Generating Aspect-Oriented Agent Architectures

Uirá Kulesza   Alessandro Garcia   Carlos Lucena

Software Engineering Laboratory – SoC+Agents Group
Computer Science Department
Pontifícal Catholic University of Rio de Janeiro - PUC-Rio - Brazil
{uira, afgarcia, lucena}@inf.puc-rio.br

Abstract

Multi-agent systems (MASs) encompass multiple features which tend to be scattered in the software artifacts produced through the system modeling, architecture specification and implementation. This paper presents the definition of a generative approach to support uniformly the modularization of MAS features since early development stages. The proposed approach explores the MAS domain to enable the code generation of agent architectures. We have defined a domain specific language (DSL) that permits us to model orthogonal and crosscutting agent features, such as the agent knowledge, autonomy, interaction, adaptation, learning, and roles. The agent features are then expressed in an aspect-oriented architecture. The implementation of the generative approach encompasses: (i) XML technologies to specify the DSL; (ii) Java and AspectJ programming languages to implement a concrete version of our aspect-oriented agent architecture; and (iii) a code generator, implemented as an Eclipse plugin, which maps abstractions in the DSL to specific components and aspects of the agent architecture.

1. Introduction

Multi-agent systems (MASs) encompass multiple features which tend to be scattered in the software artifacts produced through the system modeling, architecture specification and implementation. With MASs growing in size and complexity, the explicit separation of agent features are still deep concerns to MAS engineers [10, 17, 27]. MASs can be seen as a specific domain that can be explored to improve our capacity to develop that kind of systems. In fact, in last years many MASs development approaches [10, 17, 27] have already explored MAS domain to create facilities to develop these systems. However, these approaches have focused on orthogonal features, such as, agent types, goals, beliefs, and plans.

Generative Programming [6] has been proposed recently as an approach based on domain engineering. It addresses the study and definition of methods and tools to enable the automatic production of software from a high-level specification. This paper presents the development of a generative approach that explores features of the MAS domain to enable the code generation of agent architectures. The proposed approach supports the modeling and implementation of orthogonal (non-crosscutting) and crosscutting agent features since preliminary development phases in a uniform way.

Our generative approach for MASs defines a domain-specific language, called Agent-DSL, in the problem space. Agent-DSL permits us to model agent features, such as, knowledge, interaction, adaptation, autonomy and roles. In the solution space, we have specified an aspect-oriented agent architecture. It is composed of interacting *aspectual components,* which are the modularity units to design crosscutting features in the architectural definition of an agent.

We have used different technologies to accomplish our generative approach for MASs. First, in the problem space, feature diagrams were initially specified to establish the relevant concepts and features encountered in the definition of an agent. Thereafter, we used this information to represent the elements of the Agent-DSL using XML-Schema [28]. Second, in the solution space, we designed the agent architecture using aspect-oriented abstractions that model each of the features encountered in Agent-DSL. After that, we have used Java and AspectJ programming languages to implement specific object-oriented components and aspectual components of the architecture.

In addition, we have defined Eclipse Modeling Framework (EMF) code templates [3] that are used to generate components and aspects to a specific agent.

Finally, in the configuration knowledge of the generative approach, a code generator has been implemented as an Eclipse plugin. It is responsible for mapping abstractions in the Agent-DSL to components and aspects of the agent architecture.

The remainder of this paper is organized as follows. Section 2 describes the process of domain analysis and design for the development of the generative approach. Section 3 presents the domain implementation to accomplish our generative approach for MASs. Section 4 discusses some lessons learned and presents our conclusions. Section 5 points to directions for future work.

2. Defining a Generative Approach for MASs

Multi-Agent Systems (MASs) are a horizontal domain that involves a number of orthogonal and crosscutting features. An orthogonal (non-crosscutting) feature is easily represented by a single modular abstraction through different development phases. Crosscutting features naturally cut across different modular abstractions in the software artifacts produced in distinct stages of the software lifecycle. Some examples of crosscutting agent features are interaction, autonomy, adaptation, learning and collaboration [11, 12, 13, 23].

The MAS domain can be explored to improve the quality and productivity on the development of these systems. In this context, the purpose of our generative approach is threefold: (i) support uniformly the crosscutting and non-crosscutting features of software agents since early development stages, (ii) abstract the common and variable features, and (iii) to enable the code generation of aspect-oriented agent architectures.

Figure 1 depicts our generative approach that is composed of:

(i) a domain-specific language (DSL), called Agent-DSL, used to collect and model both orthogonal and crosscutting features of software agents;

(ii) an aspect-oriented architecture designed to model a family of software agents. This architecture is centered on the definition of *aspectual components* to modularize the crosscutting agent features at the architectural level of abstraction;

(iii) a code generator that maps abstractions of the Agent-DSL to specific compositions of objects and aspects in the agent architecture.
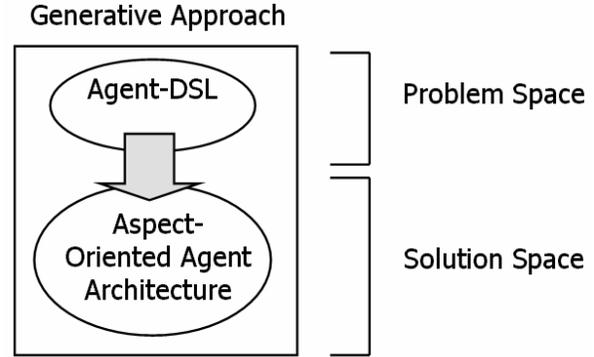

Figure 1. Generative Approach for MASs

The definition of our generative approach involved two initial phases that were concerned with the gathering of a deep understanding of the MAS domain: (i) the domain analysis, and (ii) the domain design. Following the initial phases, the implementation of the generative approach involved three additional phases: (iii) the implementation of our domain specific language, (iv) the implementation of the aspect-oriented agent architecture, and (5) the implementation of the code generator. Sections 2.1 and 2.2 present respectively the initial phases, and Section 3 presents the implementation phases.

2.1. Domain Analysis

The domain analysis was supported by our extensive work on the development of several multi-agent systems [11, 12, 13, 14], and on a survey of different modeling languages, MAS architectures and platforms [26]. To proceed with the definition of the Agent-DSL, we initially captured and analysed the different features associated with the agent definition [10, 11, 13, 17, 26, 27]. We also investigated possible relationships between these features. Feature models [20] were used to represent the features and their respective relations. This section presents a subset of the feature models produced.

Figure 2 depicts a feature model which defines the essential features associated with the *agent* concept. It is composed of its *knowledge* and its basic properties, which we termed "*agenthood*". The knowledge feature encompasses *beliefs*, *goals* and *plans*. Agent beliefs describe information about the agent itself and about the external environment with which the agent interacts. To achieve a goal, an agent executes a specific plan. During the execution of a plan, the agent manipulates its beliefs. The agenthood feature is composed of three subfeatures: *interaction*, *adaptation,* and *autonomy*. Figure 2 presents the feature diagram of the agent knowledge and agenthood.
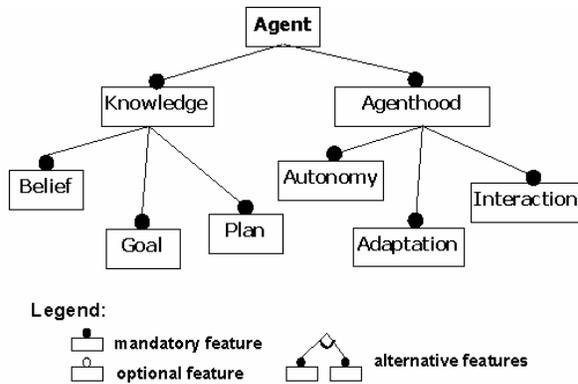
Figure 2. The Knowledge and Agenthood Features

The interaction feature (Figure 3) is the agent capacity to communicate with the environment. The agent can receive or send *messages* to the environment by means of its *sensors* and *effectors*, respectively. External messages are translated to the agent ontology using specific *parsers* in its sensors. Effector parsers translate internal messages to a specific external representation, such as FIPA ACL [9].
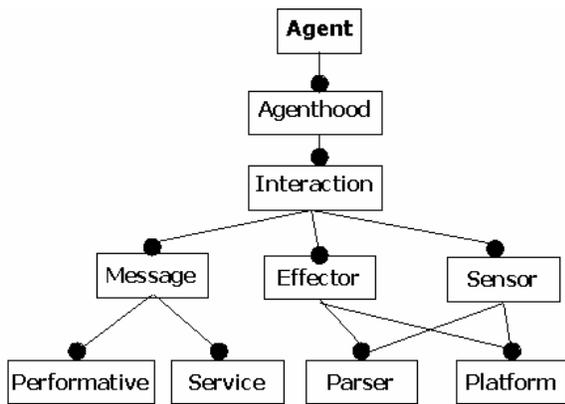

Figure 3. The Interaction Feature

The adaptation feature, depicted in Figure 4, is formed by *belief adaptation* and *plan adaptation*. Belief adaptation is responsible for interpreting received messages from the environment and manipulating its beliefs based on the message contexts. Plan adaptation determines the plan the agent must execute whenever a new goal needs to be achieved.

Figure 5 presents the main features associated with the autonomy property. The purpose of the agent autonomy is to instantiate and manage the agent goals. It deals with three types of goals: *reactive goals*, *proactive goals*, and *decision goals*. Reactive goals are instantiated when the agent receives an external request from other agents or environment components. Proactive

goals are instantiated due to internal events that occurs, such as, the end of a plan execution or the achievement of a specific agent state. Finally, the decision goals are instantiated due to external or internal events and are used to decide if special reactive or proactive goals could be instantiated. The autonomy property is also responsible for monitoring the adopted *concurrency strategy*. It supports the goal achievement by implementing a mechanism for executing concurrently agent plans.
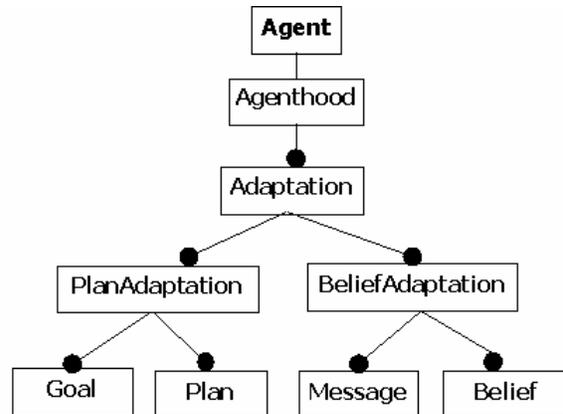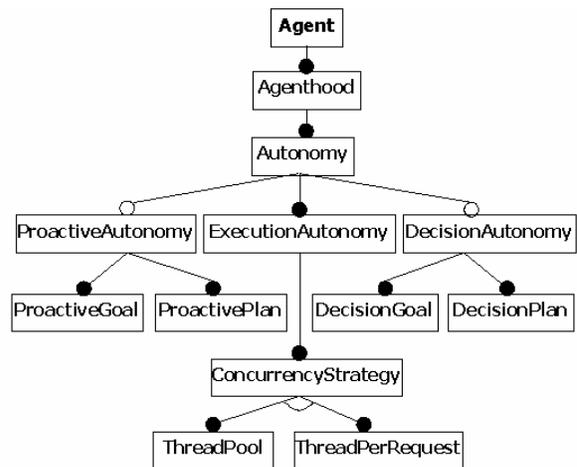

Figure 4. The Adaptation Feature

Figure 5. The Autonomy Feature

In addition to the agent knowledge and the agenthood features, an agent can incorporate additional properties. Additional features include *roles*, *mobility*, and *learning*. The initial version of our Agent-DSL provides support for the role feature. A role gives to the agent extra capacities of knowledge, interaction, adaptation and autonomy. Each agent can play different roles during its execution. A role is played by the agent

in a specific context, for instance, the need to collaborate with other agents.

Following the feature definition, we analyzed each of the agent features in order to find out which of them had orthogonal or crosscutting nature. The first step of our analysis was to capture additional relationships which

considers the orthogonal and crosscutting features of software agents (Section 2.1). The aspect-oriented agent architecture is a refinement of a previous work [11, 13]. It is composed of two kinds of components: (i) a central component that modularizes the orthogonal features associated with the agent knowledge, and (ii) the
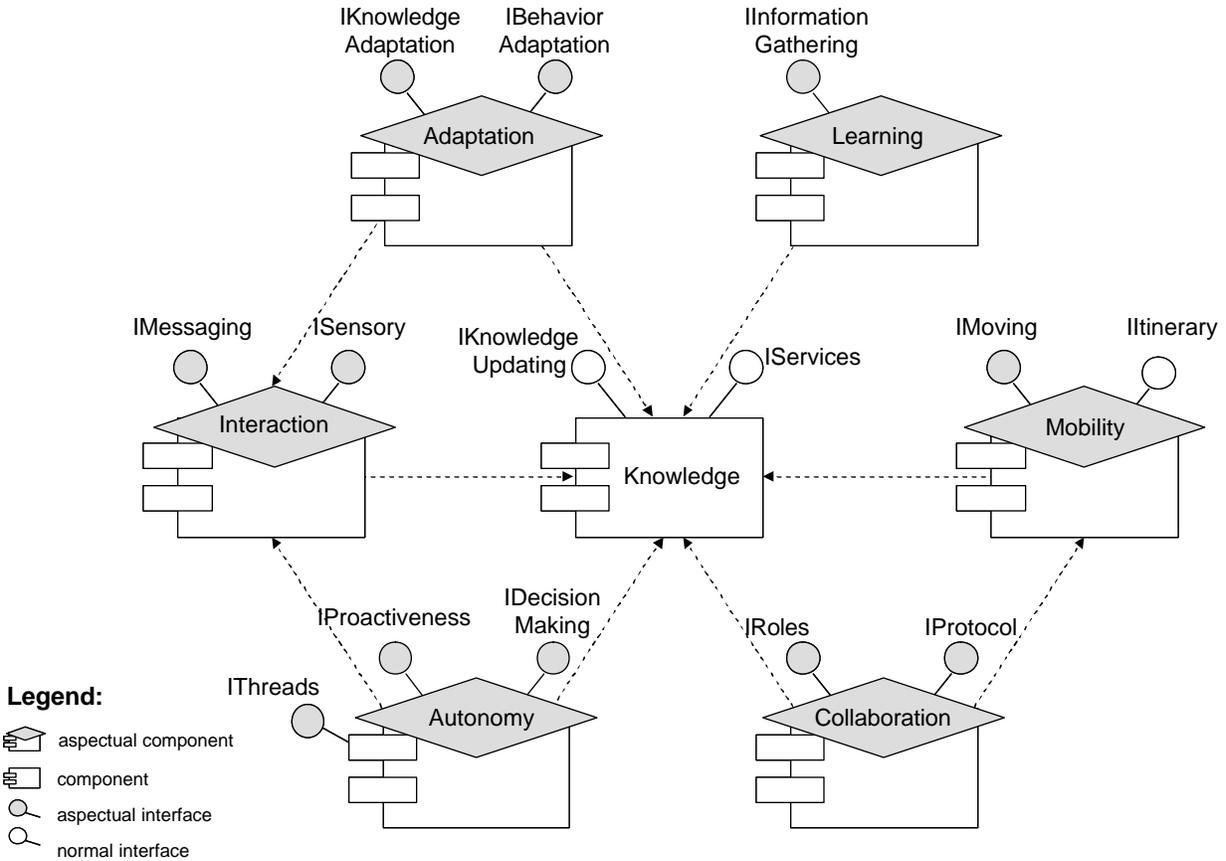

Figure 6. The Aspect-Oriented Agent Architecture

were not present in our initial feature model. The crosscutting features were identified based on the degree in which a given feature was related to other features not present in the same feature hierarchy. For example, the autonomy feature is related to the interaction feature and the knowledge feature. Orthogonal features included the knowledge elements, i.e. agent types, goals, plans, and beliefs. Crosscutting features included the agenthood features and the additional properties.

2.2. Domain Design

In the domain design, we have worked on the definition of an aspect-oriented agent architecture that

aspectual components that separate the crosscutting agent features from each other and from the knowledge component.

Figure 6 illustrates the architectural components and their relationships. We have used the aSide modeling language to represent the software architecture [5]. Each component has one or more interfaces. They have two kinds of interfaces: *normal interfaces* (colored in white) and *crosscutting interfaces* (colored in grey). The normal interfaces provide services to the other components. The crosscutting interfaces specify how an aspectual component crosscut other architectural components.

Note that each of the aspectual components is related to more than one architectural component, representing their crosscuting nature. The knowledge component contains only the orthogonal knowledge features. The knowledge component is refined as a set

of classes, representing the agent feature and its knowledge features (beliefs, goals, plans). To define specific agents, we model subclasses of the Agent class, as well as, we implement new classes to represent the belief, goals and plans of these agents.

Each of the agenthood features and additional agent properties are modeled as aspectual components due to their crosscutting nature. An aspectual component is refined as a set of aspects and auxiliary classes, which are also part of the crosscutting feature. The aspectual components crosscut elements of the Knowledge component and other aspectual components. For instance, the Interaction component introduces into the Knowledge component some operations to receive or send messages (IMessaging Interface), and pointcuts to sense events from environment objects (ISensory Interface).

The Autonomy component crosscuts the Interaction component because it needs to create goals according to messages received from the external environment. It also crosscuts the Knowledge component since goals may have to be created whenever some pieces of the agent knowledge change. The crosscutting interface IGoalCreation specifies how the Autonomy component crosscut these components in order to instantiate the agent goals. The crosscutting interface IThread describes how the execution autonomy crosscuts the Knowledge component to implement the concurrency agent strategy.

The Adaptation component crosscuts the Interaction component and the Knowledge component. It is connected with the former since adaptation of beliefs (Section 2.1) may be required upon the reception of external messages. The connection with the later is because adaptation of plans is necessary whenever a new agent goal needs to be achieved. The IKnowledgeAdaptation and IBehaviorAdaptation interfaces specify respectively how the Adaptation component affects the other architectural components.

Figure 6 also presents how the additional components (Mobility, Collaboration and Learning) crosscut each other and the agenthood components. It is worth to highlight that Figure 6 only presents the mandatory relationships between the architectural components, i.e. the relationships which are always present in agent architectures independently from specific applications. However, as the agent complexity increases, a specific component can crosscut other agent components. For instance, the Collaboration component can crosscut also the Interaction component when more sophisticated coordination strategies are required in a MAS. The Learning component also usually crosscuts the Collaboration and Interaction components.

3. Implementing the Generative Approach

We have implemented a first version of the generative approach for MASs using the following technologies: (i) XML-Schema [28] was used to specify the semantic of the Agent-DSL based on the feature models (Section 2.1); (ii) AspectJ and Java programming languages were used to implement components and aspects of the generic agent architecture; we have also defined EMF/JET code templates [3] which were useful to generate components and aspects to a specific agent; (iii) finally, a code generator supports the architecture configuration and has been implemented as an Eclipse plugin - it is responsible for mapping abstractions in the Agent-DSL into components and aspects of the agent architecture.

The following subsections describe in more detail the use of the technologies mentioned above to implement the generative approach.

3.1. Agent-DSL

The Agent-DSL was implemented using XML Schema [28] technology. The feature models were translated to XML Schema complex types. Many Agent-DSL complex types are composed of other complex types of the DSL. Below we present partially the XML Schema of the Agent-DSL.

```
...
<!-- Role Type -->
<xs:complexType name="RoleType">
  <xs:sequence>
    <xs:element name="name" type="name" minOccurs="1"
      maxOccurs="1"/>
    <xs:element name="description" type="description"
      minOccurs="0" maxOccurs="1"/>
  <xs:element name="belief" type="BeliefType"
      minOccurs="1" maxOccurs="unbounded"/>
  <xs:element name="goal" type="GoalType"
      minOccurs="1" maxOccurs="unbounded"/>
  <xs:element name="plan" type="PlanType"
      minOccurs="1" maxOccurs="unbounded"/>
  <xs:element name="interaction"
      type="InteractionType" minOccurs="1"
      maxOccurs="1"/>
  <xs:element name="adaptation"
      type="AdaptationType" minOccurs="1"
      maxOccurs="1"/>
  <xs:element name="autonomy" type="AutonomyType"
      minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>

<!-- Agent Type -->
<xs:complexType name="AgentType">
  <xs:sequence>
    <xs:element name="name" type="name" minOccurs="1"
      maxOccurs="1"/>
    <xs:element name="name" type="name" minOccurs="1"
      maxOccurs="1"/>
    <xs:element name="description" type="description"
      minOccurs="0" maxOccurs="1"/>
```

```
        <xs:element name="belief" type="BeliefType"
            minOccurs="1" maxOccurs="unbounded"/>
        <xs:element name="goal" type="GoalType"
            minOccurs="1" maxOccurs="unbounded"/>
        <xs:element name="plan" type="PlanType"
            minOccurs="1" maxOccurs="unbounded"/>
        <xs:element name="interaction"
            type="InteractionType" minOccurs="1"
            maxOccurs="1"/>
        <xs:element name="adaptation"
            type="AdaptationType" minOccurs="1"
            maxOccurs="1"/>
        <xs:element name="autonomy" type="AutonomyType"
            minOccurs="1" maxOccurs="1"/>
        <xs:element name="role" type="RoleType"
            minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
…
```

For each MAS to be generated, an agent description XML document of the Agent-DSL must be offered. This document must conform to the XML Schema that defines the Agent-DSL.

3.2. Aspect-Oriented Agent Architecture

The aspect-oriented agent architecture was implemented using the Java [15] and AspectJ [22] programming languages. It is composed of: (i) a framework that contains class and aspect hot-spots [7]; (ii) a set of components implemented to support basic and specific functionalities of the agent; and (iii) a set of meta-components, implemented as EMF/JET [3] source templates, that are customized by the code generator using information of the Agent-DSL.

The framework defines some classes and aspects as the core of the aspect-oriented agent architecture. It also contains hot-spot classes that we use to define the agent knowledge, and hot-spot aspects used to concretize architectural components implementing the agenthood and additional agent features.

Many framework components were created to implement specific functionalities associated with the agenthood features, such as:

• *interaction features*: data structures to store received and sent messages; data structures to maintain sensors and effectors of the agent; and concrete sensors and effectors to specific agent platforms (such as JADE [2]);

• *adaptation features*: data structures to interrelate goals and plans; and data structures to call class methods to update beliefs when specific messages are received by the agent;

• *autonomy features*: data structures to instantiate goals from external messages (reactive goals) and from internal events (proactive goals); the basic concurrency strategy behavior; and concrete concurrency strategy, such as, thread pool and thread per request.

Finally, the agent architecture also contains some EMF/JET source templates. We use source templates to implement components that need to be customized based on information collected by the Agent-DSL. Java Emitter Templates (JET) a generic template engine of the Eclipse Modeling Framework (EMF) [3] has been used. JET enabled us to write source templates that express structure and behavior of classes and aspects that we want to generate using the agent description file. Examples of classes and aspects that we wrote as templates are: (i) concrete instances of hot-spots (classes or aspects), such as, specific Agent type classes, specific agenthood aspects; and (ii) specific agent plans and goal classes.

3.3. Agent Architecture Generator

The current version of the agent architecture generator was implemented as an Eclipse plugin [25]. We used a kit of technologies related to Eclipse project to implement the generator. JAXB plugin [18] was used to enable the reading of the agent description XML file of the Agent-DSL. This plugin permits to generate classes that represent the elements of a XML-Schema. So, you can use these generated classes to read XML files that conform to that XML-Schema.
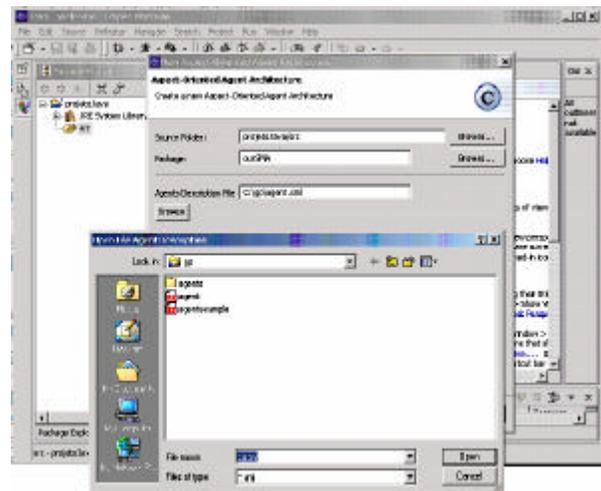

Figure 7. Agent AO-Architecture Eclipse Plug-in

The generator plugin offers a wizard in the Eclipse workbench to start the process of code generation (figure **7**). The wizard requests from the user: (i) a source folder to arrange the classes and aspects generated; (ii) a package name used as a root package to arrange classes and aspects generated; and (iii) the agent description XML file that contains descriptions of

agents to be generated. As soon as the user provides this information classes and aspects of agents are generated.

The internal implementation of the agent architecture generator plugin uses Java Development Tooling (JDT) API [25] to create Java packages and classes, and AspectJ aspects. Also, JET API is used to process templates and to pass information of agent description file to source templates.

4. Discussions and Conclusions

This paper presented a generative approach to assist the development of multi-agent systems. The aim of the approach is to explore the horizontal domain that MASs represent to enable the code generation of agent architectures. Aspectual components have been used to model crosscutting agent features from the architectural point of view. We have already used the generative approach to implement partially two case studies: (i) ExpertCommittee - a conference management system; and (ii) Portalware - a web-based environment for the construction and management of e-commerce portals [13, 14].

The definition of a generative approach, brings important benefits compared to other MAS development approaches [10, 17, 27]. The benefits include: (i) clear definition of the mapping between high-level features and implementation components (classes, aspects) of a agent architecture in the code generator; (ii) clear separation of orthogonal and crosscutting agent features; (iii) flexibility to evolve the problem space independent of solution space by creating new domain specific languages to address other MAS concerns; and (iv) flexibility to evolve or change the architectural model used in the solution space independent of problem space.

In our generative approach we have derived aspects from our domain knowledge based on previous studies [10, 11, 13, 17, 26, 27]. We identified that many important MAS features, such as, autonomy and adaptation, are commonly tangled in the specification artifacts and source code of agent implementations. Results gathered in previous empirical studies [11, 12, 13, 14] have emphasized that the separation of these MAS features improves the maintainability and reusability of MASs.

The aspect-oriented framework and code templates (Section 3.2) expose the identification and generalization of many MASs domain aspects (agenthood and additional agent features).

5. Future Work

This position paper presented a initial version of a generative approach for MASs. We are currently working in different ways to improve our ability to generate MASs source code and to handle its different artifacts (DSLs, components and aspects in the architecture, mapping between DSL elements and components/aspects).

The specification of an agent using Agent-DSL is supported currently by the edition of a XML file. We are defining a mixture of wizards and diagrams that help MAS developers to specify agents based on Agent-DSL. We have also focused on Agent-DSL refinement to support the modeling of other relevant MASs concerns, such as, learning, mobility, agent coordination, and agent organizations. We are working on a conceptual framework [26] that provides a more precise definition of these agent features in order to incorporate them into our Agent-DSL. Moreover we are investigating how to increment Agent-DSL to model the dynamic semantics of MASs, such as, plan execution, role creation and allocation.

Also, an aspect-oriented pattern language is being developed to refine our agent architecture as a set of design patterns and idioms that model progressively MASs concerns. The patterns and idioms provide detailed solutions for each architectural component in terms of classes and aspects.

The use of AspectJ in our case studies limited the capacity to reconfigure dynamically the MASs, not permitting attach and detach aspects from the agent and knowledge objects. We plan to implement a new case study based on dynamic weaving technologies, such as, PROSE [24], AspectWerkz [1] and JBoss [8].

Finally, an ongoing research work is a definition of an architectural definition language (ADL) that supports the definition of the aspectual components, normal and crosscutting interfaces (Section 2.2). We plan to use this ADL to formalize the definition of aspect-oriented agent architectures. The ADL can bring many benefits for our work, such as: (i) enable the specification of aspect-oriented architectures in a high-level independent of technologies; and (ii) facilitate architecture analysis and maintenance. When incorporating the ADL to our generative approach, we plan to offer flexible ways of specifying the transformations of elements in Agent-DSL to elements (components, aspect and interfaces) of the ADL. Also, transformation rules of the ADL to concrete technologies (for instance, Java and AspectJ) could be defined.

6. References

[1] AspectWerkz. Available at URL http://aspectwerkz.codehaus.org/

[2] F. Bellifemine, A. Poggi & G. Rimassi, "JADE: A FIPA-Compliant agent framework", Proc. Practical Applications of Intelligent Agents and Multi-Agents, pp. 97-108, April 1999.

[3] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. Grose. Eclipse Modeling Framework, Addison-Wesley, 2003.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-Oriented Software Architecture: A System of Patterns, John Wiley and Sons, 1996.

[5] C. Chavez. "A Model-Driven Approach to Aspect-Oriented Design". PhD Thesis, PUC-Rio, 2004.

[6] K. Czarnecki, U. Eisenecker. Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.

[7] M. Fayad, D. Schmidt, R. Johnson. Building application frameworks: Object-oriented foundations of framework design, John Wiley & Sons, September 1999.

[8] M. Fleury, F. Reverbel. "The JBoss Extensible Server". Proceedings of the International Middleware Conference 2003, vol. 2672 of LNCS, pp. 344-373, Springer-Verlag, 2003.

[9] Foundation of Intelligent Physical Agent: FIPA Interaction Protocols Specification, (2002). Available at URL http://www.fipa.org/repository/ips.html

[10] A. Garcia, C. Lucena. Software Engineering for Large-Scale Multi-Agent Systems. ACM Software Engineering Notes, August 2002.

[11] A. Garcia, et al. "Engineering Multi-Agent Systems with Aspects and Patterns". Journal of the Brazilian Computer Society, September 2002.

[12] A. Garcia et al. Separation of Concerns in Multi-Agent Systems: An Empirical Study. In: C. Lucena et al (Eds). "Advances in Software Engineering for Multi-Agent Systems". Springer-Verlag, LNCS 2940.

[13] A. Garcia, C. Lucena, D. Cowan. "Agents in Object-Oriented Software Engineering". Software: Practice and Experience, May 2004, pp. 1-33. (to appear)

[14] A. Garcia, M. Cortés, C. Lucena. "A Web Environment for the Development and Maintenance of E-Commerce Portals based on a Groupware Approach". Proceedings of the Information Resources Management Association International Conference (IRMA'01), Toronto, May 2001,

[15] J. Gosling, B. Joy, G. Steele. The Java Language Specification. Addison-Wesley Longman, Inc., 1996.

[16] E. Harold, W. Means. XML in a Nutshell, 2nd Edition, O´Reilly, 2002.

[17] C. Iglesias, et al. "A Survey of Agent-Oriented Methodologies". Proceedings of the ATAL-98, Paris, France, July 1998, pp. 317-330.

[18] JAXB Eclipse Plugin. Available at URL http://sourceforge.net/projects/jaxb-builder/

[19] N. Jennings, M. Wooldridge. "Agent-Oriented Software Engineering". In: J. Bradshaw (ed.), Handbook of Agent Technology, AAAI/MIT Press, 2000.

[20] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson. "Feature-oriented domain analysis (FODA) feasibility study". Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[21] G. Kiczales, et al. "Aspect-Oriented Programming". Proc. Of ECOOP'97, LNCS 1241, Springer-Verlag, Finland, June 1997.

[22] G. Kiczales, et al. "Getting Started with AspectJ". Communication of the ACM. October 2001.

[23] Pace, A., Trilnik, F., Campo, M. "Assisting the Development of Aspect-based MAS using the SmartWeaver Approach". In: A. Garcia, C. Lucena, J. Castro, A. Omicini, F. Zambonelli (Eds). "Software Engineering for Large-Scale Multi-Agent Systems". Springer-Verlag, LNCS, March 2003.

[24] A. Popovici, T. Gross, G. Alonso. Dynamic Weaving for Aspect Oriented Programming. In 1st Intl. Conf. on Aspect-Oriented Software Development, Enschede, The Netherlands, Apr. 2002.

[25] S. Shavor, J. D'Anjou, S. Fairbrother, et all. The Java Developer's Guide to Eclipse. Addison-Wesley, 2003.

[26] V. Silva, et al. "Taming Agents and Objects in Software Engineering". In: "Software Engineering for Large-Scale MASs". Springer, LNCS 2603, March 2003.

[27] M. Wooldridge, P. Ciancarini (Eds.). "Agent-Oriented Software Engineering: The State of the Art". In: Agent-Oriented Software Engineering, Springer, LNAI, 2001.

[28] XML Schema. Available at URL http://www.w3.org/XML/Schema.