A Generative Approach for
Multi-Agent System Development

Uirá Kulesza[1], Alessandro Garcia[1], Carlos Lucena[1], Paulo Alencar[2]

[1] PUC-Rio, Computer Science Department, LES, SoC+Agents Group,
Rua Marques de São Vicente, 225 - 22453-900, Rio de Janeiro, RJ, Brazil
{uira, afgarcia, lucena}@inf.puc-rio.br

[2] University of Waterloo, Computer Science Department, Computer System Group
Waterloo, Ontario, N2L 3G1 Canada
palencar@csg.uwaterloo.ca

Abstract. The development of Multi-Agent Systems (MASs) involves special concerns, such as interaction, adaptation, autonomy, among others. Many of these concerns are overlapping, crosscut each other and the agent's basic functionality. Over the last few years, several methodologies and implementation frameworks have been proposed to support agent-oriented software engineering. Although these approaches have brought some benefits to improve the productivity and quality on the MAS development, they present some restrictions. First, agent-oriented methodologies are too high level and do not indicate how to master the complexity of MAS concerns based on the object-oriented abstractions. Second, implementation frameworks provide object-oriented APIs for MAS development without providing guidelines for the modularization of agent concerns. Moreover, neither of the proposed agent oriented-approaches deals with the modeling and implementation of agent crosscutting concerns. This paper presents a generative approach for the development of MASs that addresses these restrictions. The proposed approach explores the MAS domain to enable the code generation of heterogeneous agent architectures. Aspect-oriented techniques are used to allow the modeling of crosscutting agent features. The generative approach brings several benefits to the code generation and modeling of agent crosscutting features since early development stages.

1 Introduction

Multi-Agent Systems (MASs) are composed of software entities that involve special properties (concerns), such as interaction, adaptation, autonomy, among others. With the growth of the Internet and the advances in networking technologies, the design and development of MASs have risen in importance. However, the effort and cost of designing and implementing MASs while satisfying quality requirements, such as maintainability and reusability, are still deep challenges to software engineers. [10, 14]. None is more serious than the difficulty to deal with the modularization and composition of multiple agent properties since early development stages [9, 13]. In general, a MAS has multiple software agents with different properties to be composed in different ways [9, 13]. Moreover these properties crosscut each other and the basic

agent functionality, making their modeling, modularization, and composition more difficult.

Over the last few years, several methodologies and implementation frameworks for agent-oriented software engineering have been proposed. Agent-oriented methodologies [15, 27] propose modeling languages and techniques that allow MAS developers to specify their systems using agent-oriented abstractions. Implementation frameworks [1, 18] improve the productivity of MAS development by providing customized object-oriented APIs. Despite the advantages that these agent-oriented software engineering approaches bring, each of them offers restrictions, such as: (i) most methods and modeling languages proposed are too high level and do not indicate how to master the complexity of these concerns based on the object-oriented abstractions; (ii) on the other hand, implementation frameworks do not provide guidelines for the modularization of agent concerns; (iii) agent-oriented methodologies and implementation frameworks also impose the use of MAS abstractions in the modeling and implementation of these systems considering a particular point of view; and (iv) moreover, neither of the proposed agent oriented-approaches deal with the modeling and implementation of agent crosscutting concerns typically encountered in MASs [9, 11, 12, 13, 14]. As a consequence, these restrictions decelerate the development process and affect negatively the reuse and maintenance of the MAS artifacts.

In this context, we have explored the integrated use of two software engineering approaches for dealing with the mentioned restrictions: generative programming and aspect-oriented software development. We believe that the combination of these software engineering approaches can help to define a systematic and flexible MAS approach in order to overcome many of the restrictions presented by current MAS approaches.

Generative Programming (GP) [7] has been proposed recently as an approach based on domain engineering [17, 23]. It addresses the study and definition of methods and tools to enable the automatic production of software families from high-level specifications. GP promotes the separation of problem and solution spaces, giving flexibility to evolve both independently. Problem space models the existing concepts and features in a specific domain. Solution space consists of the components that are used to build particular software systems. Code generators represent the configuration knowledge in a generative model. They define how specific feature combinations in the problem space are mapped to a set of software components in the solution space.

The use of GP in the definition of a MAS approach offers the potential to explore the MAS domain systematically. It also allows the problem and solution spaces to evolve independently, including issues related to the technologies used. Therefore, it offers more flexibility to define and evolve the MAS high-level abstractions and architectures used in the production of particular system families for this domain. Moreover, GP advocates the clear definition of the mapping between high-level features and implementation components by means of code generators. In this way, GP addresses the lack of guidelines provided by agent-oriented methodologies during the translation of high-level agent features to specific combinations of implementation components. GP can also help to reduce the cost and effort of MAS development by means of the code generation of agent architectures.

Aspect-oriented software development (AOSD) [19, 26] is an evolving approach to modularize crosscutting concerns that existing paradigms (e.g.: object-oriented) are not able to capture explicitly. Crosscutting concerns are concerns that often crosscut several modules in a software system. AOSD encourages modular descriptions of complex software by providing support for cleanly separating the basic system functionality from its crosscutting concerns. Aspect is the abstraction used to modularize the crosscutting concerns.

The use of aspect-oriented (AO) techniques makes it possible the modeling of several MAS concerns that are often scattered in the design and code of multi-agent systems and implementation frameworks [9, 11, 12, 13, 14]. Thus, AO abstractions are used in our approach to capture several crosscutting concerns encountered in the implementation of agent architectures. Most of these crosscutting concerns are not captured by the existing agent-oriented methodologies. Examples of such concerns are interaction, autonomy, adaptation and collaborative roles. The use of AO techniques enhances the maintainability and reusability of MASs, because the design and code of crosscutting agent concerns can be modularized and are not intermingled with code of non-crosscutting agent concerns.

This paper presents an aspect-oriented generative approach for the development of MASs. Our generative approach explores specific features of the MASs domain to enable the code generation of agent architectures. It allows software engineers in dealing with several agent concerns in MASs from early development phases until the system implementation. The generative approach for MASs defines a domain-specific language, called Agent-DSL, for supporting the uniform modeling of several orthogonal (non-crosscutting) and crosscutting agent concerns. The approach also includes a generic and flexible aspect-oriented agent architecture [9, 13] to enable the generation of different kinds of agents. Aspect-oriented abstractions are used to design crosscutting concerns encountered in the architectural definition of an agent. Finally, the generative approach defines a code generator that maps abstractions of the Agent-DSL to specific compositions of objects and aspects in the agent architecture.

The remainder of this paper is organized as follows. Section 2 presents our generative approach for MASs, detailing the domain analysis and design phases of its development. Section 3 describes the implementation of the elements of the generative approach. Section 4 shows the application of the generative approach to a case study. Section 5 presents some related work. In Section 6, we offer our conclusions and indicate the direction of future work.

2   A Generative Approach for Multi-Agent Systems

The aspect-oriented (AO) generative approach explores the domain of multi-agent systems (MASs) to improve their quality and productivity. The purpose of the generative approach is threefold: (i) to uniformly support crosscutting and orthogonal (non-crosscutting) features of software agents starting at early development stages; (ii) to abstract the common and variable features; and (iii) to enable the code generation of AO 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 orthogonal and crosscutting features of software agents;

(ii) an *AO architecture* [9, 13] that models a family of software agents. It is centered on the definition of aspectual components to modularize the crosscutting agent features;

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

The development of the generative approach underwent a typical domain engineering process [7, 23], organized in three phases: domain analysis (Section 2.1), domain design (Section 2.2), and domain implementation (Section 3).

In the domain analysis, common agent concerns encountered in MASs were modeled using feature models [17]. This study was supported by our extensive work on the development of several multi-agent systems [9-14] and on a survey of different modeling languages, MAS architectures and platforms [25]. Section 2.1 details the agent features modeled during domain analysis.

Domain design consisted of the specification of a generic and flexible AO agent architecture [9, 13]. Each feature modeled in domain analysis was considered. Section 2.2 presents the specification of the AO agent architecture that makes it possible to represent crosscutting agent concerns at the architectural level.

In the domain implementation, each of the elements of the generative approach was accomplished. Agent-DSL was implemented using Eclipse Modeling Framework (EMF) models [3]. The AO agent architecture was implemented as an AO framework using AspectJ and Java programming languages. Finally, the code generator of the generative approach was accomplished as an Eclipse plug-in. Section 3 details the implementation of these elements.
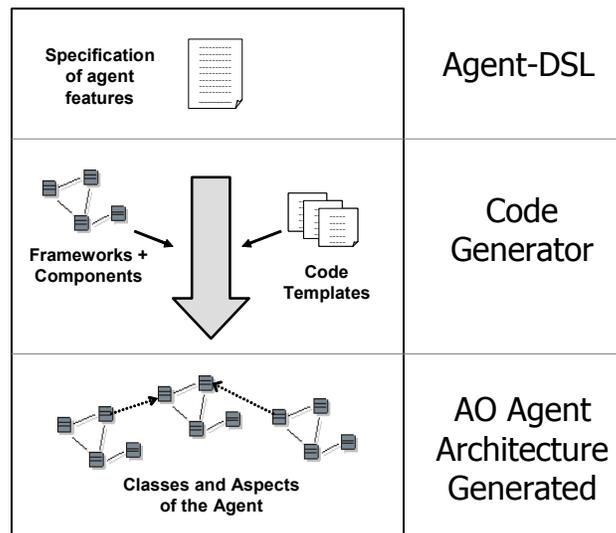

Figure 1. The Generative Approach for MASs

2.1. Domain Analysis

During the domain analysis, recurring agent concerns of MASs were modeled using feature models [17]. Feature models are used to represent common and variable features of system families. We captured the different features associated with the agent concept, including non-crosscutting and crosscutting agent features. Figure 2 depicts a partial feature model produced during this phase.

A new kind of relation between features, called *crosscuts* relation, was introduced in feature models in order to support the representation of crosscutting features. We say that a feature A crosscuts a feature B, when either A or one of its subfeatures depends and inspects B or one of the subfeatures of B. The term "inspect" means the act of observing a feature. The following agent features were characterized as being crosscutting: interaction, adaptation, autonomy, and collaboration. Each of them inspects elements of the knowledge feature in order to exhibit a specific agent property. For example, the autonomy feature inspects changes on the knowledge feature in order to detect the need for autonomous proactive behavior. Figure 2 presents the crosscutting relationships between features and respective inspected features.
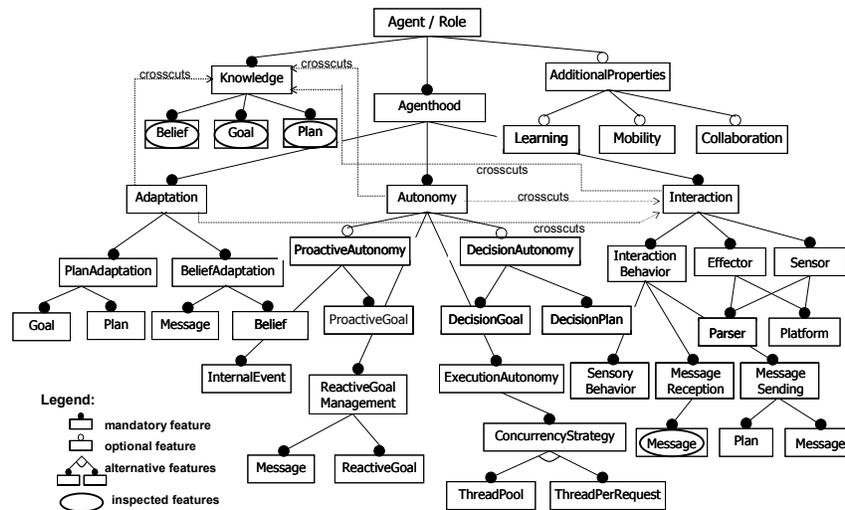

Figure 2. Partial Feature Model of the Agent

Tables 1, 2 and 3 present the definition of the agent features presented in Figure 2. Table 1 emphasizes the knowledge features, Table 2 describes the concerns common to every kind of software agent (agenthood features), and Table 3 defines the optional agent features (additional properties). The tables also indicate which agent concerns are crosscutting and detail the respective inspected features. Learning and mobility are not mandatory features in agent architectures and they have not been explored in this work.

Agent Knowledge		
Feature	Description	Crosscutting?
Belief	• Describes information about the agent itself and about the external environment.	No
Goal	• Represents a specific agent aim to be achieved.	No
Plan	• Set of actions to achieve an agent goal. • The plan execution involves the manipulation of beliefs.	No

Table 1. Agent Knowledge Features

Agenthood (Basic Properties)		
Feature	Description	Crosscutting?
Interaction	• Defines the agent ability to communicate with the environment. • The agent receives and sends messages to the environment by means of its sensors and effectors. • External messages are translated to the agent ontology using specific parsers. • Parsers translate internal messages to a specific external representation.	Yes • It crosscuts, for example, the Knowledge feature to inspect agent plans that need to send external messages.
Adaptation	• Defines changes in the agent knowledge or behavior. • It encompasses belief adaptation and plan adaptation. • *Belief adaptation* is responsible for interpreting received messages from the environment and for 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.	Yes • It crosscuts the Knowledge feature to inspect belief updates. • It also crosscuts the Interaction feature to inspect messages received.
Autonomy	• Instantiates and manages 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 occur, such as the end of a plan execution or the achievement of a specific agent state. • *Decision goals* are instantiated due to external or internal events and are used to decide whether special reactive or proactive goals could be instantiated.	Yes • It crosscuts the Knowledge feature to inspect belief updates. • It also crosscuts the Interaction feature to inspect received external messages.

Table 2. Agenthood Features

Agent Additional Properties		
Feature	Description	Crosscutting?
Collaboration	• The agent ability to cooperate with other agents through the performance of roles. • A role introduces to the agent extra features of knowledge, interaction, adaptation and autonomy. • Each agent can play different roles.	Yes

Table 3. Agent Additional Properties Feature

2.2. Domain Design

Domain design consisted of specifying a generic and flexible agent architecture for the domain at hand. Our domain design considered all the features modeled during domain analysis. The AO agent architecture is a refinement of a previous work [11, 14]. It uses two kinds of components: (i) the Knowledge component that modularizes the orthogonal features associated with the agent knowledge; and (ii) the *aspectual components* that separate the crosscutting agent features from each other and from the Knowledge component. Aspectual components represent crosscutting features at the architectural level.

Figure 3 depicts the components of the AO agent architecture. We have used a new notation to graphically represent an AO architecture. It is an extension of the ASideML modeling language [4]. We developed this notation to enable the representation of aspectual components. An aspectual component may crosscut other aspectual or non-aspectual components using its crosscutting interfaces. A crosscutting interface specifies when and how the aspect affects one or more architectural components. A crosscutting interface may both add new state or behavior in other components and intercept (and modify) the behavior of components. Non-aspectual (normal) components are represented in a similar way to UML [2] and offer their services through the normal interfaces.

The Knowledge component models the orthogonal features (belief, goal, plan) related to the knowledge feature. It realizes two normal interfaces: (i) IKnowledgeUpdating – to update the agent knowledge; and (ii) IServices – to offer agent services. The implementation (section 3.2) of this component is refined as a set of classes.

Each aspectual component was refined during the domain implementation (Section 3.2) as a set of aspects and auxiliary classes, which are also part of the crosscutting feature. The Interaction aspectual component models the interaction crosscutting feature. It is composed of two crosscutting interfaces: (i) IMessageReception – which introduces the capacity to receive external messages into the Knowledge component; and (ii) IMessageSending – which crosscuts elements of the Knowledge component to define specific points where it is necessary to send messages to the environment. It also crosscuts elements of the Collaboration aspectual component to specify specific points in collaboration plans where it also is necessary to send messages to the environment.

The Adaptation aspectual component models the adaptation crosscutting feature. It is composed of two crosscutting interfaces: (i) IBeliefAdaptation – which intercepts services of the IMessageReception interface on the Interaction component in order to update agent beliefs when new external messages are received; and (ii) IPlanAdaptation – which intercepts services of the IKnowledgeUpdating interface on the Knowledge component in order to instantiate new plans when the agent needs to achieve a specific goal.

Finally, the Collaboration aspectual component models the role crosscutting feature. It is composed of two crosscutting interfaces: (i) IExtrinsicKnowledge – which introduces new knowledge associated with the roles in the Knowledge component; and (ii) IRoleBinding – which defines specific points in the Knowledge component where agent roles are instantiated and bound to the agents.
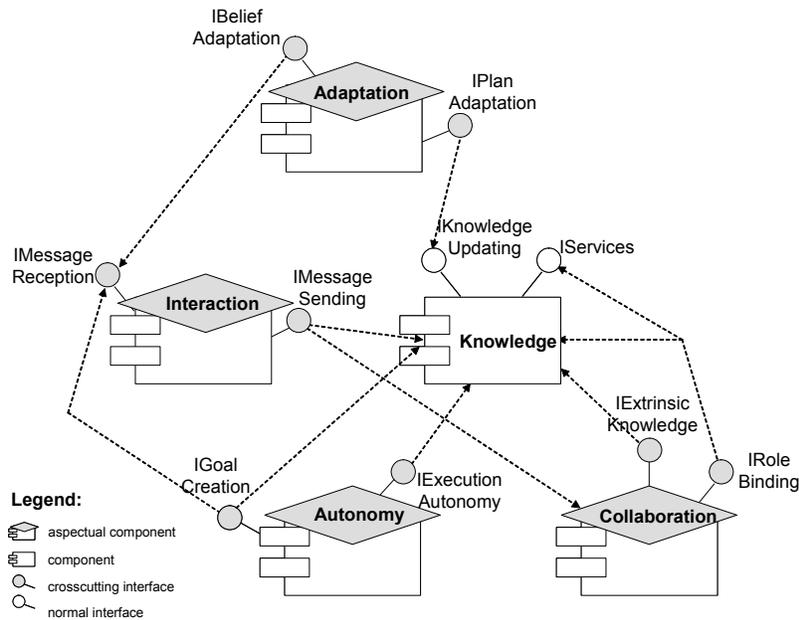
Figure 3. The Aspect-Oriented Agent Architecture

3. Implementing the MAS Generative Approach

In this section, we describe the implementation of the elements of our generative approach: (i) the Agent-DSL; (ii) the aspect-oriented agent architecture; and (iii) the code generator.

3.1. Agent-DSL

Based on the feature models defined in the domain analysis (Section 3.1), we defined a domain-specific language, called Agent-DSL. This language is used to specify the agency properties that an agent could have to accomplish its tasks. It makes it possible to model agent features, such as knowledge, interaction, adaptation, autonomy and collaboration.

The Eclipse Modeling Framework (EMF) [3] was used to specify the Agent-DSL. EMF is a Java/XML framework for generating tools and other applications based on simple class models. By using this framework, it was necessary to define a model that expresses the semantics of the Agent-DSL - in other words, the meta-model of this DSL. EMF allows the specifying of a meta-model by using XML Schema, annotated Java or UML modeling tools (e.g.: Rational Rose). After that, EMF uses this meta-model specification to automatically generate Java code and

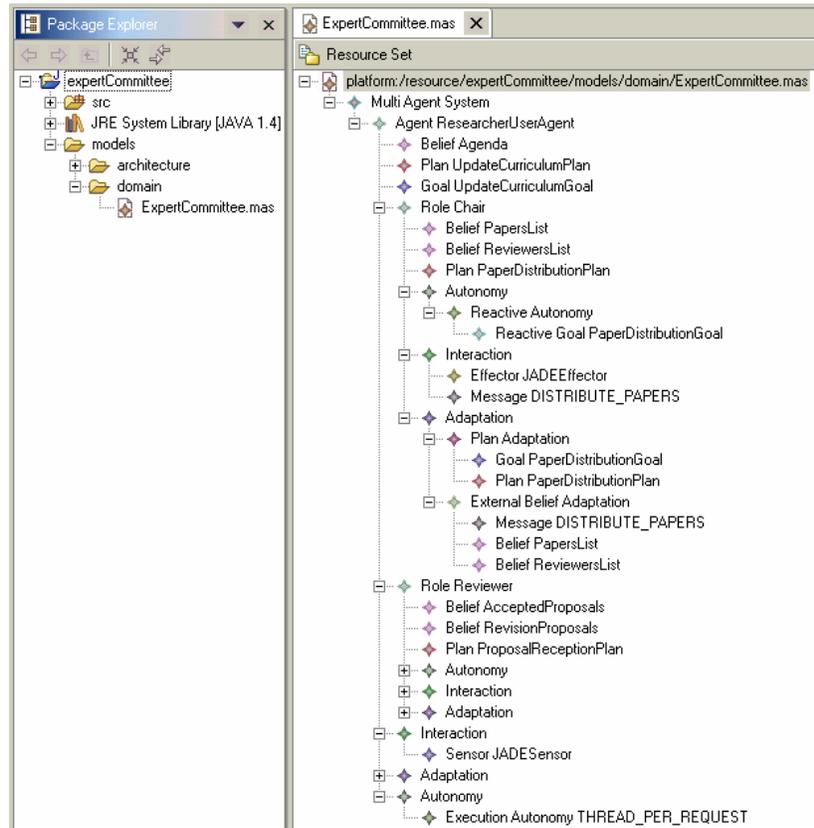Eclipse editors, which allows to create and edit models that conform to this meta-model.

Figure 4. An Agent Specification using Agent-DSL

In order to use EMF to specify the Agent-DSL, we first translated the feature models to a UML class diagram. In this translation, we basically converted: (i) features to classes; (ii) mandatory feature relations to UML composition relations; and (iii) optional feature relations to UML aggregation relations. Also, specific characteristics of each feature were introduced as class attributes. Then, this class diagram served as input for EMF generating: (i) Eclipse visual editors that permit the creation and manipulation of Agent-DSL models; and (ii) Java classes to manipulate the instances of Agent-DSL models. The latter are used by the code generator of the generative approach (see details in Section 3.3), during the customization of agent architectures.

Figure 4 presents the EMF visual editor that is used to create instances of Agent-DSL models. The figure illustrates the specification of an agent used in a case study. Section 4 presents additional details about this case study.

3.2. The Aspect-Oriented Agent Framework

The implementation of the generic AO agent architecture (Section 2.2) was realized using Java and AspectJ [20] programming languages. The basis of the architecture implementation is an AO framework that contains hot-spots as classes and aspects [8]. Figure 5 presents a partial description of the AO framework. Every class and aspect presented in the figure is a hot-spot. The ASideML modeling language [4] is used to represent visually the framework. This language extends UML with notations for representing aspects.

The notations provide a detailed description of the aspect elements. In this modeling language, an aspect is represented by a diamond; it is composed of internal structure and crosscutting interfaces. The internal structure declares the internal attributes and methods. At the detailed design level, a crosscutting interface specifies when and how the aspect affects one or more classes [4]. Each crosscutting interface is composed of inter-type declarations, pointcuts and advices. The first part of a crosscutting interface represents inter-type declarations, and the second part represents pointcuts and their attached advices. The notation uses a dashed arrow to represent the *crosscut* relationship, which relates one aspect to classes and/or aspects.

The Knowledge component (Section 2.2) was refined as a set of classes – `Agent`, `Belief`, `Goal` and `Plan` classes. Each of them represents a specific hot-spot that can be extended to define an agent type. Agent beliefs are defined in our architecture as domain classes that `Agent` instances can aggregate. Each one of the aspectual components (Section 2.2) was refined as a central aspect and a set of auxiliary classes. Figure 5 only presents the main aspects that refine the agent knowledge classes incorporating specific agent features.

The Interaction component is defined as an abstract aspect that introduces interaction capabilities (inbox, outbox, sensors, effectors, parsers) in the `Agent` class. It also intercepts domain classes and sensors in the agent environment to enable the message reception by means of AspectJ pointcuts and advices. Finally, the `Interaction` aspect defines two abstract pointcuts and some abstract methods. The abstract pointcuts are used to define specific points in role aspects and plan classes where internal messages must be sent. The abstract methods are specialized to create and initialize specific sensors and effectors. The `Interaction` subaspects define the concrete configuration of the `Interaction` aspect by implementing the abstract pointcuts and methods. It is possible to specify a different `Interaction` subaspect for each one of the agent types or roles defined in an MAS.

The Adaptation component defines the `Adaptation` abstract aspect, which enables the `Agent` class to adapt its beliefs and plans. The belief adaptation of the `Adaptation` aspect is defined by intercepting the `receiveMsg()` method of the `Agent` class (introduced by the `Interaction` aspect). After that, specific advices and methods are responsible for updating beliefs based on external messages received by the agent. The plan adaptation, defined in the `Adaptation` aspect, intercepts the `setGoal()` method of the `Agent` class and the erroneous execution of the `execute()` method of the `Plan` subclasses. The purpose is to determine new agent plans to be executed by the agent to reach a specific goal. The `Adaptation` abstract aspect also offers abstract methods to be defined by subaspects. These subaspects

allow defining specific belief and plan adaptation for each one of the agent types or roles in MASs.
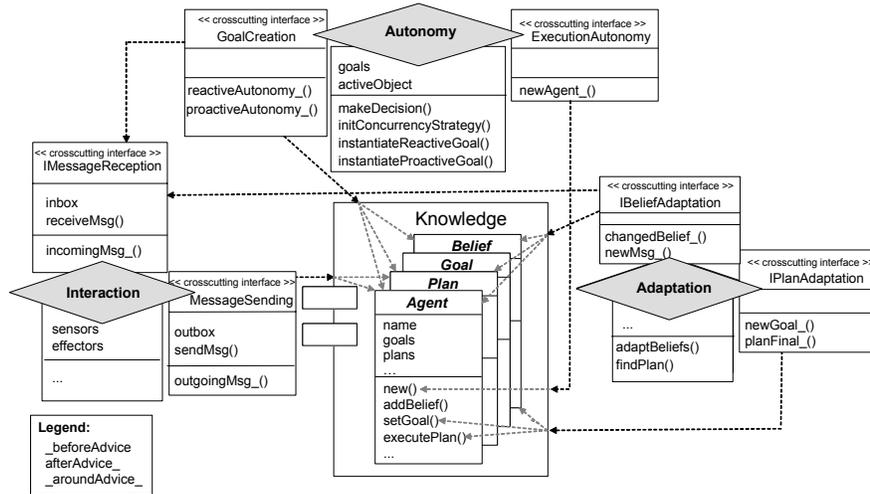

Figure 5. The Aspect-Oriented Agent Framework

The Autonomy component defines the `Autonomy` aspect, which enables the `Agent` class to instantiate and manage reactive goals and to execute concurrently several plans (execution autonomy). However, for sophisticated agent types, the `Autonomy` aspect also allows defining proactive and decision autonomy. To instantiate reactive goals, the `Autonomy` aspect also intercepts the `receiveMsg()` method of the `Agent` class. This interception is used to verify if specific external events (for instance, another agent's request) demand the instantiation of reactive goals. The execution autonomy is implemented in the `Autonomy` aspect by defining an Active Object [16], which monitors the `Agent` class's list of plans to perform to execute them in separate threads. The proactive autonomy is implemented by specifying: (i) several pointcuts in agent knowledge classes that represent specific events of interest, and (ii) an advice associated with these pointcuts, which is responsible for determining if a proactive goal must be instantiated in the occurrence of any of these events. Finally, the decision autonomy only defines a `makeDecision()` method in the `Autonomy` aspect that is invoked in the advices associated with the pointcuts of reactive and proactive goal instantiation. This method verifies whether it is necessary to execute a decision plan upon the occurrence of a specific event or upon receipt of a message. `Autonomy` subaspects can also be implemented to define specialized proactive, reactive and decision autonomy for each one of the agent types and roles defined in a MAS.

The Collaboration component is implemented by defining role aspects that introduce attributes and methods in an agent type (`Agent` class or subclass). These elements respectively define specific beliefs and behaviors of roles. Also, specific

`Plan` and `Goal` subclasses must be defined for the roles. The plans defined for a role manipulate the attributes (beliefs) and invoke methods (behaviors) introduced by the role aspect. `Goal` classes specified for a role are instantiated by an `Autonomy` subaspect that is specially created for the role. Specific `Interaction`, `Adaptation` and `Autonomy` subaspects can be defined for an agent role. Section 4 exemplifies the definition of subaspects for agent roles of a specific case study developed by our research group.

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

- interaction feature: concrete sensors and effectors specially tailored to specific agent platforms (such as JADE [1]);
- autonomy feature: concrete concurrency strategies (such as "thread pool" and "a thread per request") used by the active object [16] to implement the agent's execution autonomy.

3.3. The Code Generator

In the configuration knowledge of the generative approach, we implemented a code generator as an Eclipse plug-in [24]. This generator maps abstractions of Agent-DSL models into the components of the AO agent architecture. The AO framework (Section 3.2) supports the implementation of agent architectures. The main task of the generator is to instantiate the framework for a given multi-agent system. It creates subclasses and subaspects for specific hot-spots of the framework. Depending on the Agent-DSL model provided, the code relative to new agent types (or roles) and their respective agent properties are generated.

The implementation of the code generator was accomplished by using EMF technology. The EMF representation of the agent architecture was supported by the definition of an architectural model. This model is responsible for specifying an architecture that will be generated. An architectural model aggregates the components of an architecture. Each component is composed of classes, aspects and templates. Templates make it possible to define some class or aspect that needs to be customized, based on information collected by a DSL. The architectural model was implemented as an EMF model, similar to the Agent-DSL (Section 3.1).

Figure 6 shows the architectural model of the AO agent framework. It is composed of the classes and aspects defined in the framework. This architectural model also contains several templates that are used to express structure and behavior of classes and aspects that we want to generate. Java Emitter Templates (JET) a generic template engine of the EMF, was used to implement and process the templates. Examples of templates are: (i) concrete instances of hot-spots (classes or aspects), such as specific agent type classes, specific agenthood subaspects; (ii) specific agent plans and goals classes; and (iii) specific role aspects. The Agent-DSL collects the information required in the code generation to customize these templates for each specific agent. Each code template defines the specific information of the Agent-DSL model to be used during its customization.
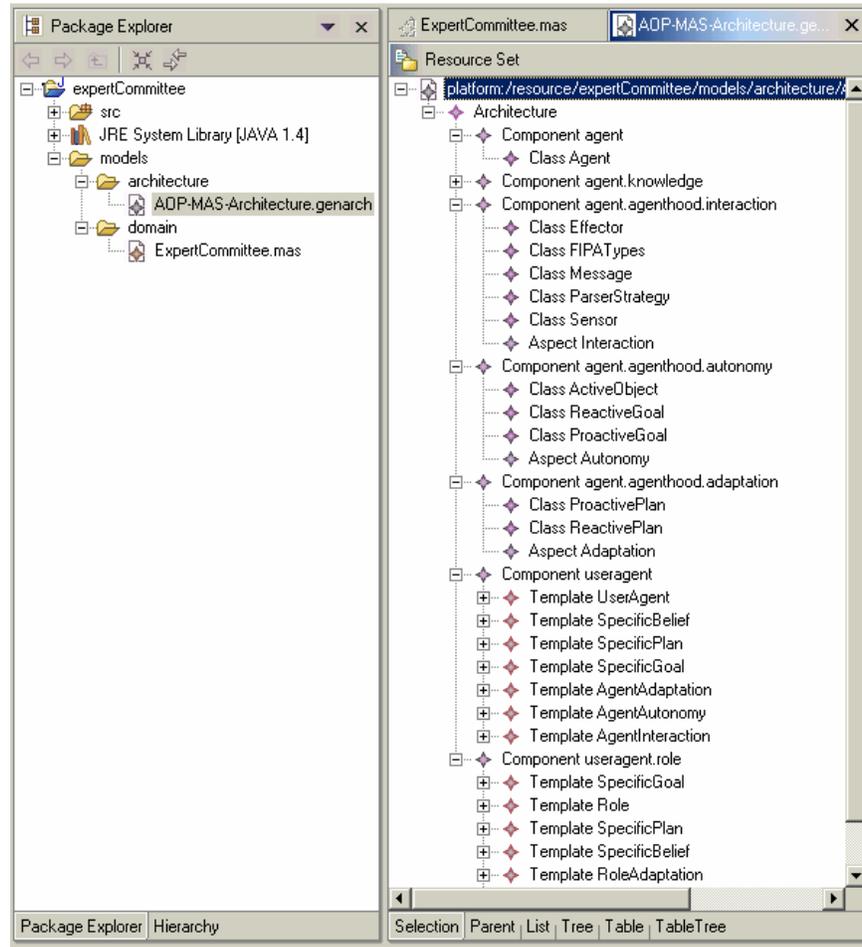
Figure 6. The Architectural Model of the AO Agent Framework

Thus the code generator uses two EMF models to instantiate the AO framework: an Agent-DSL model and an architectural model. Different agent architectures can be generated depending on the Agent-DSL and the architectural model informed by the agent developers. The plug-in of the code generator includes a wizard in the Eclipse workbench to start the process of code generation. The wizard requests from the user: (i) a source folder in a Java project to store the classes and aspects generated, (ii) the Agent-DSL model of the MAS to be generated, and (iii) the architectural model that describes the AO agent framework. During the generation process, the code generator traverses the architectural model and it proceeds as follows: (i) for each component encountered it generates a correspondent Java package; (ii) for each class and aspect encountered it loads the correspondent element in the Java project; (iii) for each template encountered it processes this element using the information collected by the

Agent-DSL model, and it loads the final element generated (class or aspect) in the specified Java project. Although we have used the architectural model of the AO agent framework in the definition of our generative approach, a different architectural model could be used during the generation process. In this case, different classes, aspects and templates could be included in the architecture model. Also, the customization of these new templates could be redefined based on information collected by the Agent-DSL model.

4. Using the Generative Approach in a Case Study

We have used the generative approach for the development of the ExpertCommittee (EC) system, which is a case study undertaken by our research group [9]. EC is an open system that supports the management of paper submissions and the reviewing process for a conference. Software agents have been introduced to EC in order to assist its users with time-consuming activities and automate repetitive user tasks. EC agents are software assistants that represent paper authors, chairs, PC members and reviewers and coordinate their activities. The EC system also includes information agents. JADE [1] is used as the communication platform in this system. As a consequence, the EC system encompasses sensors and effectors for the JADE platform.

Figures 7 and 8 show several elements generated for the EC system. Several classes and aspects are generated based on its Agent-DSL model and on the JET source templates included in the architecture model of the AO agent framework (section 3.3). First, it is generated the `ResearchUserAgent` class, which represents a specific agent type. The roles played by instances of this class are also generated. The figures represent two of them: the `Chair` and `Reviewer` aspects. Each role aspect introduces the role-specific knowledge in the `ResearchUserAgent` class. Specific `Plan` and `Goal` classes are also generated to the two roles. Figure 7 depicts the specific agent, roles and plans generated for the EC system. The code templates used to generate each of these classes and aspects are customized using the specific Agent-DSL model for the EC system. The `ResearcherUserAgent` class, for example, is customized using the agent name and beliefs collected by the Agent-DSL model. On the other hand, the `Chair` and `Reviewer` aspects use the role name and role beliefs.

Different `Interaction`, `Adaptation` and `Autonomy` subaspects are generated for each one of the roles in the EC system. Figure 8 presents these subaspects. For instance, the `ChairInteraction`, `ChairAdaptation` and `ChairAutonomy` aspects are produced to agents playing the chair role. `ChairInteraction` initializes JADE sensors and effectors to be used by the agents playing the chair role. `ChairAdaptation` realizes specific belief and plan adaptation of the chair role. Finally, `ChairAutonomy` defines: (i) a reactive autonomy – to instantiate specific goals when receiving external messages from reviewer agents; (ii) a proactive autonomy – to instantiate specific goals when internal events occur; and (iii) an execution autonomy – which defines a "thread per request" concurrency strategy to execute agent plans.
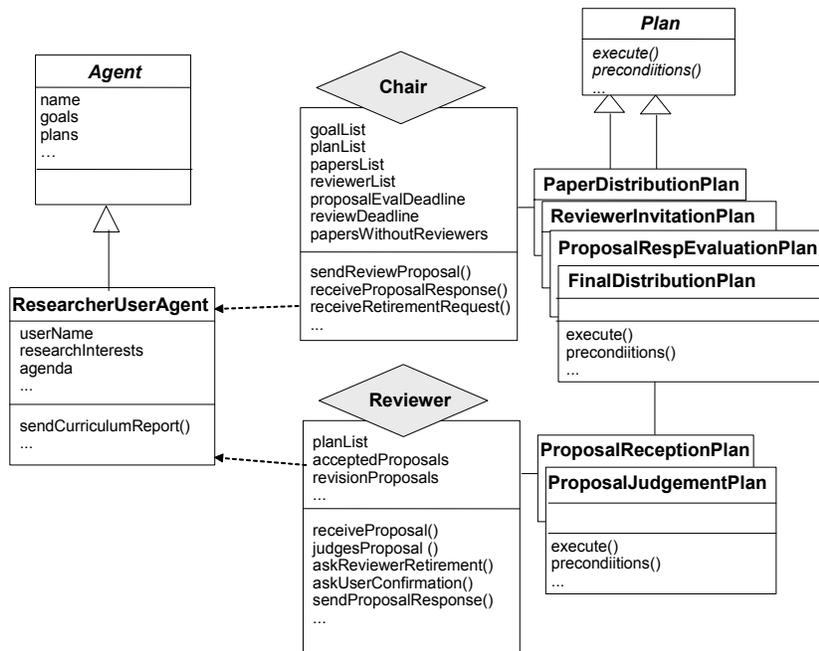
Figure 7. Specific Agent and Roles Generated for the ExpertCommittee MAS

5 Related Work

Agent-based software engineering has been studied from different perspectives, including agent-oriented methodologies and languages for higher-level development phases [15, 27], and implementation frameworks [1, 18]. Existing agent-oriented methodologies, such as Gaia [28], provide limited support to deal with concerns that are internal to software agents. In addition, they usually do not address the code generation of MASs from high-level specification. Implementation frameworks provide object-oriented APIs for MAS development, without providing guidelines for the modularization of agent concerns. Besides these frameworks do not deal with the modeling and implementation of agent crosscutting concerns typically encountered in MASs [9, 11, 12, 13, 14].

Cossentino et al [6] have proposed PASSI (Process for Agent Societies Specification and Implementation), a methodology to specify, design and implement MASs. This methodology proposes the organization of the MAS development process in different phases from the requirements specification through to system deployment. Each phase focuses on the definition or refinement of a system model. Many PASSI models are adaptations of UML standard models, such as use-case, class and activity diagrams, which incorporate agent-oriented abstractions. The use of class diagrams in

the design of MASs brings the facility to generate the skeletons of many classes of the system. The authors have also explored the reuse of recurring agent design patterns to improve the quantity and quality of code generated. However, the PASSI approach does not support the systematic modularization and generation of code relative to crosscutting agent concerns.
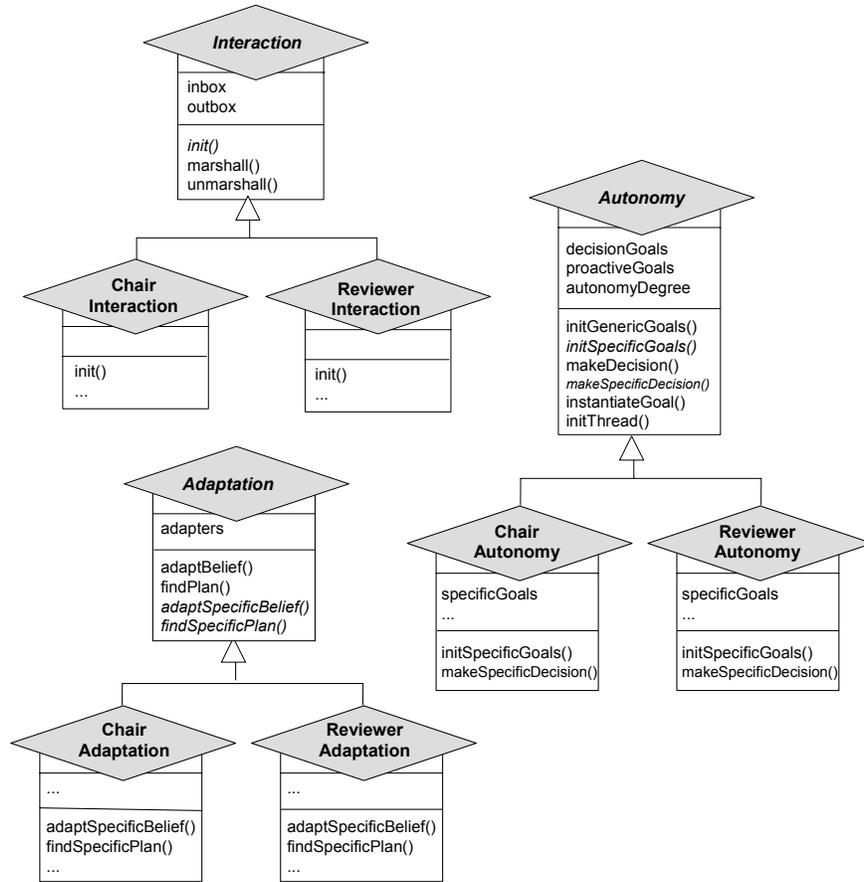

Figure 8. Specific Agenthood Subaspects Generated for the ExpertCommitte MAS

Pace et al [22] have developed the Smartweaver approach. Their approach provides assistance for the development of MAS applications by means of integration of agent-oriented and aspect-oriented frameworks. The authors demonstrate the application of their approach which consists of two components: (i) Bubble [22] – an agent-oriented framework used to the implementation of reactive agents; and (ii)

Aspect-Moderator [5] – an AO framework that supports the coordination between functional components and aspects. Aspect-Moderator is used to capture typical crosscutting concerns, such as concurrency, logging, and event handling. The Smartweaver approach systematically addresses the incorporation of aspects in agent models. However, the code generation is limited and it does not support essential agent concerns, including autonomy, interaction, and adaptation.

6 Conclusions and Ongoing Work

We have presented a generative approach for the development of MASs. The main purpose of our approach is to explore the domain of MASs to enable the code generation of agent architectures. The generative approach makes it possible to deal with orthogonal (non-crosscutting) and crosscutting agent features since early development phases in a uniform way.

Compared with existing MASs development approaches [1, 15, 18, 27], our work brings important benefits. The generative approach is flexible in the sense that it allows the problem and solution spaces to evolve independently. In problem space, new DSLs can be created to address different agent features or current DSLs can be adapted to address a better understanding of a specific agent feature or to add specificities of an MAS. Moreover, the generative approach is also very practical. It defines clearly the mapping between high-level features and implementation components (classes, aspects) of agent architectures in the code generator. It also offers a clear separation of orthogonal and crosscutting agent features in problem and solution spaces.

The use of aspect-oriented technologies in the agent architecture also brings valuable advantages to our approach. The implementation of the code generator was simplified because crosscutting agent features in the Agent-DSL are directly mapped to aspect-oriented abstractions. Using only object-oriented abstractions, crosscutting agent features need to be composed in the code of classes by the code generator. It makes the generator more complex and difficult to be implemented.

The work presented in this paper represents the current status of the generative approach. Different studies are being developed to improve our capacity to generate MAS architectures. In the problem space, we are extending the Agent-DSL to allow for the modeling of other relevant MASs concerns, such as, agent coordination, learning and mobility. In the solution space, an ongoing research project is the definition of an architectural definition language (ADL) that supports the definition of modular and aspectual components. This ADL will be used to formalize aspect-oriented architectures. We claim to enable the specification of AO architectures at a high-level independent of technologies, similar to MDA [21].

Regarding the configuration knowledge (code generators) of the generative approach, we intend to offer flexible ways to specify the transformations of: (i) elements in the Agent-DSL to elements (components, aspects and interfaces) of the ADL; and (ii) elements of the ADL to concrete technologies (for instance, Java and AspectJ). Finally, we plan to develop new and more complex case studies in order to better evaluate the usability and usefulness of our generative approach.

References

[1] 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.

[2] G. Booch, I. Jacobson, J. Rumbaugh. "Unified Modeling Language - User's ad Guide". Addison-Wesley,1999.

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

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

[5] C. Constantinides, A. Bader, T. Elrad, M. Fayad. "Designing an Aspect-Oriented Framework". Computing Surveys, 32:41, 2000.

[6] M. Cossentino, M. Potts. "A CASE tool supported methodology for the design of multi-agent systems." In Proc. of the 2002 International Conference on Software Engineering Research and Practice (SERP'02), Las Vegas, USA, June 2002.

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

[8] M. Fayad, D. Schmidt, R. Johnson. "Building Application Frameworks: Object-Oriented Foundations of Framework Design". John Wiley & Sons, September 1999.

[9] A. Garcia. "From Objects to Agents: An Aspect-Oriented Approach." PhD Thesis, PUC-Rio, April 2004.

[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, U. Kulesza, C. Lucena. "Separation of Concerns in Open Multi-Agent Systems: An Architectural Approach." Proceedings of the SELMAS'04, Edinburgh, May 2004.

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

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

[16] R. Lavender, D. Schmidt. "Active Object: an Object Behavioral Pattern for Concurrent Programming." In: Pattern Languages of Program Design, Addison-Wesley, 1996.

[17] 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.

[18] E. Kendall, et al. "A Framework for Agent Systems." Implementing Application Frameworks – Object-Oriented Frameworks at Work, M. Fayad et al. (eds.). John Wiley & Sons: 1999.

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

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

[21] J. Miller, and J. Mukerfi, MDA Guide Version 1.0, Object Management Group, Document Number: omg/2003-05-01, May, 2003.

[22] A. D. Pace, F. Trilnik, M. Campo. "Assisting the Development of Aspect-Based Multi-Agent Systems Using the Smartweaver Approach." In: "Software Engineering for Large-Scale Multi-Agent Systems." Springer, LNCS 2603, March 2003, pp. 165-181.

[23] R. Prieto-Diaz, G. Arango. Domain Analysis and Software Systems Modeling. IEEE Computer Society Press, 1991.

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

[25] V. Silva, et al. "Taming Agents and Objects in Software Engineering." In: "Software Engineering for Large-Scale Multi-Agent Systems." Springer, LNCS 2603, March 2003, , pp. 1-26.

[26] P. Tarr, et al. "N Degrees of Separation: Multi-Dimensional Separation of Concerns." Proceedings of the 21st International Conference on Software Engineering, May 1999.

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

[28] M. Wooldridge, N. Jennings, D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. Journal of Autonomous Agents and MASs, Vol. 3, 2000, pp. 285–312.