Aspectizing Multi-agent Systems: From Architecture to Implementation

Alessandro Garcia, Uirá Kulesza, and Carlos Lucena

PUC-Rio, Computer Science Department, LES, SoC+Agents Group,
Rua Marques de São Vicente, 225 - 22453-900, Rio de Janeiro, RJ, Brazil
{afgarcia,uira,lucena}@inf.puc-rio.br
http://www.teccomm.les.inf.puc-rio.br/socagents

Abstract. Agent architectures have to cope with a number of internal properties (concerns), such as autonomy, learning, and mobility. As the agent complexity increases, these agent properties crosscut each other and the agent's basic functionality. In addition, multi-agent systems encompass multiple agent types with heterogeneous architectures. Each of these agent types has different properties, which need to be composed in different ways. In this context, the separation and the flexible composition of agent concerns are crucial for the construction of heterogeneous agent architectures. Moreover the separation of agent concerns needs to be guaranteed throughout the different development phases, especially from the architectural to the implementation phase. Existing approaches do not provide appropriate support for the modularization of agent properties at the architectural stage, and do not promote a smooth transition to the system implementation. This paper presents an aspect-oriented method that allows for a better separation of concerns, supporting the systematic *aspectization* of agent properties through the architectural definition, detailed design and implementation. A multi-agent system for paper reviewing management is assumed as a case study through this paper to show the applicability of our proposal.

1 Introduction

Multi-agent systems (MASs) are composed of heterogeneous agent types with distinct agent properties (concerns), such as adaptation, mobility, collaboration, and learning. The architecture of each agent type in the system incorporates different concerns to be composed in different ways [4, 12]. None is more serious than the difficulty to modularize and compose multiple agent properties [4, 12], requiring a flexible architectural approach. These agent properties are typically overlapping and crosscut the agent's basic functionality [4, 12, 14]. The basic functionalities of agents already are quite complicated, and so agent properties should be designed separately from the agents' basic behaviors [14].

The degrees to which quality requirements (e.g. reusability and maintainability) are met on a MAS are largely dependent on its software architecture [1]. Hence, if an agent architecture that includes suitable support for the separate handling of its multiple properties is chosen from the outset, it is more likely that distinct quality attributes will be achieved throughout the development of multi-agent systems. In addition, the transition of the architectural specification to the detailed design and implementation should be straightforward.

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 121–143, 2005.
© Springer-Verlag Berlin Heidelberg 2005

However, little work has been reported so far on the definition of a development method to structure agent concerns in software systems starting at preliminary development stages. The design of multiple agent concerns with existing architectural approaches [19, 21, 26] usually increases, rather than decreases, their complexity and, consequently, it makes more difficult the task of building high-quality MASs. The perceived low quality of existing multi-agent systems is often attributed to a poor architectural design related to the agent properties [5, 11, 12, 14]. Moreover, software developers usually postpone the modularization of agent properties to the implementation stage. The agent properties are in general introduced into the software system in an ad hoc way [11, 12, 26], leading to agent architectures that are difficult to understand, maintain and reuse.

In this context, this paper presents an aspect-oriented method to support the separation of agent-specific concerns from the architecture to the implementation stages. The contributions are threefold: (i) a set of architectural guidelines to *aspectize* agent concerns on the construction of heterogeneous agent architectures - these architectural steps prescribe solutions independent of programming languages or MAS implementation frameworks [15], (ii) a set of guidelines to enable the detailed design and implementation of aspect-oriented agent architectures, and (iii) a case study of the proposed approach involving a MAS for paper reviewing management.

The basic idea of our proposal is the *aspectization* of agent architectures, using aspect-oriented abstractions to modularize agent-specific concerns at the architectural level. Aspect-oriented software development [9, 10] is an evolving paradigm to modularize concerns, which existing paradigms are not able to capture explicitly. It 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. However, aspect-oriented approaches have been rarely applied to the MAS domain [6, 12].

The remainder of this paper is organized as follows. Section 2 presents the essential concerns in the development of software agents, and explains why many agent concerns are crosscutting. Section 3 surveys and analyses existing architectural approaches that aim to support the separation of agent-specific concerns. Section 4 presents the notion of aspect-oriented agent architectures and our aspect-oriented method. Section 5 applies the proposed approach to an example. Section 6 discusses the relative advantages and disadvantages of applying the proposed approach. Section 7 discusses related work. Section 8 presents some concluding remarks.

2 Concerns in Agent Architectures

This section presents the essential concerns in the development of software agents. The main concerns are presented in *italic* throughout the section. A concern is some part of a MAS that we want to treat as a single conceptual unit [29]. Agent concerns are modularized throughout software development using different abstractions provided by languages, methods and tools.

A MAS is composed of a set of entities. These entities comprise different types of *agents* and *objects* that are immersed in *environments* [30]. Both objects and agents provide *services* to their clients. However, objects are non-autonomous entities that represent passive system elements. An agent is an interactive, adaptive, autonomous

entity that acts on the environment and manipulate objects [30-33]. As a consequence, the internal architecture of a software agent includes special concerns, which are classified in two categories: *agenthood concerns* (Section 2.1) and *additional concerns* (Section 2.2).

2.1 Agenthood Concerns

Agenthood concerns are the features incorporated by all the agent architectures independently from the agent type. Agenthood usually consists of the *basic agent concerns* – the agent services and the knowledge - and some behavioral properties. Although there is no widely accepted definition of agenthood, autonomy, interaction, and adaptation are considered agenthood properties of software agents, while collaboration, roles, learning, and mobility are neither necessary nor sufficient conditions for agenthood [30-33].

Knowledge. There are different proposed models for knowledge structuring [34, 35], but the knowledge elements are often expressed by *beliefs*, *goals*, *actions*, and *plans* [26, 34, 35]. This work focuses on such a knowledge-structuring model because many projects consider the belief-desire-intention (BDI) model [34] to be the base line for describing the agent knowledge [19, 21, 26]. The agent's beliefs are knowledge elements that describe information about the agent itself, the environment, and its partners. A goal may be realized through different plans. A plan describes a strategy to achieve an internal goal of the agent, and the selection of plans is based on agent beliefs. Actions and plans are used to implement the agent services.

Interaction. The interaction concern is the agent property that implements the communication with the external environment. The interaction behavior basically consists of *receiving messages* and *sending messages* to other agents through *sensors* and *effectors*. Since an message is received, it is *unmarshaled* and stored in an agent *inbox*. When an agent is performing actions and plans, it needs to send messages to the other agents. A message is sent from a simple action or from a plan. The sent messages are *marshaled* and stored in an *outbox*. Agent messages are structured according an agent communication language (ACL) [36]. Since different agents can use different ACLs, messages are translated to an internal message style used by the agent. The interaction protocol can also implement a *sensory behavior*, which consists of observing events in the environment objects.

Adaptation. The adaptation concern is the agent property that modifies the agent according to external and internal events [37, 38]. There are two kinds of adaptation: *knowledge adaptation* and *behavior adaptation*. They follow the same basic protocol, which consists of observing relevant environmental or internal events, gathering the information needed, selecting and invoking the associated adapters [37]. However, knowledge adaptation results in the modification of some piece of the agent knowledge. The behavior adaptation results in either the plan cancellation or the selection of new plans which should be executed next. Sophisticated adapters include reasoning techniques [37, 38] and planners [37, 38].

Autonomy. Autonomy usually means that an agent has control over its own actions and can act independently of others [31, 39, 40]. To be autonomous, the agent must

[30, 31, 39, 40]: (i) *create* its own goals on the basis of internal and external events, (ii) *make decisions* on goal instantiations, (iii) have its own control threads (*execution autonomy*), and (iv) create proactive goals without direct external intervention (proactiveness). The achievement of proactive goals depends on their *degree of autonomy*. The degree of autonomy increases or decreases according to successes and failures of agent actions in the past. These are the dimensions of agent autonomy commonly found on the literature [30, 31, 39, 40].

2.2 Additional Concerns

In addition to the agenthood concerns, the agent developer may have to face additional concerns. The agent may have move from an environment to another, gain new knowledge to improve its performance, and collaborate with other agents.

Mobility. The mobility concern encompasses the behavior to support the agent travels towards remote environments. During the execution of its plans, a mobile agent may have to move from one environment in a network to another in order to achieve its goals. Many facets of the mobility strategy need to be considered [14], including the specification of the *mobile elements*, the descriptions of when the agent should move, the *departure* to remote environments, the *return* to the home environment, and the control of its *itinerary*. In this work, we have focused on weak mobility in which only program code and instance data are moved [16]. In fact, most mobility frameworks support weak mobility [14, 16].

Learning. The learning property involves the agent behavior responsible for refining or gaining knowledge. Cognitive agents learn based on experience as a result of their own actions, their mistakes, the successive interactions with the external environment and the collaborations with other agents [38, 41]. Agents employ different learning techniques, but the general *learning protocol* is the following [38, 41]: (i) an event is detected as relevant, (ii) the event is caught and the information is gathered for the learning purpose, (iii) the learning algorithm processes the gathered information, (iv) the information is stored and alternatively leads to new conclusions, (v) whether a new conclusion is achieved, the knowledge agent is adapted.

Collaboration. Collaboration is viewed as a more sophisticated form of interaction, since it involves collaboration protocols and roles [5]. Collaboration protocols define the ways software agents can interact with other agents in a multi-agent organization. Agents play different *roles* in pursuing their individual goals and work together with other agents in multiple contexts [5]. The role structure is similar to the agent structure. As an agent, a role has *beliefs, goals, actions* and *plans* for carrying out the collaborations with other agents. It may have specific behaviors for interacting with other agents, specific decision algorithms, and specific adaptation strategies. It may also have specialized behaviors related to the additional properties. However, a role cannot exist without an agent.

2.3 Crosscutting Agent Concerns

Several authors have identified that most of the agent properties are often crosscutting, such as mobility [14,54], interaction [4,5], learning [28,53], autonomy [19,44],

and collaboration [14,27]. Some empirical studies confirm their findings [4,7,12]. Fig. 1 shows a partial representation of a multi-agent system [5], which will be used in Section 5 to show the applicability of our proposal. Each set of classes, surrounded by a gray rectangle, has the main purpose of modularizing a specific agent concern, namely interaction, environment, basic concerns, learning, and collaboration.

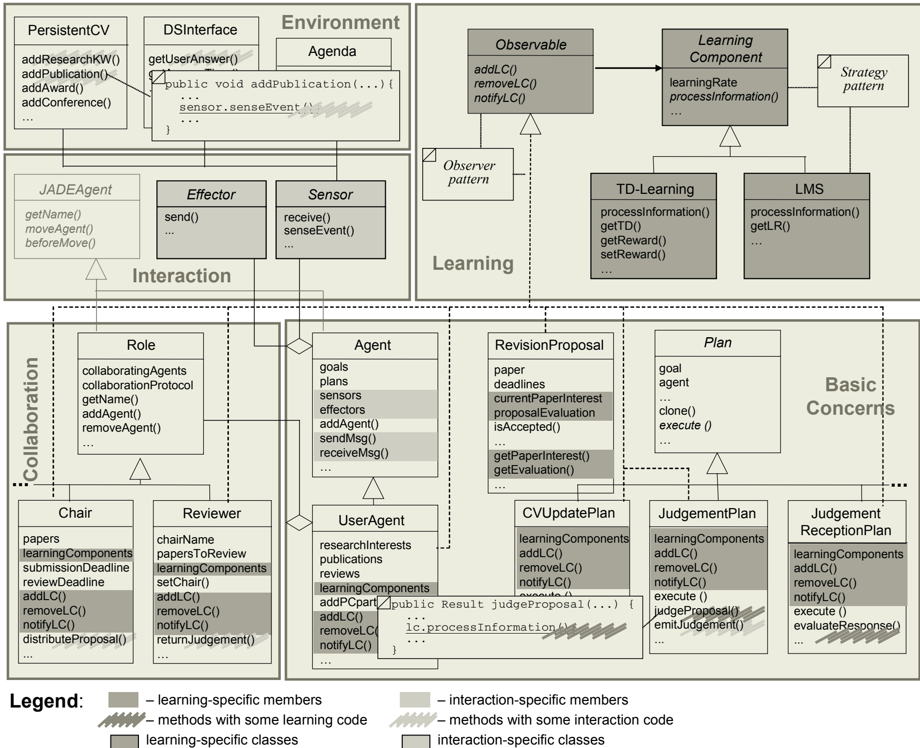


Fig. 1. Crosscutting Agent Concerns

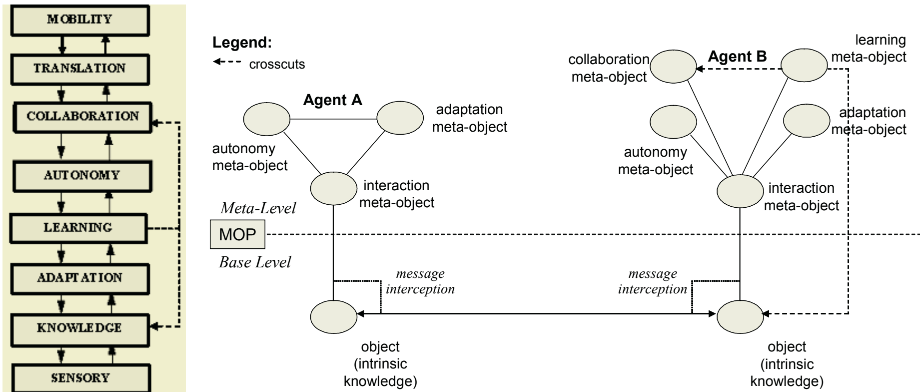
However, note that, for example, the learning concern crosscuts classes implementing other agent concerns; it has a huge impact on the basic agent structure and the collaboration design. Although part of the learning concern is localized in the classes of the Strategy and Observer patterns, learning-specific code replicates and spreads across several class hierarchies of a software agent. Several participants have to implement the observation mechanism and the gathering information and, as a consequence, have learning code in them. Some classes (e.g. RevisionProposal class) have learning-specific knowledge. Adding or removing the learning code from classes requires invasive changes in those classes. Note that even if we try to refactor the object-oriented solution presented in Fig. 1, we cannot find a more modular solution. This problem happens because learning is a crosscutting concern independently of the object-oriented decomposition used [53]. Fig. 1 also illustrates similar problems for the interaction concern, which is usually crosscutting.

3 Existing Architectural Approaches

There are some architectural approaches [19, 21, 26] to promote the separation of agenthood and additional concerns. This section provides a comparison and evaluation of these architectures as well as the identification of the primary limitations of applying them to the separation and integration of crosscutting agent concerns. They rest on traditional architectural patterns, such as the Layers pattern [26], Reflection pattern [19], and the Mediator pattern [21].

Layered Agent Architectures. Kendall et al [26] propose the Layered Agent architectural pattern with multiple layers for the separate representation of the agent concerns (Fig 2a). The interaction concern is modularized in two layers: the translation layer and the sensory layer. The layered architecture establishes a composition style in which all of the interactions feature two-way information flow and only subjacent layers communicate with each other. However, this composition style is very restrictive since agent properties can interact with each other in multiple ways (Section 2.3).

For example, as the agent complexity increases, the learning concern cuts across the different agent layers, such as knowledge and collaboration (Fig 2a). Moreover the evolution of this design approach is cumbersome since removing any of these layers is not a trivial matter; it requires the reconfiguration of the adjacent layers. This layered agent architecture promotes some degree of separation only at the architecture level. When the architectural components – the layers – are decomposed using design patterns, as proposed by the Kendall's approach [26], the architectural separation of agent concerns is degenerated at the detailed design and implementation levels. Previous work has highlighted similar shortcomings of layered architectures [23].



(a) Learning: Crosscutting Layers

(b) Learning: Crosscutting Meta-Objects

Fig. 2. Layered vs. Reflective Agent Architectures

Reflective Agent Architectures. Amandi [19] proposes an architectural approach based on the Reflection architectural pattern [1], called Brainstorm. Reflective software architectures are organized in two levels: the base level that contains the objects, and the meta-level composed of meta-objects. A MOP (meta-object protocol) implements the interface between the base-level and the meta-level. The MOP is responsi-

ble for redirecting the control flow at the base-level to the meta-level in certain execution points of base-level objects. Brainstorm explores meta-objects as abstractions to support the modularization of agent concerns. Each agent concern is modularized in specific meta-objects and associated with based-level objects, which implement the agent's basic concerns (Fig 2b).

Reflective agent architectures improve the separation of agent concerns since meta-objects localize them. However, this architecture introduces some drawbacks. First of all, the composition of multiple meta-objects is not trivial. Meta-objects are objects and their composition rests on inheritance and delegation mechanisms, leading in turn to the problem of crosscutting concerns (Section 2.3). Fig 2b illustrates this problem for the learning concern. Second, a meta-object is often associated with one object. It is very restrictive since several agent properties (meta-objects) can affect directly the basic agent concerns (objects).

Mediator-Based Agent Architectures. The use of mediators is an architectural approach to address the composition of agent concerns that interact in multiple ways. Composition patterns, such as the Mediator pattern [20] and the Composite pattern [20], are mediator-oriented solutions. They provide means of allowing integration of agent properties using a central component, the mediator. The Mediator pattern, for instance, defines a mediator component that encapsulates how a set of components, the colleagues, interact with each other. This solution promotes loose coupling by keeping components from referring to each other explicitly, and it lets the agent developers vary their interaction independently. The Skeleton Agent framework [21] realizes a mediator-based architecture by implementing the Composite pattern. The use of a mediator-based architecture leads to the following problems [4, 6, 7]: (i) the encapsulation of the agent's basic functionality is lost, (ii) agent concerns are scattered and tangled up with each other in the mediator-based design, and (iii) the construction of heterogeneous agent types is difficult as a result of (ii).

4 Aspectizing Software Agents: From Architecture to Implementation

This section overviews aspect-oriented agent architectures [5], which are the foundation of our approach (Section 4.1). This section also presents the proposed guidelines to aspectize MASs. The guidelines are grouped in terms of the different development phases, namely *architecture definition* (Section 4.2), *detailed design* (Section 4.3), and *implementation* (Section 4.4). The guidelines will be applied to an example (Section 5) in order to show the use of our method in practice.

4.1 Aspect-Oriented Agent Architectures

In this paper, the architecture modeling is based on the aSideML language [2], which is a UML extension for representing aspects at different levels of abstraction. Aspects are modular units to encapsulate crosscutting concerns [9, 10]; a*spectual components* (or *architectural aspects*) are aspects [9, 10] at the architectural level. The aSideML language provides two distinct modes for presenting an aspect: (i) condensed or architectural view, and (ii) full view or detailed design view (see Section 4.3). The archi-

tectural view of an aspect suppresses all information about its inner elements. Architectural aspects are UML components represented as diamonds. Each of the aspectual components is related to more than one architectural component, representing their crosscutting nature.

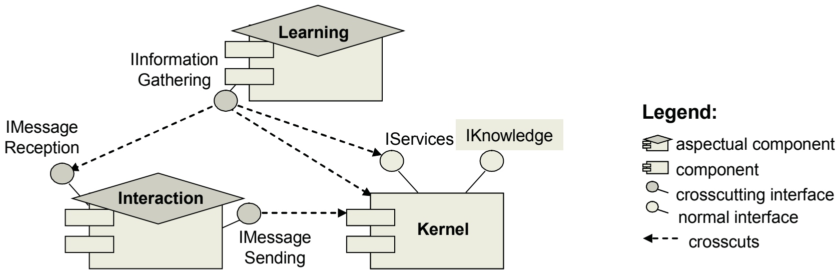


Fig. 3. Aspect-Oriented Architecture

Fig. 3 illustrates some architectural components and their interfaces. Each interface is displayed as a small circle with the interface name placed next to the circle. Each architectural component has one or more interfaces. The interfaces are categorized in two groups: (i) *normal interfaces*, and (ii) *crosscutting interfaces*. A crosscutting interface is different from a normal interface. The latter only provides services to other components. Crosscutting interfaces specify when and how an architectural aspect affects other architectural components. The purpose of crosscutting interfaces is to modularize parts of a concern which usually crosscut other concerns in traditional kinds of decomposition, such as object-orientation (Section 2.3). For example, Fig. 3 shows the IInformation Gathering interface in the Learning component that modularizes the event observation and information gathering, which are issues that usually crosscut the other concerns (Section 2.3). An aspectual component conforms to a set of crosscutting interfaces. Normal interfaces are colored in white and crosscutting ones in gray.

An aspect-oriented agent architecture provides components for *aspectizing* crosscutting agent concerns (Section 2.3). Each agenthood and additional property is modularized as an individual *aspect* [9, 10]. The aspect-oriented architecture is composed of two kinds of architectural components: (i) the *Kernel* component that modularizes the basic agent concerns, and (ii) *aspectual components* (or *architectural aspects*) that separate the crosscutting agent concerns from each other and from the Kernel component. Fig. 3 shows a partial representation of an aspect-oriented agent architecture; it illustrates a Kernel component, two aspectual components, and crosscutting relationships.

The Kernel component implements the services provided for the agent's clients. The Kernel component realizes an interface that makes available services implemented by the agent. This component is also responsible for modularizing the knowledge elements, such as actions, plans, goals, and beliefs. An aspectual component can realize more than one crosscutting interface since it can crosscut multiple agent components in different ways. The interface of an architectural aspect can crosscut the Kernel component and other architectural aspects. An aspect interface crosscuts either internal elements of an agent component or elements of other interfaces. The first case

means that the architectural aspect affects the internal structure or dynamic behavior of the agent component. The second case means that the aspect affects directly an agent architectural aspect.

4.2 Steps for the Architectural Stage

This section presents a set of guidelines to assist software engineers in the design of aspect-oriented agent architectures. The guidelines assist in the configuration of the architectural components and their composition through the specification of normal and crosscutting interfaces in a stepwise fashion. The definition of a crosscutting interface involves the description of the architectural components or interfaces affected by that crosscutting interface. This process determines the relationships between the agent's architectural components, abstracting the internal intricacies of each component.

The steps and substeps should be followed for the architectural definition of each of the system's agents. The following subsections walk through the guidelines to generate the aspect-oriented agent architecture. The steps A1-A4, D1-D4 are mandatory for all agent types since they represent guidelines for dealing with the agenthood concerns. The remaining steps are optional because they comprise the additional concerns.

Step A1. Define the Kernel component.

a) Define the agent's basic interfaces. Each agent can have one or more normal interfaces which make the agent services available to the environment.
b) Define the normal interface for the agent knowledge maintenance.

Step A2. Define the Interaction aspectual component.

a) Define the crosscutting interfaces for the sensory behavior.
b) Define the crosscutting interfaces for message reception.
c) Define the crosscutting interfaces for message sending.

Step A3. Define the Adaptation aspectual component.

a) Define the crosscutting interfaces for knowledge adaptation.
b) Define the crosscutting interfaces for behavior adaptation.

Step A4. Define the Autonomy aspectual component.

a) Define the crosscutting interface for addressing the thread management. This interface specifies the policies for starting and finalizing agent threads. This interface may be not necessary in the cases where the thread is attached to the agent by the enclosing system and not by the application.
b) Define the crosscutting interface for goal creation.
c) Define the crosscutting interfaces for controlling the autonomy degree.
d) Define the crosscutting interfaces for decision making. Reactive agents [38] only need to decide according to external events. Proactive agents [38] make decisions on the basis of both internal and external stimulus.

Step A5. Define the Mobility aspectual component.

a) Define the crosscutting interfaces for specifying the elements to be moved together with the agent.
b) Define the crosscutting interfaces for the agent departure and the agent return.

Step A6. Define the Learning aspectual component.

a) Define the crosscutting interfaces for observing the agent's internal events and gathering the contextual information.
b) Define the crosscutting interfaces for describing the learning-specific knowledge.

Step A7. Define the Collaboration aspectual component.

a) Define the crosscutting interfaces for enforcing the collaboration protocols.
b) For each role, define a Role aspectual component, and:
    a. define the role architecture by starting from Step A1.
    b. define the crosscutting interface for role binding.
    c. define the crosscutting interface for describing the role-specific knowledge.
    d. associate the Role component with the respective protocol interfaces (A7-a).

4.3 Steps for the Detailed Design Stage

Each step in this phase is associated with an architectural step (Section 4.2), refining an architectural component previously defined. A design step has two basic procedures: (i) the refinement of the corresponding architectural component, which is usually decomposed in terms of an abstract aspect, concrete aspects, and/or classes; and (ii) the refinement of the corresponding crosscutting or normal interfaces. The detailed design of a normal interface involves the definition of the services to be made available by the interface.

The detailed design of a crosscutting interface encompasses the specification of the join points, pointcuts, advices, and inter-type declarations. *Join points* are well-defined points in the dynamic execution of the system components. Examples of join points are method calls and method executions. *Pointcuts* have name and are collections of join points. *Advice* is a special method-like construct attached to pointcuts. *Inter-type declarations* introduce attributes, methods, and interface implementation declarations into the components to which the crosscutting interface is attached.

Step D1. Refine the Kernel component.

a) Create a class to represent the agent type. This class should extend a generic, abstract Agent class that captures the common behavior of all the system agents.
b) Define the main and auxiliary methods that implement the agent's basic services.
c) Define the agent actions as methods.
d) Define the agent plans as classes.Specify plan actions as methods of plan classes.
e) Define the agent beliefs as simple strings or classes.
f) The knowledge elements are subclasses of the Belief, Goal and Plan classes.

Step D2. Refine the Interaction aspectual component.

a) Define the interaction infrastructures and corresponding sensors and effectors.
b) Define the agent's internal message format.
c) Define parsers for translating external messages to the internal message format.
d) Create the abstract and concrete aspects to modularize the interaction concern.
e) Refine the sensory interfaces, defining the external objects to be observed.
f) Refine the message reception interfaces, defining the join points where messages should be received.
g) Refine the message sending interfaces, picking out the joint points where messages should be sent from the agent.

Step D3. Refine the Adaptation aspectual component.

a) Create the abstract and concrete aspects to modularize the adaptation concern.
b) Refine the knowledge adaptation interfaces, enumerating the agent's internal events to be observed.
c) Refine the behavior adaptation interfaces, enumerating the agent's internal events to be monitored.

Step D4. Refine the Autonomy aspectual component.

a) Define the reactive, decision and proactive goals for the agent.
b) Create the abstract and concrete aspects to modularize the autonomy concern.
c) Refine the thread management interface, specifying the join points where threads should be started and finalized.
d) Refine the goal creation interface, specifying the events to instantiate goals.
e) Refine the decision-making interfaces for capturing events that trigger agent decisions.
f) Refine the crosscutting interfaces for capturing the events that affect the agent's autonomy degree.

Step D5. Refine the Mobility aspectual component.

a) Create the abstract and concrete aspects to modularize the mobility concern.
b) Refine the crosscutting interfaces for mobile elements, specifying the elements to be moved together with the agent.
c) Refine the crosscutting interfaces for agent travel, picking out the join points that trigger the agent travel and the agent return.

Step D6. Refine the Learning aspectual component.

a) Create the abstract and concrete aspects to modularize the learning concern.
b) Refine the crosscutting interface for information gathering, describing the join points to provide information and trigger the learning process.
c) Refine the crosscutting interfaces for learning knowledge, specifying the attributes and methods with learning-specific knowledge.

Step D7. Refine the Collaboration aspectual component.

a) Create the abstract and concrete aspects to modularize the collaboration concern.
b) Define the collaboration protocols and the corresponding roles.
c) Refine the crosscutting interfaces for enforcing the collaboration protocols.
d) For each role:
    a. Refine the interfaces for the role binding, describing the join points where the role should be bound to the agent.
    b. Refine the interfaces for role-specific knowledge, describing the methods and attributes with role knowledge which should be introduced to the agent.
    c. Refine the role aspects by starting from Step D2.

4.4 Implementation Stage

There are several aspect-oriented programming languages to support the implementation of aspect-oriented agent architectures, such as AspectJ [10] and Hyper/J [42]. AspectJ is the most widely used programming language, which extends the Java programming language. The implementation of aspect-oriented agent architectures in AspectJ is straightforward, since this language supports the definition of inter-type

declarations, pointcuts and advices. However, there are some implementation steps that require some guidance due to AspectJ features and restrictions as summarized below. The reader can find a extensive list of implementation guidelines at [5].

For example, each agent instance must have, in general, its own instance of agenthood or additional aspect. As a consequence, the agent aspects must be instantiated per Agent instance. The current version of AspectJ supports the specification of per-object aspects. We could describe the instantiation of the agent aspects using perthis. However, the use of perthis restricts the scope of the aspect. When one AspectJ aspect is declared to be singleton or static, its scope is the whole system and the aspect can crosscut all system classes. Per-object aspects can only crosscut the object with which it is associated. Since agent concerns crosscut several classes, not only the Agent class or the Role class, the perthis clause cannot be used in this context. As a result, agent aspects are declared as singletons and introduce the methods and attributes to the Agent and Role classes as inter-type declarations.

5 ExpertCommittee: The Case Study

This section introduces a MAS in order to illustrate the application of the guidelines presented in the previous section. This system is a prototype derived from a case study undertaken in the Software Engineering Laboratory at PUC-Rio in Brazil, from herein referred to as EC (ExpertCommittee). EC is an open multi-agent system that supports the management of paper submissions and the reviewing process for a conference. The EC system has been chosen because it is a classical example of agent-based application [43] and it involves all the agenthood and additional properties. This system includes several combinations of agent concerns, which are typical of many existing agent-driven applications.

The EC system encompasses two agent types: user agents and information agents. Each agent type provides different services, but everyone is interactive, adaptive and autonomous. The architecture of each agent type has different agent properties. For simplicity purposes, this section focuses on the description of the user agents. User agents are software assistants that automate time-consuming tasks of paper authors, chairs, PC members and reviewers and coordinate their activities. Fig. 1 shows a partial representation of the object-oriented design of the EC system.

5.1 The Architectural Stage

We describe below the accomplishment of the architectural steps (Section 4.2) to define the architectural design of the user agents. Fig. 4 depicts the architectural view of the EC user agents. The user agents have all the agenthood and additional architectural aspects.

Step A1. The Kernel component of user agents has a normal interface, which makes the agent services available to the environment (A1.a). The agent services can be accessed either by sending an asynchronous request through the communication infrastructure or by directly invoking them. The agents also have a normal interface for accessing and updating the knowledge elements (A1.b). This is a private interface and is not accessible by elements external to the agent.

Step A2. User agents have an Interaction component that implements the three crosscutting interfaces: ISensory, IMessageReception, and IMessageSending. The Sensory interface senses events in environment objects (A2.a). The IMessageReception interface intercepts the messages arrival (A2.b). The IMessageSending interface defines when messages need to be sent from agent plans and actions (A2.c). The Interaction aspect's interfaces crosscut the Kernel component, the Collaboration component, and the Environment component. The Environment component represents the communication platform and the external entities observed or monitored by the agent.

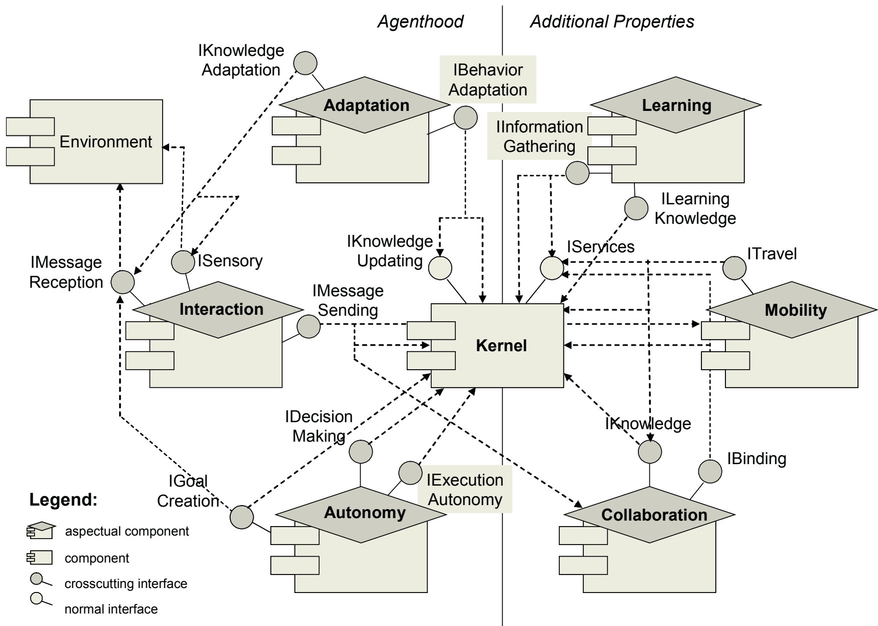


Fig. 4. The Aspect-Oriented Agent Architecture of User Agents

Step A3. The adaptive behavior of user agents involves two kinds of adaptation: knowledge adaptation and behavior adaptation. As a consequence, they have a Adaptation component that realizes the crosscutting interfaces for both of them: IKnowledgeAdaptation and IBehaviorAdaptation. The Adaptation component crosscuts the Interaction component and the Kernel component. It is connected with the former through the IKnowledgeAdaptation interface since knowledge adaptation may be required upon the receipt of external messages (A3.a). The connection with the later is because knowledge adaptation is necessary whenever given internal events happen, such as the change of beliefs. In addition, the Adaptation component crosscuts the Kernel component through the IBehaviorAdaptation interface, since the selection of a new plan is necessary whenever a new goal is set, and the plan execution may have to be canceled due to the change of specific beliefs (A3.b).

Step A4. The Autonomy component conforms to the following crosscutting interfaces: (i) IExecutionAutonomy, which specifies the kernel initialization as the join point to create the agent threads (A4.a), (ii) IGoalCreation, which crosscuts the Interaction

component and the Kernel component because it may be necessary to create goals whenever messages are received and pieces of the agent knowledge are changed (A4.b), and (iii) IDecisionMaking triggers the agent decisions (A4.d). User agents do not have an interface to control the autonomy degree (A4.c).

Step A5. User agents have a Mobility component that conforms to two crosscutting interfaces: IMobileElement and ITravel. IMobileElement specifies all the Kernel elements as mobile elements because at least the agent kernel needs to be moved when the agent departs to a remove environment (A5.a). ITravel crosscuts the Kernel and Collaboration components since the execution of actions and plans triggers agent travels across different hosts as well as the agent return to its home host (A5.b).

Step A6. User agents have a Learning component with two crosscutting interfaces. The first one, IInformationGathering, defines internal events in the Kernel component as the information sources to be observed for learning purposes (A6.a). The second one, ILearningKnowledge, specifies the learning-specific knowledge to be introduced to the agent kernel (A6.b).

Step A7. EC agents have also a Collaboration component that aggregates the roles played by the agents. The interfaces for enforcing collaboration protocols (A7.a) are not represented in Fig. 4 since EC agents do not require this feature. It determines when a given role is bound to the agent. Inner aspectual components represent the agent roles. The Collaboration component is formed by four inner Role components (A7.b), each one for a specific agent role: author, reviewer, PC member, and chair. Each of the Role components implements the IBinding interface and the IKnowledge interface.

5.2 The Detailed Design Stage

Due to space limitations, in this work, we discuss in detail only the steps involving the agenthood concerns (D2-D4). The other steps (D1, D5-D7) are shortly described. A more detailed description is found at [5]. Some figures are used to illustrate the detailed design of the architectural aspects, which crosscut several agent classes and aspects in the EC system. However, for simplification reasons, the figures only present some of these classes and aspects. The others essentially follow the same pattern. The figures represent the crosscut elements in gray.

The aSideML language (Section 4.1) also supports the modeling of the detailed design of aspects. The full view of an aspect provides a detailed description of its elements. An aspect is represented by a rectangle, like classes, with a diamond on its top. The aspect's internal structure declares the internal attributes and methods. Each crosscutting interface is presented using the rectangle symbol with compartments. The first compartment of a crosscutting interface represents inter-type declarations, and the second compartment represents pointcuts and their attached advices. The notation uses a dashed arrow to represent the crosscutting relationship.

Step D1. The Kernel component is refined as a set of classes, which represent the agent itself, and knowledge elements (goals, beliefs, and plans). The Agent class specifies the behavior common to the system's agent types. The UserAgent class extends the Agent class (D1.a), and contains the methods that implement the agent

actions and agent's basic services (D1.b, D1.c). The knowledge elements of user agents are subclasses of the Belief, Goal and Plan (D1.f). Attributes of Agent subclasses can be used to represent simple agent beliefs. Plan actions are methods of Plan subclasses (D1.d). The agent beliefs are attributes of the Agent subclasses (D1.e).

Step D2. User agents interact with the environment using two communication infrastructures: JADE [15] and a blackboard architecture (D2.a). The blackboard is used for internal communication between the agents and the JADE infrastructure is used to interact with agents external to the system. The effectors and sensors associated with these infrastructures are represented by separate class hierarchies (D2.a). The Sensor and Effector subclasses represent sensors and effectors respectively, and cooperate with environment classes. ACL [36] is the used communication language. The agents also use an internal communication language, which is also compliant to the FIPA specification [36] (D2.b). Specific classes are responsible for implementing the parsers (D2.c).

The Interaction architectural component is decomposed in an abstract aspect (D2.d), a concrete aspect (D2.d), and various auxiliary classes. Fig. 5 shows only the aspects, sensors/effectors, and the crosscut elements (in gray); it omits the auxiliary classes. The abstract aspect defines the interaction logic, which is common to all the agent types and roles. It holds the inbox, the outbox, an abstract initialization method, and methods to marshal and unmarshal the messages. This aspect also refines the three crosscutting interfaces defined in the Interaction architectural aspect: ISensory, IMessageSending, and IMessageReception.

The ISensory interface (D2.e) implements the abstract sensing pointcut that declares which methods of the environment classes must be monitored. The sensing_ advice processes the external events and updates the inbox. The sensing pointcut is also declared as abstract since the join points depend on the specific agent types and roles.

The IMessageReception interface (D2.f) introduces the receiveMsg() method to the Agent class in order to enable it to receive messages (inter-type declaration). This interface also defines an incomingMsg pointcut for intercepting executions of the method sense() on the Sensor classes; the goal is to detect the arrival of messages. This pointcut is associated with an after advice responsible for processing the incoming messages and updating the inbox.

The IMessageSending interface (D2.g) extends the Agent class to enable it to send messages, by introducing the sendMsg() method to the Agent class. The interface also defines an outgoingMsg pointcut that specifies the message senders. Note that the outgoingMsg pointcut is abstract (Fig. 5) because the join points depend on the specific agent types and roles. The pointcut is concretized in the Interaction subaspects. This pointcut contains an advice which runs after executions of those join points. The purpose of the advice is to capture the information needed to send the message and update the outbox. Note that the scattering of the interaction concern presented in Fig. 1 is overcome in the aspect-oriented solution (Fig. 5).

The concrete aspect, called UserAgentInteraction, extends the Interaction aspect to define the interaction behavior specific to the user agent. It implements the sensing pointcut by specifying DB components, GUI objects, and business logic components as environment entities to be observed. User agents monitor these elements in order to adapt their knowledge and behavior and learn about the user preferences. This spe-

cific aspect also implements the initialization methods, and the outgoingMsg pointcut that specifies join points in the agent elements from which messages need to be sent to the external world, such as methods of Plan subclasses, Agent subclasses, and Role aspects (D2.g). This pointcut also crosscuts the Mobility aspect because the user agent needs to send notification messages to local agents before moving to a remote environment and after returning to the original environment.

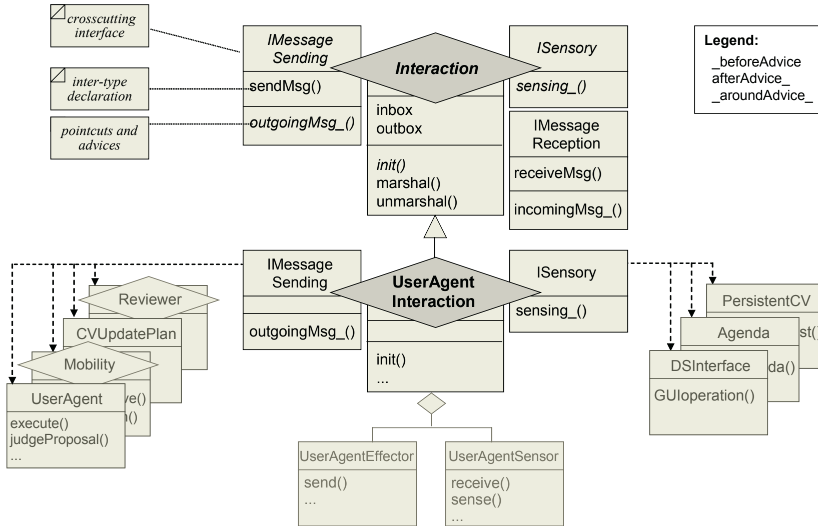


Fig. 5. The Detailed Design of the Interaction Component.

Step D3. The Adaptation architectural component of EC agents is decomposed in an abstract aspect (D3.a), a concrete aspect (D3.a), and auxiliary classes. The abstract aspect defines the generic adaptation protocol: events are sensed, conditions checked, and adapters triggered. The aspect holds the adapter methods and a list of the adapter objects. It contains the advices which either invoke either adapter methods or a specific adapter. The Adaptation aspect is extended by the UserAgentAdaptation aspect to implement the adaptive behavior for the specific context of the user agents.

The abstract aspect also implements the two crosscutting interfaces: IKnowledge-Adaptation (D3.b) and IBehaviorAdaptation (D3.c). They define pointcuts with generic events that always trigger the knowledge and behavior adaptation, independently from the agent type. The agent adaptation occurs in several circumstances: due to external events – for example, message receptions – or due to internal events, belief changes, new goal setting, exceptions thrown during a plan execution, and so forth.

Step D4. The Autonomy architectural component is refined as an abstract aspect (D4.b), a concrete aspect (D4.b), and auxiliary classes. There are various Goal subclasses to define the reactive, decision, and proactive goals of the user agents (D4.a). The DecisionPlan and ProactivePlan subclasses modularize the implementation of more sophisticated decision algorithms and proactive strategies. The abstract Autonomy aspect defines the autonomy behavior common to all the agent types in the EC system. The UserAgentAutonomy aspect extends the Autonomy aspect to implement

the autonomous behavior for the specific context of user agents. This aspect holds the decision and proactive goals, an integer number representing the autonomy degree, initialization methods, and defines the autonomy protocol. It implements three crosscutting interfaces: IExecutionAutonomy, IGoalCreation, and IDecisionMaking. These crosscutting interfaces define how the Autonomy aspect crosscuts different classes and other aspects of software agents.

The IExecutionAutonomy interface (D4.c) defines the pointcuts that specify when control threads are attached to and detached from the Agent instances. It defines the execution of Agent constructors as the join point to start the threads, and the agent destruction (execution of the method kill()) as the join point to finalize the threads. There are after advices associated with these pointcuts in order to invoke the components that implement the Active Object pattern [24].

The IGoalCreation interface (D4.d) specifies join points in the agent classes which events need to be detected to start a goal creation. It contains an advice which runs after executions of actions on agent classes, actions on beliefs classes, action on plan classes, and actions on another aspects associated with the agent (for example, Interaction aspects). Since the Autonomy aspect implements the autonomy protocol, it is associated with the agent, plan or belief classes where changes to their state may trigger a goal creation.

The IDecisionMaking interface (D4.e) specifies the receiveMsg() method on the Interaction aspect (D2) as join point because an agent often needs to decide whether and which reactive goal instance should be created depending on the received messages. After the receiveMsg() method is executed, there is an advice that takes the control on the program execution and instantiate, if necessary, the goal decisions associated with the message type. If the decision is positive, a reactive goal is instantiated. Otherwise, a decision plan sends a message to the sender notifying the agent that the service request will not be performed.

Steps D5-D7. The Mobility, Learning and Collaboration architectural aspects are also decomposed in terms of abstract aspects, concrete aspects, and auxiliary classes. Unlike the agenthood aspects, they are not associated with the Agent class because they are not part of the agenthood. They are associated only with the UserAgent class. For example, the Mobility aspects modularize the following issues: (i) the pointcuts that describe the events which may lead the agent to travel to a remote environment or to go back to the home environment, (ii) the advices responsible for checking the need for the agent roaming and for calling the mobility actions, (iii) the data structures and methods which control the agent itinerary, and (iv) the inter-type declarations that specify which agent elements are mobile and serializable. As a consequence, the agent classes are not intermingled with mobility code, therefore improving their maintainability and reusability. JADE is used as the mobility framework; some auxiliary classes connect the mobility aspects with the JADE framework. Each mobility aspect in the EC system crosscuts about 7 classes and aspects.

Learning aspects encapsulate the entire implementation of the learning concern, including the learning-specific knowledge and the information gathering. Fig. 6 shows that the Learning aspect separates the learning protocol from the kernel and other aspects, such as UserAgent class, Plan classes, and role aspects. The Learning aspects connect the execution points (events) on different agent classes with the corresponding learning components, making it transparent to the agent's basic functionality the

particularities of the learning algorithms in use. These aspects are able to crosscut some agent execution points in order to change their normal execution and invoke the learning components. The execution points include the change of a knowledge element, execution of actions on plans, roles, and agent types, or still some threw exception. Auxiliary classes are used to implement different learning techniques. This learning experience is indirect because the agent will build its knowledge through the results of the negotiations. Machine learning is used to address the knowledge acquisition. Distinct learning techniques are used in the EC system: Temporal Difference Learning (TD-Learning) [41] and Least Mean Squares (LMS) [41]. TD-Learning is used by the reviewer role in order to learn the user preferences in the subjects he/she likes to review. LMS is used by the chair to learn about the reviewer preferences. Note that the scattering of the learning concern presented in Fig. 1 is overcome in the aspect-oriented solution (Fig. 6).

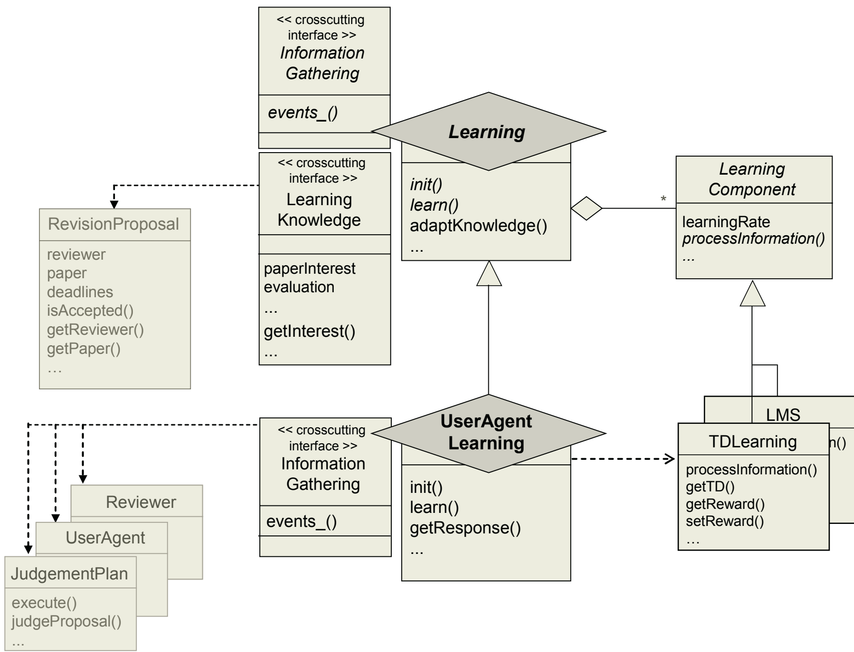


Fig. 6. The Detailed Design of the Learning Component.

5.3 The Implementation Stage

The implementation of the EC system was based on version 1.3 of the AspectJ language [10]. The work [5] presents in detail implementation issues and sample code. The EC implementation also used the JADE framework [15] and a blackboard architecture to support the communication among agents. The integration of the aspect-oriented implementation with those infrastructures was almost straightforward. However, the agent architectures could be also implemented using other aspect-oriented frameworks, such as AspectWerkz and JBoss. Although those frameworks support dynamic weaving, they incorporate constructs similar to AspectJ.

However, some inter-aspect conflicts needed to be solved. For example, the Adaptation and Autonomy aspects have pointcuts defined for the same join point: the executions of the method receiveMsg(). The AspectJ construct declare precedence has been used to specify the order of execution between these aspects. Regarding the interaction concern, there were several join points where messages should be sent to other agents. The join points include methods on plan classes and role aspects. The declaration of all those methods in the pointcut outgoingMsg is time-consuming. In order to facilitate the specification of pointcuts, such methods have been named with the prefix "prepare". The definition of the pointcuts used a simple wildcard prepare* to capture all those methods.

6 Lessons Learned

Three prototypes were built based on our proposed approach and using the AspectJ programming language: (i) a multi-agent system for traffic management [5, 25], (ii) the EC system [5], and (iii) a multi-agent system to manage a development environment for web portals [4, 6, 7]. These systems involved both reactive and cognitive agents with different combinations between the agent concerns. This section presents lessons learned on the design and implementation of aspect-oriented agent architectures, and on empirical assessments [4, 5, 6].

Inseparable Concerns. The Interaction aspects do not modularize the message assembling from different plans or roles; the message needs to be prepared within a method on plan classes or on role aspects because its assembling is very coupled to the role or plan context. One solution would be to separate the message assembling with aspects, but it would result in higher complexity.

Repetitive and Time-Consuming Definitions. All the message senders of the system must be specified in the pointcut inside the Interaction aspect. This might indeed be repetitive and tedious, suggesting that AspectJ should have more powerful metaprogramming constructs. However, this is not an unsolvable problem because code-generation tools can assist MAS engineers in this development step. In addition, we can establish a naming convention and use wildcards supported by most aspect-oriented languages. The implementation of the EC system used naming conventions.

Required Refactoring. In some circumstances, refactoring of the already defined aspects or classes may be needed as the system development evolves. For instance, the realization of the Autonomy architectural aspect requires restructuring of the base code associated with other agent components in order to expose suitable join points. For instance, we need to enforce that each method which asks for the user confirmation (when an agent decision is taken) returns a boolean value. This allows the aspect to capture the user response and control the agent autonomy degree. In addition, we have extracted code from existing methods into a new method to expose a method-level join point. Tools to help with the restructuring would make it easier to introduce aspects into an existing system.

Complex Structure for Simple Agents. Some simple reactive agents do not require thread control, react only to few events, make very simple decisions, and do not have proactive behavior. In this case, the autonomy code tends to be localized in fewer

methods. The use of aspects in this specific situation can increase rather than decrease the agent complexity.

Iterative Process. During our case studies [4, 5, 7, 22, 25], we have tried to "incrementally" deal with agent concerns at the architecture and design stages, following the order prescribed in Section 4. We have found that, as the MASs increases in complexity, the boundary between increments is not as transparent as implied. For example, the design and implementation of the mobility aspects required the creation of new pointcuts in the interaction aspects previously defined. In this way, we mostly had to follow an iterative process rather than an incremental approach in order to implement the aspect-oriented agent architectures.

Empirical Evidence. A systematic evaluation has been carried out to assess the proposed aspect-oriented approach with respect to relevant quantitative criteria [4]. We have compared quantitatively our architectural approach with a mediator-based architecture [4, 5] using a metric-based assessment framework [8]. The tallies of lines of code and number of attributes for the developed MAS in the mediator-based implementation were respectively 12% and 9% higher than in the aspect-oriented code. The aspect-oriented project also produced better results in terms of complexity of operations (6%), component couplings (9%), and component cohesion (3%). The complete description of the data gathered in this experiment can be found in [4].

7 Related Work

Dealing with several agent concerns, such as adaptation and learning, at the phase of architecture definition has been recognized as a serious problem that has not received enough attention [11, 12, 14]. In fact, related work in this area has been scarce, making no attempt in considering agent concerns within the architectural stage. Research in agent-oriented software engineering has concentrated on high-level methodologies and modeling languages [17]. Our previous work [6] dealt with crosscutting agent concerns, but it was a initial version of our approach and was focused on the detailed design and implementation levels.

Section 3 presented the existing architectural approaches for the separation of agent concerns. Software architects want to separate the application concerns in separate components, but the existing architectural styles are not able to address this separation in multi-agent systems. Our proposed agent architecture is different from a mediator-based architecture because the composition of agent concerns is not centralized in a single component, the mediator. Each architectural aspect specifies how it affects the other architectural components. The proposed architecture is not a reflective architecture since architectural aspects, unlike meta-objects, are not limited to be attached to a single object. In addition, architectural aspects can change the interface of other components by introducing fields and methods to them. Finally, the aspect-oriented agent architecture is also different from layered agent architectures defined in Kendall's approach [26]. The architectural aspects are not structured as layers; each architectural component can be associated with more than two components.

The proposed approach describes a set of architectural decisions, which contribute to improve the maintainability of MASs. The achieved segregation limits significantly

the impact of a change since the architectural components modularize the crosscutting agent concerns. Unlike the use of mediator-based agent architectures, the use of aspect-oriented architectures support the functional encapsulation of the agent's basic functionality since the Kernel component is not intermingled with agent properties. The crosscutting interfaces allow the addition of agenthood and additional properties to the basic functionality in a way that is not intrusive. As a consequence, the architectures of existing objects can be transformed into agent architectures without any changes to their methods.

The aspect-oriented architecture also improves the chances for reuse of the agent components. Applications that adopt this architecture can reuse and refine the architectural components in a more modular way since the crosscutting agent concerns are encapsulated in aspects. The agent concerns are not scattered and tangled up with each other. The improved separation of concerns facilitates also the construction of heterogeneous agent architectures. Each architectural component is oblivious in how it is modified by agent aspects. There is no reference in the Kernel component to the agent aspects. As a consequence, it is easier add or remove aspects from the agent architecture.

8 Conclusions and Ongoing Work

This paper presented an aspect-oriented stepwise approach meant to be simple enough to be used in the development of reusable and maintainable agent architectures in different kinds of agent-oriented systems. The approach follows an aspect-oriented agent architecture [5], which is a high-level description of the agents' internal structure in terms of architectural aspects and their relationships. The proposed aspect-oriented approach supports: (i) the modularization of crosscutting agent concerns from the architectural definition to the system implementation, (ii) the flexible integration of the agent concerns, and (iii) the independence of programming languages or MAS implementation frameworks. Since the aspect-oriented method is independent of programming language or implementation framework, a wide range of application developers can employ it.

The use of the aspect-oriented architecture can minimize the complexity caused by the crosscutting nature of agent properties. It proposes the use of aspects to provide a clear separation of concerns between the agent's basic functionality and the crosscutting agent properties. Moreover, aspect-oriented architectures allow that the agent properties to be incorporated into an object-oriented system when the developers want to transform their predefined objects into agents. The incorporation of agent properties can be made by attaching the corresponding agent aspects to the existing non-agent objects. However, some refactoring of the predefined objects may be required to expose the suitable join points, as discussed in Section 6.1.

We have also worked on the definition of a pattern language to support the detailed design of aspect-oriented agent architectures [5]. Each pattern in the language provides an aspect-oriented design solution for a specific crosscutting agent concern, such as learning [22] and mobility [18]. As future work, we are planning to study whether crosscutting agent concerns need to be managed at the requirements-level and if so how to support this management.

Acknowledgements

This work has been partially supported by CNPq under grants No. 141457/2000-7 and No. 381724/2004-2 for Alessandro, grant No. 140252/2003-7 for Uirá, and by FAPERJ under grant No. E-26/150.699/2002 for Alessandro. The authors are also supported by the PRONEX Project under grant 7697102900, and by ESSMA under grant 552068/2002-0.

References

1. F. Buschmann et al. Pattern-Oriented Software Architecture: A System of Patterns. John Wiley Sons, 1996.
2. C. Chavez. A Model-Driven Approach to Aspect-Oriented Design. PhD Thesis, Computer Science Department, PUC-Rio, April 2004, Rio de Janeiro, Brazil.
3. A. Garcia, M. Cortés, C. Lucena. A Web Environment for the Development of E-Commerce Portals. Proceedings of the IRMA'01, Toronto, May 2001.
4. A. Garcia et al. Separation of Concerns in Multi-Agent Systems: An Empirical Study. In: "Software Engineering for Multi-Agent Systems II", Springer, LNCS 2940, April 2004.
5. A. Garcia. From Objects to Agents: An Aspect-Oriented Approach. PhD Thesis, Computer Science Department, PUC-Rio, April 2004, Rio de Janeiro, Brazil.
6. A. Garcia, C. Lucena, D. Cowan. Agents in Object-Oriented Software Engineering. Software: Practice and Experience, Volume 34, Issue 5, April 2004, pp. 489-521.
7. A. Garcia et al. Engineering Multi-Agent Systems with Aspects and Patterns. Journal of the Brazilian Computer Society, Number 1, Volume 8, July 2002, pp. 57-72.
8. C. Sant'anna, A. Garcia, C. Chavez, C. Lucena, A. Staa. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. Proc. 17th Brazilian Symposium on Software Engineering (SBES'03), Manaus, Brazil, October 2003.
9. G.Kiczales, et al. Aspect-Oriented Programming. Proc.ECOOP'97, LNCS 1241, June 1997.
10. G. Kiczales et al. Getting Started with AspectJ. CACM, October 2001.
11. A.Pace et al. Architecting the Design of Multi-Agent Organizations with Proto-Frameworks. In: "Software Engineering for MASs II", LNCS 2940, Feb 2004, pp. 75-92.
12. A. Pace et al. Assisting the Development of Aspect-based MAS using the SmartWeaver Approach. In: "Software Engineering for Large-Scale MASs", LNCS 2603, March 2003.
13. 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.
14. N. Ubayashi, T. Tamai. Separation of Concerns in Mobile Agent Applications. Proc. of the 3rd Conference Reflection 2001, LNCS 2192, Kyoto, September 2001, pp. 89-109.
15. F. Bellifemine et al. JADE: A FIPA-Compliant Agent Framework. Proc. of the Practical Applications of Intelligent Agents and Multi-Agents, pp. 97-108, April 1999.
16. A. Fuggetta, G. Picco, C. Vigna. Understanding Code Mobility. IEEE Transactions on Software Engineering, vol.24, No.5, pp.342-361, 1998.
17. C. Iglesias, et al. A Survey of Agent-Oriented Methodologies. Proceedings of the ATAL-98, Paris, France, July 1998, pp. 317-330.
18. A. Garcia et al. The Mobility Aspect Pattern. Proc. of the 4th Latin-American Conference on Pattern Languages of Programming, SugarLoafPLoP'04. August, 2004, Fortaleza, Brazil.
19. A. Amandi, A. Price. Building Object-Agents from a Software Meta-Architecture. In: Advances in Artificial Intelligence, LNAI, vol. 1515, Springer-Verlag, 1998.
20. E. Gamma et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, 1995.

21. D. Camacho. Coordination of Planning Agents to Solve Problems in the Web. AI Communications, IOS Press, Vol. 16 (4), November, 2003, pp. 309-311.
22. A. Garcia et al. The Learning Aspect Pattern. Proc. of the 11th Conference on Pattern Languages of Programs (PLoP2004), September 2004, Monticello, USA.
23. E. Pulvermüller, A. Speck, A. Rashid. Implementing collaboration-based Designs using Aspect-Oriented Programming. Proc. of TOOLS-USA, 2000, p. 95 - 104, Jul 2000.
24. R. Lavender, D. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. In: “Pattern Languages of Program Design”, Addison-Wesley, 1996.
25. A. Costa. An Aspect-Oriented Software Architecture for Traffic Simulators. Master’s Dissertation, University of Sao Paulo, October 2003. (In Portuguese)
26. E. Kendall et al. A Framework for Agent Systems. Implementing Application Frameworks – OO Frameworks at Work, M. Fayad et al. (ed). John Wiley & Sons: 1999.
27. E. Kendall. Role Model Designs and Implementations with Aspect-oriented Programming. Proceedings of OOPSLA’99, ACM Press, 1999, pp. 353-369.
28. M. D'Hondt, K. Gybels, V. Jonckers. Seamless Integration of Rule-Based Knowledge and Object-Oriented Functionality with Linguistic Symbiosis. Proceedings of the 19th Annual ACM Symposium on Applied Computing (SAC 2004), Nicosia, Cyprus, March 2004.
29. E. Dijkstra. A Discipline of Programming. Prentice Hall, Englewood Cliffs, NJ, 1976.
30. N. Jennings. Agent-Oriented Software Engineering. Proc. of the 12th Intl. Conference on Industrial and Engineering Applications of Artificial Intelligence, 1999, pp. 4-10.
31. M. Huhns, M. Singh (Eds.). Agents and Multiagent Systems: Themes, Approaches, and Challenges. Readings in Agents, Chapter 1, Morgan Kaufmann Publishers, USA, pp. 1-23.
32. D. Rasmus. Rethinking Smart Objects: Building Artificial Intelligence with Objects. Cambridge University Press, New York, 1999.
33. J. Briot, L. Gasser. Agents and Concurrent Objects. IEEE Concurrency, Special Issue on Actors and Agents, 1998.
34. A. Rao, M. Georgeff. BDI Agents: From Theory to Practice. Proceedings of the 1st Intl. Conference on Multi-Agent Systems (ICMAS-95), San Francisco, 1995; 312-319.
35. Shoham, Y. Agent-Oriented Programming. Artificial Intelligence, 60(1):51-92, Mar 1993.
36. FIPA, Agent Communication Technical Committee. Agent Communication Language - FIPA'99 Draft Specification, 1999. http://www.fipa.org.
37. S. Splunter, N. Wijngaards, F. Brazier. Structuring Agents for Adaptation. In: E. Alonso et al (Eds), Adaptive Agents and Multi-Agent Systems, LNAI, Vol. 2636, 2003, pp. 174-186.
38. S.Russell, P.Norvig. Artificial Intelligence: A Modern Approach.Prentice Hall, 2 ed, 2002.
39. T. Norman, D. Long. Goal Creation in Motivated Agents. In: Wooldridge, Jennings (Eds.), Intelligent Agents: Theories, Architectures, and Languages, LNAI 890: Springer, 1995.
40. B. Ekdahl. How Autonomous is an Autonomous Agent? Proc. of the 5th Conference on Systemic, Cybernetics and Informatics (SCI 2001), July 22-25, 2001, Orlando, USA.
41. T. Mitchell. Machine Learning. McGraw Hill, New York, 1997.
42. P. Tarr, H. Ossher. Hyper/J User Manual, 2000. www.alphaworks.ibm.com/tech/hyperj
43. F.Zambonelli,N.Jennings,M.Wooldridge. Organizational Abstractions for the Analysis and Design of Multi-agent Systems. In:“Agent-Oriented Software Engineering”, Springer, 2001.
44. Z. Guessoum, J. Briot. From Active Objects to Autonomous Agents. IEEE Concurrency, Special Series on Actors and Agents, Vol. 7, N. 3, 1999, pp. 68-76.