KQML* message that *AgentA* sends to *AgentB* asking for the execution of its service *PrintSum*. The message includes values 5 and 6 for the service's two integer parameters, *a* and *b*, respectively:

```
(achieve
  :sender AgentA
  :receiver AgentB
  :language XML
  :content "<service>
      <name>PrintSum</name>
      <args>
        <arg name="a" type="int"><![CDATA[5]]></arg>
        <arg name="b" type="int"><![CDATA[6]]></arg>
```

```
    </args>
  </service>" )
```

So even if the order of *arg* tags changes, the agent is still asked to execute the same service.

Unlike services, functions can be overloaded.

4.2. Agent runtime state

ALAS is a strongly and statically typed language. In addition to *void*, the language supports *byte*, *short*, *int*, *long*, *float*, *double*, *boolean*, and *char* primitive data types. These types match the appropriate primitive data types of the Java programming language. Currently, the only supported complex data type is *String*, while the support other complex data types (e.g. for artifacts modeling) will be included later.

The *runtime state* of an *ALAS* agent is represented by a set of *persistent* and *temporary* properties. During the migration process, values of persistent properties are automatically saved before the agent leaves its host *MAS*, and later automatically restored once it reaches the target *MAS*. Temporary properties, on the other hand, should be used only to store values that are not supposed to be transferred along with the agent.

The syntax for defining runtime state of an *ALAS* agent is as follows:

```
AgentState = ( "state" "{" { LocalVar ";" } "}" |
               LocalVar ";" ) ;
LocalVar   = Type Var { "," Var } ;
Var        = Identifier [ "=" Expression ] ;
```

Any property defined within the *state* block will be considered persistent. Properties defined outside of this block will be considered temporary.

As show, expressions can be used to set initial values of properties along with declarations. If more complex initialization steps are required, a function with the following signature can be defined: *void initialize()*. This special-purpose, parameterless function is automatically invoked during the agent's startup, and before any other function or service. Therefore, it corresponds to a constructor in traditional OO programming languages (or the $\_\_init\_\_$ method in Python).

4.3. Support for agent mobility

In order to support agent mobility, *ALAS* includes two programming instructions: *copy* and *move*. The first instruction makes a clone of the original agent in the target *MAS*, which means that the agent continues to operate in the source *MAS*. This instruction can appear at any point in a service or function implementation. The *move* command, on the other hand, physically moves the agent to the target *MAS*, which means that the original agent is disposed. This is why the *move* instruction can be only the very last instruction in a service implementation. The syntax of *copy* and *move* instructions is as follows:

```
MoveStatement = ( "copy" | "move" )
                  "(" Expression [ "," Expression
                  { "," MoveArg } ] ")" ";" ;
MoveArg       = StringLiteral "=" Expression ;
```

Each instruction accepts at least one parameter, an expression that represents the target *MAS*. The value can be the concrete name of the *MAS*, or the physical network address of the machine that hosts it, in the format $host : port$. The second expression, if specified, is the name of the agent's service that should be automatically invoked once the target *MAS* is reached. Finally, a set of values that should be passed to the service can be specified, in the form of $ParameterName = Value$.

Once the migration process is initialized, the agent is serialized into an *XML* stream. The transferred data includes:

- The original (*ALAS*) source code of the agent.
- Identifier of the agent's originating *MAS* (included for convenience reasons).
- Agent's persistent properties.
- Name of the service to be automatically invoked.
- The set of arguments for the service.

The presented set of *ALAS* features already offers the possibility of writing powerful agents. With the backing of the *ALAS* platform, these agents are able to operate in networks consisting of Java EE-based *SOM*, Python-based *SOM*, and *JADE* instances, as shown in the next section.

5. A mobile *TimeSync ALAS* agent

The example of using the *ALAS* platform presented in this section is intended to serve as a proof of concept. The experiment will demonstrate how an *ALAS*-based agent operates and migrates in a truly heterogeneous network environment.

For the purpose of this experiment, a *TimeSync* agent was developed. The agent, upon receiving an appropriate message, visits all *MAS*s in the network and synchronizes their timers. The message sent to the agent includes a comma-separated list of network addresses to be visited, and the value of time (of type *double*) to be set in each *MAS*. Once it synchronizes the timers of all systems, the agent returns to its originating *MAS*.

The network includes 3 different multi-agent systems:

- A Java EE-based implementation of *SOM*
- A Python-based implementation of *SOM*, named *PySOM*
- *JADE* version 4.1

Each system features a module that accepts a serialized form of the *ALAS* agent, de-serializes it, invokes the *ALAS* compiler, restores the agent's runtime state, and then sends it the message for service execution.

Python was chosen as the second implementation platform for *SOM* in this experiment because the language itself is very different from Java in many aspects, such as the syntax, dynamic vs. static typing, multi-paradigm vs. OO paradigm, etc. In this way, the experiment will show that the *ALAS* platform is not tied to the Java platform.

JADE was included in order to demonstrate the ability of *ALAS* agents to operate in heterogeneous environments. The system was extended with an *ALAS* platform plug-in that performs the aforementioned steps from accepting the agent, to requesting service execution.

The full source code of the *TimeSync* agent written in *ALAS* is shown in Listing 5.1.

Listing 5.1. Full source code of the mobile *TimeSync ALAS* agent, capable of operating in a network consisting of *SOM*, *PySOM*, and *JADE* instances

```
1  package example.agents;
2
3  agent TimeSync {
4    state { String startingHome; }
5    String next, remaining;
6
7    service void SyncTimers(String hosts, double time) {
8      if (startingHome == null)
9        startingHome = host(); // remember the starting point
10     else if (startingHome.equals(host())) { // am I back home?
11       log("I'm back!");
12       startingHome = null;
13       return; }
14     // apply the time
15     log("Setting the system time to ", time);
16     applySystemTime(time);
17     // go to the next host
18     if (hosts.length() == 0) // no more hosts, go back home
19       next = startingHome;
20     else
21       parseHosts(hosts);
22     move(next, "SyncTimers", "hosts"=remaining, "time"=time); }
23
24   void parseHosts(String hosts) {
25     int n = hosts.indexof(",");
26     if (n == -1) {
27       next = hosts;
28       remaining = "";
29     } else {
30       next = hosts.substring(0, n);
31       remaining = hosts.substring(n + 1); } } }
```

Line 1 sets the agent's package to *example.agents*, and the agent definition starts at line 3. Lines 4 and 5 define the agent's runtime state as a set of three

String properties: *startingHome*, *next*, and *remaining*. The first property is persistent. It holds the identifier of the agent's starting *MAS*, i.e. the *MAS* that was hosting the agent when it received the request to run the time synchronization process. This value is used by the agent to return home once it finishes the process. The latter two properties are temporary, and are used by the *parseHosts()* function described later.

TimeSync agent exposes a single service (lines 7–22) called *SyncTimers*. The service has no return value, and accepts two parameters: a comma-separated list of *MAS* instances the agent needs to visit, and the time value to be set in each *MAS*. The starting *MAS* is stored persistently in lines 8 and 9. As noted earlier, when a mobile agent arrives to a new *MAS*, its runtime state is restored after the initialization, but before any service execution is requested. This means that the expression *startingHome == null* will resolve to true only at the agent's starting *MAS*. Lines 10–13 are used to detect if the agent has returned back home. For this evaluation, the *host()* library function is invoked, returning the address of the agent's current host.

Line 16 invokes a function called *applySystemTime()*, which is used to set the system time to the given value. For security purposes, however, this is a dummy functions. *ALAS* agents are not actually able to change the system time.

Lines 18–21 are used to extract the next *MAS* that needs to visited. Line 18 in particular demonstrates the usage of the *ALAS* standard library. The *hosts* parameter is of *ALAS* String complex type, which offers a set of functions for string manipulation, including (among others):

- *int length()* – returns the length of the string.
- *int indexof(String sub)* – returns the index of the first occurrence of *sub* within the string, or -1 if the parameter does not occur. The first character in a string has the index of 0.
- *String substring(int start [, int end])* – returns the substring of the string, starting with index *start* (inclusive), and until the index *end* (exclusive). If *end* is not specified, the call corresponds to *str.substring(start, str.length())*.

Because the String type is included both in Java and Python, the *VM* selector (Fig. 1) replaces these calls with calls to appropriate native string manipulation methods.

The *SyncTimers* service utilizes a helper function, named *parseHosts()* (lines 24–31) which extracts the next *MAS* from the comma-separated list. The next *MAS* is stored into the temporary property *next*, while the updated list (e.g. the list without the extracted *MAS*) is stored into the second temporary property, *remaining*.

The agent's final step is to move to the next *MAS*, and it does so by invoking the *move* instruction (line 22). The instruction's arguments indicate that, once it reaches the target, the agent should be asked to again execute the *SyncTimers* service with parameter values *remaining* and *time*.

5.1. Running the *TimeSync* agent

The network used in this experiment consists of three *MAS*s, i.e. a single instance of each of the *SOM*, *PySOM*, and *JADE*. The network addresses of the systems are, respectively, $192.168.0.1:8081$, $192.168.0.2:8081$, and $192.168.0.3:8081$. Initially, the agent is located in the *SOM* instance.

A client can ask for the *SyncTimer* service execution by sending the following *KQML* message to the agent:

```
(achieve
  :sender ALAS_IDE
  :receiver TimeSync
  :language XML
  :content "<service>
      <name>SyncTimers</name>
      <args>
        <arg name="hosts" type="String">
          <![CDATA[192.168.0.2:8081,192.168.0.3:8081]]>
        </arg>
        <arg name="time" type="double">
          <![CDATA[40893,639141169]]>
        </arg>
      </args>
    </service>")
```

The sender of this message is *ALAS IDE*, which, among common features such as syntax highlighting, provides the means for specifying the target *MAS*, in form of $type@address$. In this scenarion, the compilation process will automatically load the agent in the specified *MAS*.

Once the agent executes the service, it will set the time of the current *MAS*, extract the network address of the next *MAS* to visit (*PySOM* instance at $192.168.0.2:8081$), and initiate the migration process. The migration process will serialize the agent, resulting in the following *XML* stream:

```
<?xml version="1.0" encoding="UTF-8"?>
<alas xmlns="http://alasagents.org/SerializedAgent"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://alasagents.org/SerializedAgent
      SerializedAgent.xsd ">
  <source>
    <![CDATA[package example.agents; agent ...]]>
  </source>
  <home>192.168.0.1:8081</home>
  <state>
    <property name="startingHome" type="String">
      <![CDATA[192.168.0.1:8081]]>
    </property>
  </state>
  <service>SyncTimers</service>
```

```
  <args>
    <arg name="hosts" type="String">
      <![CDATA[192.168.0.3:8081]]>
    </arg>
    <arg name="time" type="double">
      <![CDATA[40893,639141169]]>
    </arg>
  </args>
</alas>
```

The type information for persistent properties and service arguments may be redundant, but is included in the stream for convenience reasons.

Running the agent in *PySOM* The *PySOM* instance accepts the serialized stream, extracts the agent source code, and passes it to the *ALAS* compiler. The agent's persistent property is then restored to $192.168.0.1 : 8081$. Finally, the system sends a message to the agent, asking for the execution of the *SyncTimers* service. The generated Python source code of the *TimeSync* agent is shown in Listing 5.2 [3].

Listing 5.2. Source code of the mobile *TimeSync ALAS* agent, regenerated for *PySOM*

```
1  class TimeSync:
2    def __init__(self):
3      # original, ALAS source code of the agent
4      self.__AGENT_SOURCE_ = "package example.agents; ..."
5      # comma-separated list of persistent properties
6      self.__PERSISTENT_VARS_ = "startingHome"
7      # agent state
8      self.startingHome = None
9      self.next = None
10     self.remaining = None
11
12   # handler of incoming messages
13   def onMessage(self, service):
14     if service.getName() == "SyncTimers":
15       # SyncTimers service implementation
16       def SyncTimers(hosts, time):
17         # remember the starting point
18         if self.startingHome == None:
19           self.startingHome = host()
20         else: # am I back home?
21           if self.startingHome == host():
22             alas.stdlib.pysom.Log.write("I'm back!")
23             startingHome = None
24             return
```

[3] This and subsequent listings showing transformed source code of the agent have been manually reformatted to make them more human-readable.

```
25          # apply the time
26          alas.stdlib.pysom.Log.write("Setting the ",
27            "system time to ", time)
28          alas.stdlib.pysom.Dummy.applySystemTime(time)
29          # go to the next host
30          if len(hosts) == 0: # no more hosts, go back home
31            self.next = self.startingHome
32          else:
33            self.parseHosts(hosts)
34          # prepare service parameters
35          __mv_obj_2 = time
36          __mv_obj_1 = self.remaining
37          # start the migration process
38          alas.stdlib.pysom.PySOMFacilitator.instance().move(
39            self, self.next, "SyncTimers", "hosts", __mv_obj_1,
40            "time", __mv_obj_2)
41        # execute the service
42        SyncTimers(service.get("hosts"), service.get("time"))
43        return
44
45    def parseHosts(self, hosts):
46      n = hosts.find(",")
47      if n == -1:
48        self.next = hosts
49        self.remaining = ""
50      else:
51        self.next = hosts[0:n]
52        self.remaining = hosts[n+1:]
```

The *TimeSync* agent is represented by a class which includes a constructor (lines 2–10), a function for handling incoming messages (lines 13–43), and the helper *parseHosts()* function (lines 45–52). The actual code of the *SyncTimers* services is inserted as an inner function of *onMessage()* (lines 16–43), right under the conditional statement that determines the name of the requested service. The actual call to this inner function is made in line 42.

Agents that wish to leave a *PySOM* instance can do so by calling the *move* method of *PySOMFacilitator*. The method relies on Python reflection features for extracting the necessary runtime information about the agent, such as its original, *ALAS* source code stored in line 4, values of persistent properties listed in line 6, etc.

As it can been seen from the given source code, the *ALAS* standard library for *PySOM* is available under the *alas.stdlib.pysom* package.

After executing the service here, the agent will move to the *JADE* instance.

Running the agent in *JADE* Once the *TimeSync* agent reaches *JADE*, its *ALAS* source code will be processed to produce *JADE behavior* class, shown in Listing 5.3. *MyBehavior* is defined as an inner class of the *TimeSync* class, which represents the actual agent. State properties, original *ALAS* source code, and the list of persistent properties are all defined in this agent class.

Listing 5.3. Source code of the mobile *TimeSync ALAS* agent, regenerated for *JADE*

```
1  private class MyBehavior
2      extends jade.core.behaviours.CyclicBehaviour {
3    private TimeSync agent;
4
5    public MyBehavior(TimeSync agent) { this.agent = ag; }
6
7    @Override public void action() {
8      // wait for a message
9      jade.lang.acl.ACLMessage msg = agent.receive();
10     if (msg == null) { block(); return; }
11
12     // extract service description from the msg content
13     alas.stdlib.java.common.migration.ServiceDesc content = null;
14     try {
15       content = (alas.stdlib.java.common.migration.ServiceDesc)
16         msg.getContentObject();
17     } catch (jade.lang.acl.UnreadableException ex) { return; }
18
19     if (content.isService("SyncTimers")) {
20       // SyncTimers service implementation
21       class Service_SyncTimers {
22         void SyncTimers(String hosts, double time) {
23           // remember the starting point
24           if (startingHome == null)
25             startingHome = alas.stdlib.java.common.Facilitator.
26               instance().getHome();
27           // am I back home?
28           else if (startingHome.equals(alas.stdlib.java.
29               common.Facilitator.instance().getHome())) {
30             alas.stdlib.java.common.Log.write("I'm back!");
31             startingHome = null;
32             return; }
33           // apply the time
34           alas.stdlib.java.common.Log.write("Setting the ",
35             "system time to ", time);
36           alas.stdlib.java.common.Dummy.applySystemTime(time);
37           // go to the next host
38           if (hosts.length() == 0) // no more, go back home
39             next = startingHome;
40           else
41             parseHosts(hosts);
42           // prepare service parameters
43           Object mv_obj2 = time;
44           Object mv_obj1 = remaining;
45           // start the migration process
46           alas.stdlib.java.jade.JADEFacilitator.instance().
47             move(this, next, "SyncTimers",
```

```
48                "hosts", mv_obj1, "time", mv_obj2); } }
49        // execute the service
50        String hosts = content.get("hosts", String.class);
51        double time = content.get("time", double.class);
52        new Service_SyncTimers().SyncTimers(hosts, time);
53        return; } }
54
55    private void parseHosts (String hosts){
56      int n = hosts.indexOf(",");
57      if (n == -1) {
58        next = hosts;
59        remaining = "";
60      } else {
61        next = hosts.substring(0, n);
62        remaining = hosts.substring(n + 1); } } }
```

Inside *JADE*, the agent will wait for an incoming message (lines 9 and 10), and then extract the message content (lines 13–17). The content is defined as a *ServiceDesc* class, which stores all the information about the service the agent is asked to execute. The actual source code of the *SyncTimers* service is inserted as the inner *Service_SyncTimers* class (lines 21–48), and it's invoked using lines 50–52.

Two root packages, *alas.stdlib.java.common* and *alas.stdlib.java.jade*, make up the *ALAS* standard library for *JADE*. All classes under the first package are shared between all Java-based *MAS*s (in this case, *SOM* and *JADE*) and include functionalities that are architecture-independent. The second package incorporates classes that implement *JADE*-specific behavior (e.g. *JADEFacilitator*).

Similarly as with *PySOM*, the *move* instruction (lines 46–48) relies on Java reflection *API* to extract and serialize agent's runtime properties.

After executing the service in *JADE*, the agent determines that there are no more *MAS*s to visit, and returns to its home – the *SOM* instance.

Running the agent in *SOM* The *TimeSync* source code produced for *SOM* is shown in Listing 5.4. Again, the agent is defined as a regular Java class implementing the *SOM*-specific *Agent* interface that represents all agents.

Listing 5.4. Source code of the mobile *TimeSync ALAS* agent, regenerated for *SOM*

```
1  public class TimeSync
2        implements xjafs.agentmanager.ejb.interfaces.Agent {
3    // original, ALAS source code of the agent
4    private final String __AGENT_SOURCE__ =
5      "package example.agents; ...";
6    // comma-separated list of persistent properties
7    private final String __PERSISTENT_VARS__ = "startingHome";
8    // agent state
9    private String startingHome, next, remaining;
```

```
10
11    // handler of incoming messages
12    @Override
13    public void onKQMLMessage(String kqml, String agentID) {
14      // unmarshall the KQML message
15      final xjafs.agentmanager.ejb.utils.xml.kqml.KqmlMessage msg =
16      xjafs.agentmanager.ejb.utils.XMLMapper.unmarshallKQML(kqml);
17
18      if (message.getCommand().equals("SyncTimers")) {
19        // SyncTimers service implementation
20        class Service_SyncTimers {
21          void SyncTimers(String hosts, double time) {
22            // remember the starting point
23            if (startingHome == null)
24              startingHome = alas.stdlib.java.common.Facilitator.
25                instance().getHome();
26            // am I back home?
27            else if (startingHome.equals(alas.stdlib.java.common.
28                Facilitator.instance().getHome())) {
29              alas.stdlib.java.common.Log.write("I'm back!");
30              startingHome = null;
31              return; }
32            // apply the time
33            alas.stdlib.java.common.Log.write("Setting the ",
34              "system time to ", time);
35            alas.stdlib.java.common.Dummy.applySystemTime(time);
36            // go to the next host
37            if (hosts.length() == 0) // no more, go back home
38              next = startingHome;
39            else
40              parseHosts(hosts);
41            // prepare service parameters
42            Object __mv_obj_$2 = (time);
43            Object __mv_obj_$1 = (remaining);
44            // start the migration process
45            alas.stdlib.java.som.SOMFacilitator.instance().move(
46              this, next, "SyncTimers",
47              "hosts", __mv_obj_$1, "time", __mv_obj_$2); } }
48        // execute the service
49        SyncTimers task = XMLMapperSyncTimers.
50          unmarshallSyncTimers(msg.getContent());
51        new Service_SyncTimers().SyncTimers(task.getHosts(),
52          task.getTime());
53        return; } }
54
55    private void parseHosts (String hosts) { ... ommitted ... }
56  }
```

SOM agents communicate by exchanging *KQML* messages, and the system relies on *JAXB marshalling/unmarshalling* [19] for message serialization/dese-

rialization (lines 15 and 16). Similarly as with *JADE*, in this *SOM*-specific definition of *TimeSync*, the *SyncTimers* service implementation is provided in form of an inner class (lines 20–47), and is invoked using lines 49–52.

This experiment demonstrates how *ALAS* can be used to develop agents in the *write once, run anywhere* manner. The *ALAS*-based *TimeSync* agent is implemented once to solve a specific problem, and is then able to work in *SOM*, *PySOM*, and *JADE* instances without any interventions on the developer's part. Therefore, the main goal behind the development of *ALAS* has been achieved.

6. Conclusions and future work

XJAF is *FIPA*-compliant *MAS* developed by the authors of this paper. It is designed as a pluggable, manager-based architecture, which allows for easy additions of new functionalities. The system is implemented in Java EE, today's leading development platform for building large-scale, scalable, secure, and reliable software. In the course of improving its interoperability, *XJAF* has recently been redesigned as a service-oriented architecture. The resulting system, named *SOM*, follows the manager-based approach of *XJAF*, but with managers being implemented as web services. The main advantage of this approach is that external clients and third-party tools can use *SOM* and interact with its agents through *SOAP*, the standardized communication protocol.

SOM is a conceptual specification of web services, and it can be implemented using many modern programming languages. But, this poses a major problem: an agent written for, e.g., Java-based implementation of *SOM* cannot move to a Python-based implementation. In order to overcome this issue, a new agent-oriented programming language named *ALAS*, has been proposed. The two main goals of *ALAS*, as originally described in [22], are:

1. To provide developers with programming constructs that simplify the overall complexity of agent development.
2. To include tools for agent code regeneration, and enable migration across *SOM* instances implemented using different programming languages.

Since this original proposal, however, the design goal of *ALAS* has been extended. The language itself and its accompanying set of tools have been upgraded to support true *heterogeneous* agent mobility. Unlike the agent regeneration, heterogeneous mobility assumes that agents are able to migrate across the network consisting of *MAS* instances that offer different sets of *API*s *and* are implemented using different programming languages. This, obviously, is more difficult problem to solve, as it requires both the regeneration of agent's executable code and modifications of the code in order to adapt to the *API* of the underlying *MAS*.

As shown in this paper, the desired goal has been achieved. *ALAS* agents are able to seamlessly, without any interventions on the developer's part, operate inside Java-based *SOM*, Python-based *SOM*, and *JADE* instances. The

presented experiment demonstrates how the agent's executable code is regenerated and modified transparently, on-the-fly, to suit the requirements of the underlying *MAS*. This enables the developer to focus on solving the concrete problem, and, in a truly platform-independent manner, disregard information about the target *MAS*. To the best of our knowledge, there currently exists no other agent-oriented programming language that offers this significant benefit.

Future research directions will be concentrated onto improving the expressive power of *ALAS* and extending its standard library of functions. This will simplify the agent development process even further.

The support of other *MAS*s is planned as well.

In the long run, the language will be enriched with programming constructs for defining agent's beliefs, desires, intentions, and goals, in order to support the development of *BDI*-style agents.

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

References

1. Aranda, G., Palanca, J., Criado, N.: SPADE user's manual. http://spade.gti-ia.dsic.upv.es/manuals/html-chunk/index.html (October 2007), retrieved on December 7, 2011
2. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing multi-agent systems with JADE. John Wiley and Sons (2007)
3. Bordini, R.H., Wooldridge, M., Hübner, J.F.: Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology). John Wiley & Sons (2007)
4. Bradshaw, J., Breedy, M., Groth, P., Hill, G., Jeffers, R., Mitrovich, T., Suri, N.: An overview of the NOMADS mobile agent system. In: 2nd International Symposium on Agent Systems and Applications, ASA/MA2000 (September 2000)
5. Dastani, M.: 2APL: a practical agent programming language. Autonomous Agents and Multi-Agent Systems 16(3), 214–248 (2008)
6. Dastani, M., van Riemsdijk, B., Dignum, F., Meyer, J.J.C.: A programming language for cognitive agents - goal directed 3APL. In: Dastani, M., Dix, J., Fallah-Seghrouchni, A.E. (eds.) PROMAS. Lecture Notes in Computer Science, vol. 3067, pp. 111–130. Springer (2003)
7. Davies, W.H.E., Edwards, P.: Agent-K: An integration of AOP and KQML. In: Proceedings of the Third International Conference on Information and Knowledge Management (1994)
8. Finin, T., Fritzson, R., McKay, D., McEntire, R.: KQML as an agent communication language. In: Proceedings of the third international conference on Information and knowledge management. pp. 456–463. CIKM '94, ACM, New York, NY, USA (1994), http://doi.acm.org/10.1145/191246.191322
9. FIPA abstract architecture specification. http://www.fipa.org/specs/fipa00001/SC00001L.pdf (2002), retrieved on December 7, 2011
10. FIPA ACL message structure specification. http://www.fipa.org/specs/fipa00061/SC00061G.pdf (2002), retrieved on December 7, 2011

11. FIPA homepage. http://www.fipa.org/, retrieved on December 7, 2011
12. Fortino, G., Garro, A., Russo, W.: Achieving mobile agent systems interoperability through software layering. Information and software technology 50(4), 322–341 (2008)
13. Grimstrup, A., Gray, R.S., Kotz, D., Carvalho, M.M., Cowin, T.B., Chacón, D.A., Barton, J., Garrett, C., Hofmann, M.: Toward interoperability of mobile-agent systems. In: International symposium on mobile agents. pp. 106–120 (2002)
14. Hapner, M., Burridge, R., Sharma, R., Fialli, J., Stout, K.: Java Message Service (JMS) specification. http://www.oracle.com/technetwork/java/jms/index.html (April 2002), retrieved on December 7, 2011
15. Hindriks, K.V.: Programming rational agents in GOAL. In: El Fallah Seghrouchni, A., Dix, J., Dastani, M., Bordini, R.H. (eds.) Multi-Agent Programming:, pp. 119–157. Springer US (2009)
16. Ivanović, M., Mitrović, D., Budimac, Z., Vidaković, M.: Metadata harvesting learning resources – an agent-oriented approach. In: Proceedings of the 15th International Conference on System Theory, Control and Computing (ICSTCC 2011). pp. 306–311 (October 2011)
17. JADE homepage. http://jade.tilab.com/, retrieved on December 7, 2011
18. Jason homepage. http://jason.sf.net/, retrieved on December 7, 2011
19. Java Architecture for XML Binding (JAXB) homepage. http://www.oracle.com/technetwork/articles/javase/index-140168.html, retrieved on December 7, 2011
20. Mitrović, D., Budimac., Z., Ivanović, M., Vidaković, M.: Improving fault-tolerance of distributed multi-agent systems with mobile network-management agents. In: Proceedings of the International Multiconference on Computer Science and Information Technology. vol. 5, pp. 217–222 (October 2010)
21. Mitrović, D., Budimac, Z., Ivanović, M., Vidaković, M.: Agent-based approaches to managing fault-tolerant networks of distributed multi-agent systems. Multiagent and Grid Systems 7(6), 203–218 (December 2011)
22. Mitrović, D., Ivanović, M., Vidaković, M.: Introducing ALAS: a novel agent-oriented programming language. In: Simos, T.E. (ed.) Proceedings of Symposium on Computer Languages, Implementations, and Tools (SCLIT 2011) held within International Conference on Numerical Analysis and Applied Mathematics (ICNAAM 2011). pp. 861–864. AIP Conf. Proc. 1389 (September 2011), iSBN 978-0-7354-0956-9
23. Moreau, L.: Distributed directory service and message router for mobile agents. Science of Computer Programming 39(2–3), 249–272 (2001)
24. Overeinder, B.J., Groot, D.R.A.D., Wijngaards, N.J.E., Brazier, F.M.T.: Generative mobile agent migration in heterogeneous environments. Scalable computing: practice and experience 7(4), 89–99 (2006)
25. Pinsdorf, U., Roth, V.: Mobile agent interoperability patterns and practice. In: Proceedings of the 9th IEEE international conference on engineering of computer-based systems. pp. 238–244 (2002)
26. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: de Velde, W.V., Perram, J.W. (eds.) MAAMAW. Lecture Notes in Computer Science, vol. 1038, pp. 42–55. Springer (1996)
27. Schmidt, D.C.: Model-driven engineering. Published by IEEE Computer Society (February 2006)
28. Shoham, Y.: Agent-oriented programming. Artificial Intelligence 60(1), 51–92 (1993)
29. SPADE homepage. http://code.google.com/p/spade2/, retrieved on December 7, 2011

30. Suri, N., Bradshaw, J., Breedy, M.R., Groth, P.T., Hill, G.A., Jeffers, R.: Strong mobility and fine-grained resource control in NOMADS. In: Kotz, D., Mattern, F. (eds.) Proceedings of the Second international Symposium on Agent Systems and Applications and Fourth international Symposium on Mobile Agents. Lecture Notes In Computer Science, vol. 1882, pp. 2–15 (September 2000)
31. Tapia, D.I., Bajo, J., Corchado, J.M.: Distributing functionalities in a SOA-based multi-agent architecture. In: Demazeau, Y., Pavón, J., Corchado, J.M., Bajo, J. (eds.) 7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 2009), Advances in Intelligent and Soft Computing, vol. 55, pp. 20–29. Springer Berlin / Heidelberg (2009)
32. Tapia, D.I., Rodríguez, S., Bajo, J., Corchado, J.M.: FUSION@, a SOA-based multi-agent architecture. In: Corchado, J.M., Rodríguez, S., Llinas, J., Molina, J. (eds.) International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008), Advances in Soft Computing, vol. 50, pp. 99–107. Springer Berlin / Heidelberg (2009)
33. Thomas, S.R.: The PLACA agent programming language. In: Wooldridge, M., Jennings, N.R. (eds.) ECAI Workshop on Agent Theories, Architectures, and Languages. Lecture Notes in Computer Science, vol. 890, pp. 355–370. Springer (1994)
34. Vidaković, M.: Extensible Java based agent framework. Ph.D. thesis, Faculty of Technical Sciences, University of Novi Sad, Serbia (2003)
35. Vidaković, M., Sladić, G., Konjović, Z.: Security management in J2EE based intelligent agent framework. In: Proceedings of the 7th IASTED International Conference on Software Engineering and Applications (SEA 2003). pp. 128–133 (November 2003)
36. World Wide Web Consortium (W3C) SOAP version 1.2. http://www.w3.org/TR/soap/, retrieved on December 7, 2011
37. World Wide Web Consortium (W3C) XML Schema. http://www.w3.org/XML/Schema, retrieved on December 7, 2011
38. Winikoff, M.: JACK Intelligent Agents: an industrial strength platform. In: Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E. (eds.) Multi-Agent Programming, Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 15, pp. 175–193. Springer (2005)
39. Wooldridge, M., Jennings, N.: Agent theories, architectures, and languages: A survey. In: Wooldridge, M., Jennings, N. (eds.) Intelligent Agents, Lecture Notes in Computer Science, vol. 890, pp. 1–39. Springer Berlin / Heidelberg (1995)
40. Wooldridge, M., Jennings, N.: Intelligent agents: Theory and practice. Knowledge Engineering Review 10, 115–152 (1995)
41. XMPP standards foundation homepage. http://xmpp.org/, retrieved on December 7, 2011

Dejan Mitrović is a teaching and research assistant at Faculty of Sciences, University of Novi Sad, Serbia. He graduated in 2006 (Informatics), and received master's degree (Computer Science) in 2008, enrolling the PhD studies afterwards. He published 11 research papers on software agents, multi-agent systems, and distributed computing. He is a member of several science research projects.

Mirjana Ivanović holds position of full professor since 2002 at Faculty of Sciences, University of Novi Sad, Serbia. She is head of Chair of Computer Sci-

ence. She is author or co-author of 13 textbooks and of more then 230 research papers on multi-agent systems, e-learning and web-based learning, software engineering education, intelligent techniques (CBR, data and web mining), most of which are published in international journals and international conferences. She is/was a member of Program Committees of more then 80 international Conferences and is Editor-in-Chief of Computer Science and Information Systems Journal.

Zoran Budimac holds position of full professor since 2004 at Faculty of Sciences, University of Novi Sad, Serbia. Currently, he is head of Computing laboratory. His fields of research interests involve: Educational Technologies, Agents and WFMS, Case-Based Reasoning, Programming Languages. He was principal investigator of more then 20 projects and is author of 13 textbooks and more then 220 research papers most of which are published in international journals and international conferences. He is/was a member of Program Committees of more then 60 international Conferences and is member of Editorial Board of Computer Science and Information Systems Journal.

Milan Vidaković received the BSc, MSc and PhD degrees in electrical engineering from the Faculty of Technical Sciences, University of Novi Sad, in 1995, 1998 and 2003 respectively. He is a professor at Computing and Control Department, University of Novi Sad. He participated in several science projects and published more than 60 scientific and professional papers. His research interest covers web and internet programming, distributed computing, software agents, embedded systems, and language internationalization and localization.

Received: January 2, 2012; Accepted: Jun 12, 2012.