Domain specific agent-oriented programming language based on the Xtext framework

Dejan Sredojević*, Dušan Okanović**, Milan Vidaković**, Dejan Mitrović***, Mirjana Ivanović***
* Novi Sad Business School, Novi Sad, Serbia
** University of Novi Sad/Faculty of Technical Sciences, Novi Sad, Serbia
*** University of Novi Sad/Faculty of Sciences, Novi Sad, Serbia
dsredojevic.vps@gmail.com, {oki, minja}@uns.ac.rs, {dejan, mira}@dmi.uns.ac.rs

Abstract— **The agent technology represents one of the most consistent approaches to the development of distributed systems. Multiagent middleware XJAF, developed at the University of Novi Sad, presents a runtime environment that supports the execution of software agents. To solve the problem of interoperability, we propose a domain-specific agent language named ALAS, whose main purpose is to support the implementation and execution of agents on heterogenous platforms. To define the structure of the language, a metamodel and a grammar of the ALAS language has been created, in accordance with the requirements and needs of the agents. This paper describes the construction of the compiler and the generation of executable Java code that can be executed in XJAF.**

I. INTRODUCTION

A software agent is a program that works autonomously while performing tasks that are assigned to it [1]. These are target-oriented computer programs which react to their own environment. They work without direct supervision and perform tasks for the end user or other programs. Features of software agents include autonomy, intelligence, mobility, persistence and communication [2]. Autonomy implies that agents must be able to independently perform tasks. Mobility implies that agents have ability to leave the place where they currently execute a task and to continue the execution of the task on another node in the network [3][4]. Communication implies that agents must be able to communicate with other agents in the system.

A system that consists of several software agents is called a *multiagent system* - MAS. Such agents are capable of collectively solving a task that is most difficult to be solved by a single agent or monolithic system. Its main features include agent lifecycle management, messaging, security mechanisms, and service subsystem that gives agents the ability to access resources, execute complex algorithms, etc [1].

The rest of the paper is organized as follows. The *Related work* section describes a couple of existing agent-oriented programming languages (AOPL) that had a strong influence on the development of ALAS. The multiagent middleware *Extensible Java EE-based Agent Framework* (XJAF) [5] is described in the third section. In the fourth section of the paper, an agent-oriented programming language ALAS is described [4][6][7]. The fifth section presents the results of testing the ALAS using the Eclipse framework, along with the Xtext-based plugin installed. Last section gives concluding remarks.

II RELATED WORK

Agent-oriented programming (AOP) [8] is a software development paradigm aimed at efficient development of software agents and multi-agent systems. AOP shares many features with object-oriented programming (OOP), and it is based on agents which definition include agent state, actions, services and messaging system. Since object-oriented and agent-oriented paradigm share many programming concepts, development of an OOP-inspired AOPLs is a natural process.

One of the first AOP languages was AgentSpeak. The AgentSpeak programming language was introduced in [9]. It is a natural extension of logic programming for the beliefs desires-intentions (BDI) agent architecture, and provides an abstract framework for programming BDI agents.

JACK [10] is a light-weight framework for rapid development of multi-agent systems. It is based on the Java programming language, but offers new keywords and language constructs. The accompanying compiler produces pure Java code, which allows for each JACK agent to be used plain Java object.

Agent mobility is an essential property of agents. However, this property can be quite complex to implement. Any AOPL should hide this complexity from end-users. SAFIN [11] and CLAIM [12] hide complex support for agent mobility from the programmer. They hide the functional complexity from developers by providing them with simple, yet powerful programming constructs.

JIAC V [13] is a multi-agent system that can execute agents developed in pure Java or by using an accompanying AOPL named JADL++. Similarly to AgentSpeak, an action of a JADL++ agent can be either private, for internal use, or public, in which case it is called a *service* and offered to other part of the system.

We have introduced the early version of the ALAS programming language in papers [6] and [7]. This early version have used *javacc* [16] parser generator, while this paper proposes the Xtext framework for the language development. We have decided to use Xtext, since we wanted to start with the ECore metamodel, to separate validation from compilation and to have Eclipse plugin made for ALAS without additional programming. This plugin offers both validation and code generation to any programming language.

III XJAF DEVELOPMENT FRAMEWORK

XJAF is a multiagent middleware based on the FIPA standards [14]. FIPA is a non-profit organization which has produced a set of specifications that enable interaction between the agent and the framework, as well as interaction between agents.

The main tasks of XJAF are to provide an efficient environment for the execution of its agents and to provide a reusable interface to external clients [2]. XJAF is designed as a modular system that contains specialised modules called *managers*. Each manager is relatively independent module responsible for handling certain agent management tasks.

The latest version of XJAF is focused on using the advantages of computer clusters [15]:

- Load balancing – XJAF agents are automatically distributed in a cluster in order to reduce the load on individual computer nodes.
- State replication and failover: state of each agent is copied to other nodes making them resistant to hardware and software faults.

The main drawback of the original XJAF architecture was the fact that it was limited to the Java programming language. The consequence of this approach was that the agents needed to be written in Java, and could not interact with agents in other, non-Java frameworks. To increase the interoperability and enable its wider usage, XJAF has been redesigned as a service-oriented architecture - SOA. The multiagent framework based on SOA called SOM - *SOA-based multiagent system*, kept Java EE as the implementation platform, but managers were redefined as web services [3][6]. This enabled that implementation of SOM could be done in many modern programming languages that support web services (Java, JavaScript, Python, C#, etc.). Interoperability is also increased and external clients and independent tools can employ agents through web service interfaces.

However, the use of web services does not solve all the problems. Considering that other programming languages can be used for the implementation of an agent framework, it is impossible to write an agent that will successfully execute on any platform. The problem becomes apparent when the agents that are written in different programming languages, moving through the network arrive to a SOM that is implemented in some other programming language. For example, agent developed using Java programming language can not be easily adapted to agent framework implemented for example, in Python. To solve this problem the authors of XJAF have developed a new agent language - ALAS [7].

IV SPECIFICATION OF ALAS LANGUAGE

The main goals of ALAS are:

- *Hot compilation* - to ensure that the agents can be executed in target platform, regardless on the underlying programming language,
- *Hiding complexity* of agent development from programmers.

Agents must adapt to the environment in which they arrive. When they arrive to some framework that is implemented in a programming language X, they must be automatically transformed to source code written in X.

According to these requirements it is necessary to implement a compiler for the ALAS language. Input parameters for this compiler are the original file written in ALAS, and the identification of the destination platform, such as Java, JavaScript, Python C#, etc. Depending on these parameters, the compiler generates executable code depending on the destination platform. If the platform on which the agent arrived has been implemented in the Java programming language, it will be Java byte code. In this way, developers are able to focus on solving concrete tasks and do not have to take care about interoperability or details of the implementation of SOM.

A. Development of ALAS using Xtext framework

Due to restrictions of the *javacc* system previously used for ALAS transformation [16], development of ALAS has been restarted using Xtext. This framework is most commonly used for domain-specific language development [17].

Xtext enables the development of agent-based domain-specific language ALAS, so it could meet the aforementioned requirements. The development of domain-specific language using Xtext is performed using specialized languages - Java and Xtend. Xtend is a statically-typed programming language which translates to comprehensible Java source code. Syntactically and semantically Xtend has its roots in the Java programming language but it is improved in many aspects. Xtext is an open-source framework for the development of domain-specific languages - DSL. It is based on ANTLR [18]. Unlike standard parser generators, Xtext not only generates a parser, but also a class model for the abstract syntax tree and a fully featured, customizable Eclipse-based IDE.

B. Domain-specific language ALAS modeling

In the last decade, great advancements appeared in the modeling field and defining of domain-specific languages. These languages allow developers and domain experts to focus on specific domain problems instead of dealing with the programming language features. Formation of OMG (Object Management Group), an organization that gathered the most important industrial subjects in order to establish standards in the field, has greatly contributed to the acceleration of the development methodology for designing software controlled by models [19]. OMG initiative called MDA (Model-Driven Architecture) represents one of the most important movements in the development of software controlled by models. MDA is a specialization of a broader approach called MDE (Model-Driven Engineering) which is a development metodology focused on creating domain models. The domain-specific languages are programming languages designed for solving problems in a specific, clearly defined, domain.

MOF (Meta-Object Facility) is an OMG standard, which represents the core of infrastructure for support to MDA [20]. MOF is a specification that enables

cooperation between different domains and different modeling language and is a mechanism for formally defining modeling languages, i.e. metamodels. MOF is a four layered architecture whose layers are marked as M0, M1, M2 and M3 (Fig. 1). MOF can be viewed as an abstract syntax of DSL which used to define the metamodel.

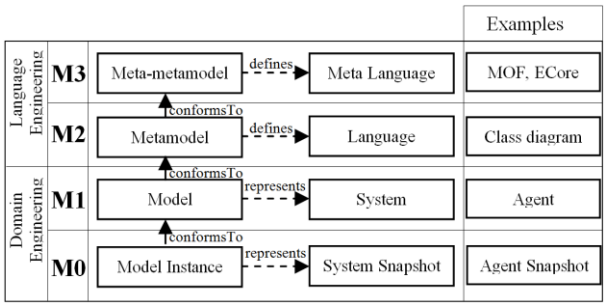

Figure 1. Four-layered metamodel architecture

There are several implementations of MOF infrastructure which more or less follow specification standards. Implementation that is used to create a metamodel of ALAS is ECore. ECore was developed by the Eclipse Foundation and EMF project, which basically is an implementation of EMOF 2.0. ECore is also open source and is part of the Eclipse platform [21].

C. *Implementation of ALAS metamodels, by using ECore meta-metamodel*

To define domain-specific language ALAS, a metamodel has been created first i.e. class diagram of our domain-specific language. ALAS metamodel is implemented by using ECore meta-metamodel which is a part of Eclipse framework.

To represent the metamodel graphically, the Eclipse framework uses Sirius 2.0 plugin [22]. ALAS ECore metamodel may be defined manually, using the tree editor, but this kind of modeling is tiresome and impractical. Other, often used method is to define the metamodel using the class diagram which is accessible in existing modeling tools. In Fig. 2 you can see the class diagram, which defines the structure of the domain-specific language ALAS.

D. *Grammar of ALAS language*

Created metamodel is then used for automatic generation of the ALAS grammar. After the integration of Xtext plugin into the Eclipse framework, it is possible to create an Xtext project which will implement the previously mentioned ECore metamodel. The grammar of ALAS language is shown in Listing 1. Considering that the graphic representation of the ALAS metamodel does not fully define the structure of the language, it is necessary to modify the generated grammar by adding keywords and identifiers that are listed under the single quotes and new rules which can be created only after import of certain packages.

Considering that Xtext is compatible with Java programming language, existing Java based rules such as: `XimportSection`, `XblockExpression`, `XExpression`, `XMoveExpression`, `XprimaryExpression` and `XvariableDeclaration` can be usedafter import package from Listing 1, line 4. These rules are automatically added to ECore metamodel of the language and marked as red frame in Fig. 2.

Each rule in the grammar represents an appropriate class in the ALAS metamodel diagram. The grammar contains a set of declared events, variables, functions, services, actions and agent states. The grammar is based on the EBNF - Extended Backus-Naur Form - Listing 1.

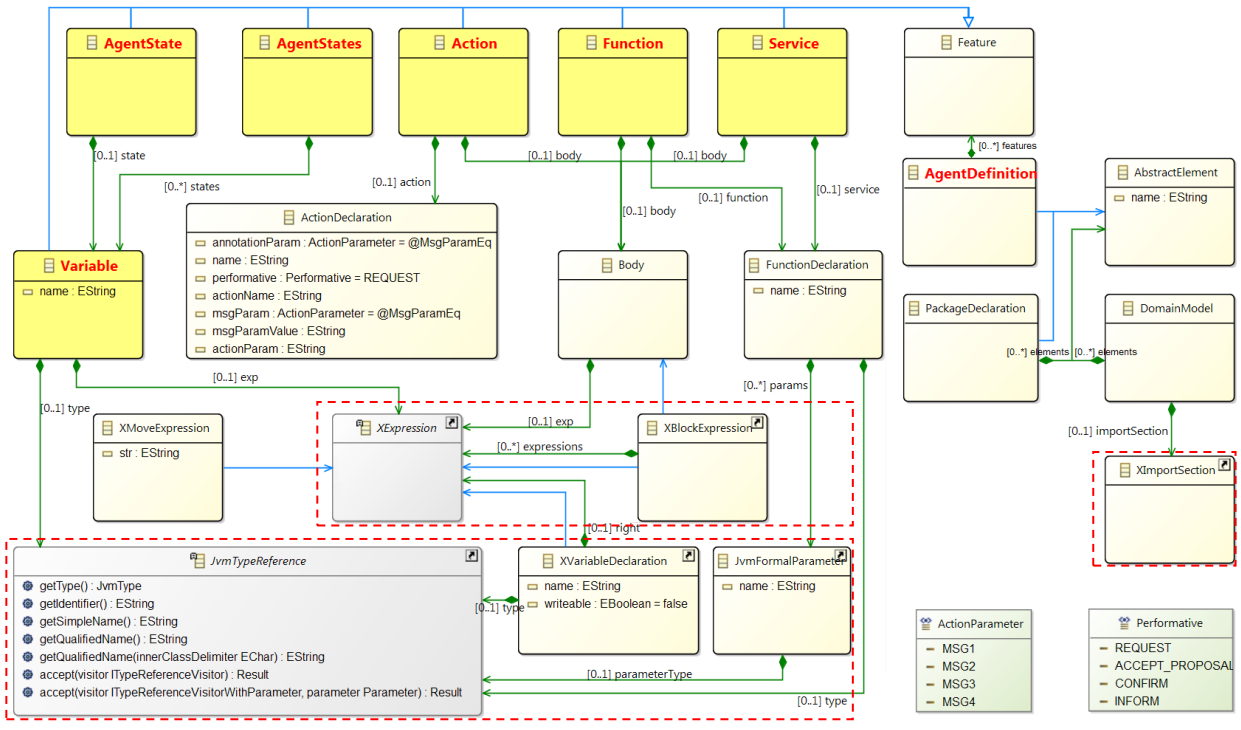

Figure 2. ALAS metamodel

```
1.  grammar rs.ac.uns.alas.Alas with
org.eclipse.xtext.xbase.Xbase
2.
3.  generate alas
"http://www.ac.uns/rs/alas/Alas"
4.  import
"http://www.eclipse.org/xtext/xbase/Xbase"
5.
6.  DomainModel:
7.      importSection = XImportSection?
8.      Elements += AbstractElement* ;
9.
10. AbstractElement:
11.     PackageDeclaration | AgentDefinition ;
12.
13. PackageDeclaration:
14.     'package' name = QualifiedName '{'
15.     Elements += AbstractElement*
16.     '}' ;
17.
18. AgentDefinition:
19      'agent' name = ValidID '{' features +=
         Feature* '}' ;
20.
21. Feature:
22.  =>AgentStates | AgentState | =>Variable |
       Function | Service | Action;
23.
24. AgentStates:{AgentStates}
25.     'state' '{'states += Variable* '}' ;
26.
27. AgentState:
28.     'state' state = Variable ;
29.
30. Variable:
Type = JvmTypeReference name = ValidID ( '='
exp = XExpression)?';' ;
31.
32. Function:
33.     Function = FunctionDeclaration body =
Body ;
34.
35. Service:
36.     'service' service =
FunctionDeclaration body = Body ;
37.
38. Action:
39.     Action = ActionDeclaration body = Body
;
40.
41. FunctionDeclaration:
43.     type = JvmTypeReference name=ValidID
44.     '('(params += FullJvmFormalParameter
(',' params += FullJvmFormalParameter)*)?')' ;
45.
46. XMoveExpression returns XExpression:
47.     {XMoveExpression}'move'
       '('(str=STRING)')' ';' ;
```

Listing 1. Part of ALAS grammar

After any change in the language grammar, it is necessary to generate the appropriate artifacts, i.e. language infrastructure.

The first rule in the grammar - `DomainModel` is always used as an input or initial rule. In this case, the

```
1. int value = 1;
2. String str = "host";
```

Listing 2. An example of variable definitions by using 'Variable' rule

`DomainModel` may or may not (quantifier ?) contain an imports. Also, it contains an arbitrary number (quantifier *) of `AbstractElement` rules that will be added to the parameter elements (quantifier **+=**). Within the `AbstractElement` rule the `PackageDeclaration` or `AgentDefinition` rule (quantifier '|') can be used. Within the `PackageDeclaration` rule the name of the package is defined the program will be written and again within the same package – `PackageDeclaration`, the choice can be made between `PackageDeclaration` rule or `AgentDefinition` rule.

`AgentDefinition` rule defines an agent. Agent is defined with keyword *`agent`*, then his name, and then the body of an agent in brackets ('{' and '}'). Agent can contain an arbitrary number of `Feature` rules. The `Feature` rule allows us to write any of the following six rules: `AgentStates`, `AgentState`, `Variable`, `Function`, `Service` or `Action`. Some of these rules can be added within another, and this can lead to compiler error, since the compiler does not know which rule to execute. To prevent this, we use the '**=>**' quantifier. It gives an advantage to the rule, in front of which is located. This way the compiler first applies this rule.

The difference between the first two rules, `AgentStates` and `AgentState` is as follows: within the `AgentStates` rule an arbitrary number of `Variable` rules can be defined, and within the `AgentState` rule only one `Variable` rule can be defined. This definition enables us to define a variable in the form in Listing 2.

One of the future goals of the ALAS programming language will be to provide mobility of an agent. One of the functions that will be used for this purpose is the *`'move'`* command - `XMoveExpression`. This command is specified in grammar of the language and is implemented within the `XblockExpresion`. To achieve this, the `XPrimaryExpression` rule has been redefined from the source of the Xbase grammar by adding the `XMoveExpression` rule. The `XMoveExpression` rule introduces a new keyword *`'move'`* and within the brackets that follow a String argument is placed. This argument represents an address to which the agent shall move (Listing 4, line 32).

Because the agent is written in ALAS, regardless of the task or problem that it solves, it can not be used as such in some environment that uses a different programming language. This is the appropriate moment to introduce the conversion - mapping of an agent to the destination programming language. The next section will describe the process of generating Java code from a program written in ALAS.

E. Translating program from ALAS agent language to Java code

To generate code for target platform, from an agent written in ALAS language, we can use automatically generated class that is provided by Xtext – it is generated with other language infrastructure constructs. The name of the used grammar must be provided at the beginning of the grammar file. ALAS uses the `Xbase.xtext` grammar, which can be seen in the line 1 of Listing 1. Based on the built-in grammar that is used, the compiler generates packages and classes that will be used for mapping to the concrete programming language. If the grammar uses `Terminals.xtext`, it will generate the package *generator* and within it classes that will be used

for mapping to different programming languages. Since the Xtext is closely related to the Java programming language and ALAS is Java-like language, the `Xbase.xtext` grammar is used. The parser will not generate the *generator* package but *jvmmodel* package and within it `AlasJvmModelInferrer` class that will be used to translate an agent written in ALAS to the Java program.

The `AlasJvmModelInferrer` class is implemented in the Xtend programming language. The main goal of the ALAS is that it can be transformed to any programming language, not only to Java. To achieve this, it was necessary to reimplement the `AlasJvmModelInferrer` class with all the rules that will be used by the parser to generate any destination code. To do so it was necessary to implement the `AlasModelGenerator` class and the `doGenerate` method, which generates destination code using rules from the `AlasJvmModelInferrer` class (Fig. 3).

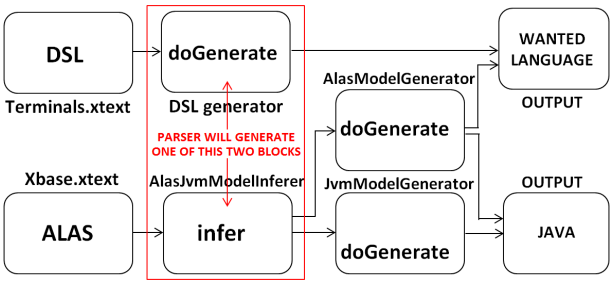

Figure 3. Working principle of parser generators

To implement a *service*, rules in the Listing 3 were used. Listing 4, lines 14 to 33, shows a service written in the ALAS language, while Listing 5, lines 18-28, shows Java implementation of that service. The Java code was generated from code in the Listing 4, applying the rules from the Listing 3. The *parameter* keyword was used to define *service* parameters.

```
1. Service : {
2.    members += f.toMethod(f.service.name,
         f.service.type ?: inferredType) [
3.      documentation = f.documentation
4.      visibility = JvmVisibility.PRIVATE
5.      for (p : f.service.params) {
6.         parameters +=
      p.toParameter(p.name, p.parameterType)
7.      }
8.      body = f.body.exp
9.    ]
10.}
```

Listing 3. The rules for mapping services from ALAS to Java

Considering that agents are Java-like and that writing the body of services, functions and actions uses `XblockExpression`, the parser easily generates the body by applying the *body* rule. If the target language is not Java, the process of generating code is more complex and mapping is then performed in the `AlasModelGenerator` class.

F. Agents code validation

One of the main advantages of a domain-specific language is the possibility of static validation of code segments. Xtext provides the ability to define validation rules and constraints. During the generation process an

```
1. package example.agents{
2.
3.    agent TimeSync {
4.
5.      state {
6.        String startingHome;
7.        String next;
8.      }
9.      state String remaining;
10.
11.     String host = "192.168.0.1";
12.     int n;
13.
14.     service void syncTimers (String hosts,
double time){
15.       if(startingHome == null){
16.         startingHome = host;
17.       }
18.       else if (startingHome.equals(host)){
19.         System.out.println("I'm back home");
20.      startingHome = null;
21.       }
22.       //apply the time
23.         print("Setting the system time to "+
time);
24.         print("ALAS command:
applySystemTime(time)");
25.
26.       //go to the next host
27.       if(hosts.length() == 0) {
28.         next = startingHome;
29.       }
30.       else
31.         parseHosts(hosts);
32.         move("192.168.0.2");
33.     }...
```

Listing 4. Part of test.alas

appropriate *validator* package is automatically generated. This package contains an implementation of the validation rules and required constraint definitions.

The `AlasValidator` class implements various constraints: checking the validity of local and global variables, checking arguments and names of functions, services and activities. In order to invoke validation, the `@Check` annotation has been introduced. This annotation triggers the validation process. An example of validation triggering is shown in the Fig. 4.

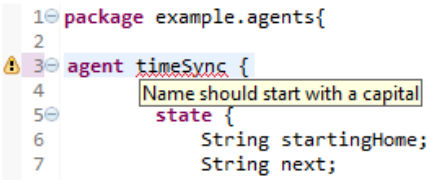

Figure 4. Example of Validation

V TESTING AND RESULTS

The framework is tested using the Eclipse framework with the Xtext-based plugin installed. This plugin enables users to create a new project with the ALAS source code in it - `test.alas` file in Listing 4. The code is automatically converted to Java code and part of the resulting code is displayed in Listing 5.

One of the requirements of XJAF framework is that generated code should contain *onMessage* method - a part of the generated *onMessage* method is shown in Listing 5,

```
1. @Stateful
2. @Remote(AgentI.class)
3. @SuppressWarnings("all")
4. public class TimeSync {
5. @AgentState
6. private String startingHome;
7.
8. @AgentState
9. private String next;
10.
11. @AgentState
12. private String remaining;
13.
14. private String host = "192.168.0.1";
15.
16. private int n;
17.
18. private void syncTimers(final String hosts,
    final double time) {
19.   boolean _equals =
Objects.equal(this.startingHome, null);
20.   if (_equals) {
21.     this.startingHome = this.host;
22.   } else {
23.     boolean _equals_1 =
this.startingHome.equals(this.host);
24.     if (_equals_1) {
25.       System.out.println("I\'m back home");
26.       this.startingHome = null;
27.     }
28.   }
     //..
29. public void onMessage(final ACLMessage msg)
    {
30.   if(msg.getPerformative() ==
31.     Performative.REQUEST){
32.     String s = msg.getContent().toString();
33.     JSONParser parser = new JSONParser();
34.     JSONObject json;
35.     try {
36.       json = (JSONObject) parser.parse(s);
37.       String serviceName =
            json.get("serviceName").toString();
38.       switch (serviceName){
39.         case "syncTimers":
40.          String hosts =
            json.get("hosts").toString();
41.          double time =
Double.parseDouble(json.get("time").toString())
42.
43.            syncTimers(hosts, time);
44.     }//..
```

Listing 5. Part of TimeSync.java

lines 29 to 44. Within the *onMessage* method the data is received in the JSON format [23]. JSON or JavaScript Object Notation, is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. All the necessary JSON-related libraries must be imported and invoked. By specifying the *onMessage* method, parser automatically imports all relevant classes in the executable Java class.

VI CONCLUSION

Based on the previous work on agent domain-specific language ALAS [4][6][7], we can conclude that the Xtext is the favorable environment for the development of domain-specific languages like ALAS because not only the syntax and validation can be defined, but even the code in an arbitrary target programming language can be generated. By using the Xtext framework it is possible to write agents which will execute specific tasks, but which will also be automatically converted to the executable code of the target platform. Because of these advantages Xtext framework is a better tool than the *javacc* which was used in the previous version of ALAS [7]. This tool can be used to generate source code for other agent-oriented languages, as well as general purpose languages used for agent-oriented programming [24].

Future work for the ALAS language will include implementation of the `AlasModelGenerator` class which will be able to transform ALAS code to an arbitrary programming language. So far, we have been able to manually transform ALAS code to JavaScript and Python, so the `AlasModelGenerator` will be implemented to transform ALAS code to any of those two programming languages. The future work will include analysis of suitability of other programming languages for the XJAF framework. Finally, it is necessary to integrate the Xtext-based plugin into the XJAF framework, so the ALAS code transformation could be performed outside the Eclipse framework. That way, agents written in ALAS will be able to migrate between different XJAF servers.

VII REFERENCES

[1] Vidaković, M., Ivanović, M., Mitrović, D., Budimac, Z.: Extensible Java EE-based agent framework - past, present, future. In: Ganzha, M., Jain, L.C. (eds.) Multiagent Systems and Applications, Intelligent Systems Reference Library, vol. 45, pp. 55 - 88. Springer Berlin Heidelberg, 2013

[2] M. Ivanović, M. Vidakovi´c, D. Mitrović, and Z. Budimac. Evolution of Extensible Java EE-Based Agent Framework. In G. Jezić, M. Kusek, N.-T. Nguyen, R. Howlett, and L. Jain, editors, Agent and Multi-Agent Systems. Technologies and Applications, volume 7327 of Lecture Notes in Computer Science, pages 444–453. Springer Berlin Heidelberg, 2012.

[3] Mitrović, D., Ivanović, M., Budimac, Z., Vidaković, M., - An overview of agent mobility in heterogeneous environments, Proceedings of the workshop on Applications of Software Agents, pp. 52 – 58, 2011

[4] Mitrović, D., Ivanović, M., Budimac, Z., Vidaković, M., —Supporting heterogeneous agent mobility with ALAS, ComSIS, Vol. 9, No. 3, pp. 1203-1229, 2012

[5] Bădică, C., Budimac, Z., Z., Burkhard, Hans-Dieter, and Ivanović, M., „Software Agents: Languages, Tools, Platforms“, Computer Science and Information Systems, ComSIS 8(2), pp. 255–296, 2011

[6] Mitrović, D., Ivanović, M., Vidaković, M., „Introducing ALAS: a novel agent-oriented programming language“, In Symposium on computer languages, implementations and tools, SCLIT 2011, Greece, pp. 19–25, 2011

[7] Mitrović, D., Ivanović, M., Vidaković, M., Sredojević, D., “Okanović, D., Integracija agentskog jezika ALAS u Java agentsko okruženje XJAF“, XX naučna i biznis konferencija, 9-13. Mart 2014. pp. 457-461, 2014

[8] Shoham, Y., —“Agent-oriented programming“, Robotics Laboratory Computer Science Department, Stanford University Stanford, CA 94305, USA, 1993

[9] Anand S. Rao, "AgentSpeak(L): BDI agents speak out in a logical computable language", Springer Berlin Heidelberg, number 1038, pp 42–55, 1996

[10] M. Winikoff, "Jack™ Intelligent Agents: An Industrial Strength Platform," in Multi-Agent Programming: Languages, Tools and Applications, Springer US, pp 175-193, 2005

[11] D. Xu, G. Zheng, and X. Fan, "Information and Software technology", pp. 435-442, 1998

[12] A. E. Fallah-Seghrouchni, and A.Suna, "CLAIM and SyMPA: A Programming Enwironment for Intelligent and Mobile Agents," in Multi-Agent Programming: Languages, Tools and Applications, Springer US, pp. 95-122, 2005

[13] B. Hirsch, T. Konnerth, and A. Heßler, "Merging Agents and Services – the JIAC Agent Platform," in Multi-Agent Programming: Languages, Tools and Applications, Springer US, pp. 159-185, 2009

[14] FIPA Abstract Architecture Specification, http://www.fipa.org/specs/fipa00001/SC00001L.pdf, 12.10.2014.

[15] Mitrović, D., Ivanović, M., Vidaković, M., Budimac, Z., Extensible Java EE-based Agent Framework in Clustered Environments, 12th German Conference, MATES 2014, 23-25. Septembar, Štutgart, Nemačka, Proceedings, pp. 202-215, 2014

[16] JavaCC, https://javacc.java.net/ 15.11.201.

[17] Xtext, http://www.eclipse.org/Xtext/documentation.html, 12.10.2014.

[18] ANTLR, http://www.antlr.org, 15.10.2014.

[19] Object Managament Group, http://www.omg.org 15.11.2014.

[20] Meta Object Facility, http://www.omg.org/mof 15.11.2014.

[21] Eclipse Modeling Framewok, http://www.eclipse.org/modeling/emf 15.11.2014.

[22] Sirius 2.0, http://eclipse.org/sirius/ 20.11.2014.

[23] JSON http://docs.oracle.com/javaee/7/tutorial/doc/jsonp.htm, 18.10.2014

[24] Pokahr, A., Braubach, L., Haubeck, C., Ladiges, J., "Programming BDI agents with pure java," In: Muller, J.P., Weyrich, M., Bazzan, A.L. (eds.) Multiagent System Technologies, Lecture Notes in Computer Science, vol. 8732, pp. 216–233. Springer International Publishing, 2014