

Taming Heterogeneous Agent Architectures with Aspects

Alessandro Garcia

*Computing Department
InfoLab 21, South Drive
Lancaster University
LA1 4WA
Lancaster – United Kingdom
e-mail: garciaa@comp.lancs.ac.uk*

Carlos Lucena

*Computer Science Department
Pontifical Catholic University of Rio de Janeiro
Rua Marquês de São Vicente, 225 - 22453-900
Edifício Pe. Leonel Franca, 10º. andar
Rio de Janeiro – Brazil
e-mail: lucena@inf.puc-rio.br*

From Objects to Agents: New Driving Architectural Concerns

The recent advances in network-based software applications and the advent of ubiquitous computing are pushing us inevitably towards a world of autonomous software architectures. This trend has spurred the revitalization of agent technology as a complement to the object paradigm for a variety of modern application domains, including e-commerce, software development environments, and personal assistants. There is explicit evidence to believe that the penetration of software agents is also high for systems used in military and government contexts [2, 8]. Agents, like objects, provide services to their clients, but are recognizably different from objects as seen from an architectural point of view [3, 5, 8]. Unlike objects, an agent is an autonomous entity that takes the initiative to achieve system goals and represent software users [2, 3].

As a result, architects of software agents are faced with basic concerns, such as the agent services that are made available to the clients, and a number of additional concerns on top of the basic concerns. The internal architecture of a single agent encompasses multiple properties, including autonomy, interaction, adaptation, collaboration, learning, and mobility. Hence architects of agent-based systems are also concerned with issues such as making an agent interact appropriately, handling the agent's adaptive behavior, structuring the agent's autonomous behavior, designing the agent roles and protocols for inter-agent collaboration purposes, and incorporating learning mechanisms into the agent's structure in a modular manner.

Not surprisingly, separation of concerns is at the core of the development of agent-based software systems [5, 12]. The reuse and maintenance of agent elements depend largely on the ability of used architectural abstractions to support the separate handling of agent-specific concerns since an early state of design. The applied architectural styles must enable the modularization of each agent concern and their proper composition, so that the achieved segregation significantly limits the impact of a change and improves the chances for architecture reuse in other software projects. This separation of concerns needs to be guaranteed throughout the different development phases, especially from the architectural to the implementation phase. The architectural separation will be lost if the implementation abstractions are not able to preserve it. In fact, the sole use of existing well-known agent platforms, such as JADE and JACK [12],

do not provide advanced mechanisms to achieve separation of concerns, which are restricted to exploit the object-oriented composition and decomposition mechanisms [5]. On the other hand, even though such advanced implementation mechanisms are delivered, the benefits of separation of agent concerns would be hindered if existing architectural abstractions do not allow the achievement of proper system modularization from the design outset.

Heterogeneity in Agent Architectures

Although separation of concerns is critical to architects of agent-based software, it is often difficult to achieve in realistic systems for several reasons. First, these systems typically encompass heterogeneous types of agents [2]. The internal architecture of distinct agent types differs widely from each other since they incorporate distinct properties [8]. Second, each property is orthogonal and interacts with the agent's basic functionality and often with other agent properties. These properties typically crosscut several modules of an agent architecture, independently from the adopted internal model, such as the constraint-oriented model or the BDI (Belief-Desire-Intention) model [10], and from the agent's cognitive level, such as reactive agents, deliberative agents, or even hybrid agents. Third, these crosscutting agent-specific concerns are related in dramatically different ways that depend on the agents' types and the application requirements [6]. For example, in a given agent-based application, the mobility behavior of an agent may only directly affect the basic functionality of the agent, while it may also crosscut the collaboration and interaction concerns in a second application.

As a consequence, agent-based applications require an architectural approach that is flexible enough to support adjustable composition of agent concerns and the construction of heterogeneous agent architectures according to the application demands. This flexibility requirement is even more stringent in open agent-based systems due to their adaptive and open nature. In these contexts, the agent architecture needs to be modular enough to support the dynamic reconfiguration of its internal elements. For example, roles potentially need to be changed as the agent moves to new environments. The degree of autonomy and the learning strategies may also need to be adapted or disabled according to the dynamic execution contexts. While part of the reconfiguration system-level facilities are typically provided by specific middleware implementations, many of the agent concerns and their composition are essentially application-dependent. Software architects need to prepare and conceive at the design outset modular agent architectures in order to cope with these heterogeneity issues.

Shortcomings of Classical Agent Architectures

Agent-oriented software engineering has been studied from different perspectives, including agent-oriented methodologies and languages for higher-level development phases, conceptual modeling, and implementation frameworks [2, 3]. Although separation of concerns is widely recognized as crucial to the development of maintainable multi-agent software [5, 6, 12], existing approaches do not scale up to

support the separation of agent properties in heterogeneous architectures. Developers have to rest on traditional architectural patterns, such as the Mediator pattern [4] and the Layers pattern [4], in order to build their systems. As illustrated in Figure 1, these solutions define architectural abstractions, such as mediators and layers, and composition rules to support the isolation of agent concerns and their further composition.

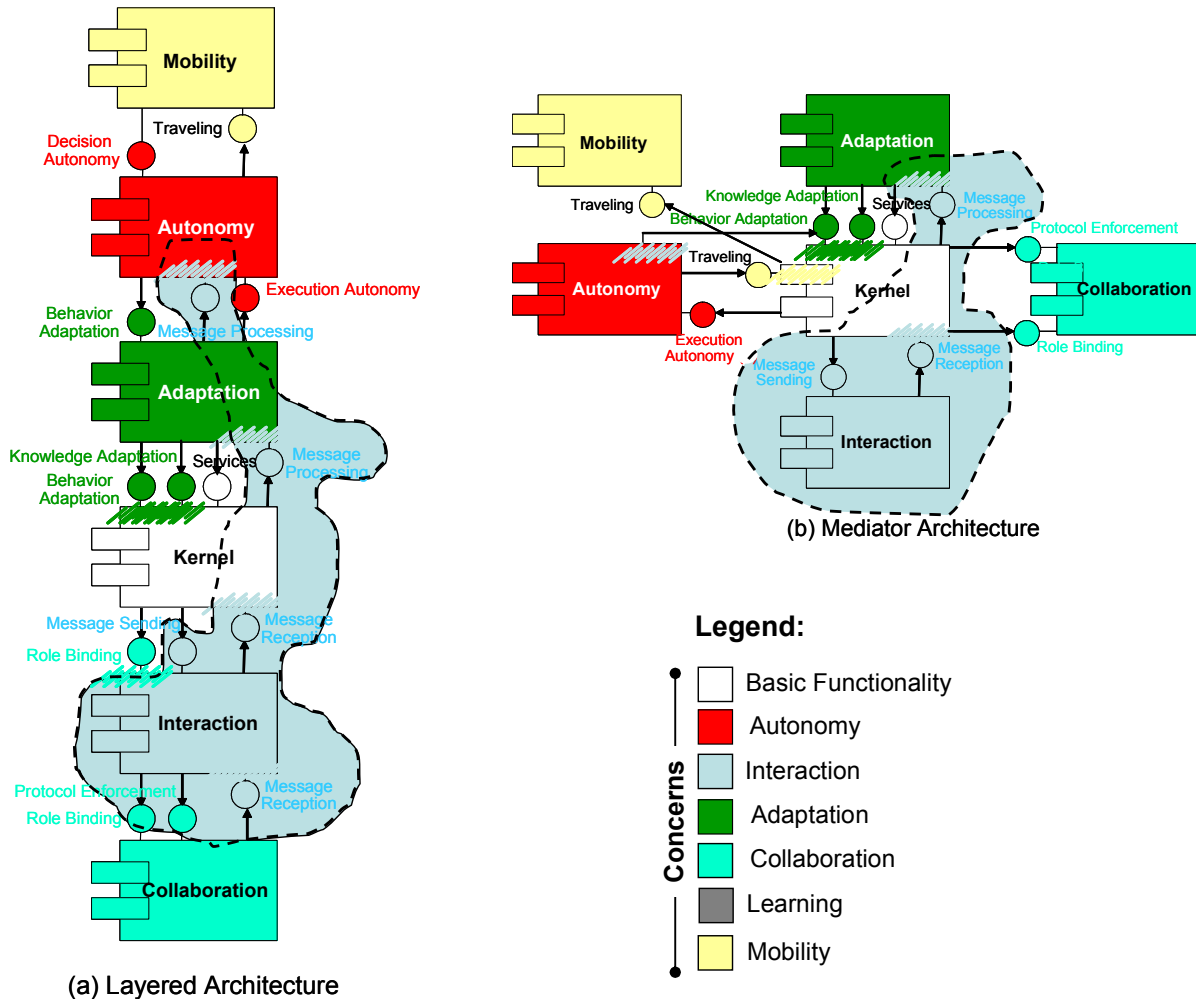


Figure 1. Crosscutting Concerns in Heterogeneous Agent Architectures

However, these solutions impose rigid connections on the architectural components, which make the construction of heterogeneous agent types difficult and not scalable to cope with the complexity of multiple interactive agent concerns. These existing architectural proposals are applicable to simple agent architectures and with few agenthood properties. The constraints on the composition rules imposed by those solutions do not scale up to master the intrinsic crosscutting character of agent properties in heterogeneous contexts. Figure 1 illustrates this problem in terms of a layered agent architecture (Fig. 1a) and a mediator-based agent architecture (Fig. 1b), represented with a simplified UML notation.

ARCHITECTURAL PROBLEM	DESCRIPTION
<i>Architectural scattering</i>	The occurrence of architectural elements, such as interfaces, that belong to one agent concern in architectural components encapsulating other agent concerns. For example, the interaction-related interfaces are scattered over the agent architectures in Fig. 1.
<i>Architectural tangling</i>	The mix of multiple agent concerns together in the same module. For instance, tangling is evident in the agent kernel of Figs. 1a and 1b as both are implementing interfaces associated with different agent concerns.
<i>Hindering of modular reasoning</i>	The inability of the architect being able to reason and make decisions about an agent architecture's module while looking only at its functionality and its interfaces. For example, the architects treating the collaboration and interaction concerns in Fig. 1a need to consult the definitions and interfaces of other modules, leading to an expanded or global reasoning rather than a modular reasoning.
<i>High architectural coupling and low cohesion</i>	The presence of an overly strong connection between architecture components (high coupling), and the lack of closeness between the internal functionalities and interfaces of a specific component (low cohesion). In Fig. 1b, for example, the system coupling is increased, and the cohesion of each component (e.g. the Kernel) is decreased.
<i>Architectural interface bloat</i>	The evidence of complexity increase in the component interfaces. For example, the number of interfaces of several components in Figs 1a and 1b is augmented due to the crosscutting nature of the interaction concern.
<i>Poor traceability from requirements to architecture</i>	The inability of directly mapping in the architecture description the separation of agent concerns achieved in the requirements description. For example, the mobility-specific requirement cannot be directly traced to the Mobility component (Fig. 1b) as it also affects the boundaries of the Kernel component.
<i>Poor modularization of design decisions and rationale</i>	The incapacity of the architectural decisions and rationale associated with one agent concern being documented in a localized manner. The rationale and design decisions related to the interaction concern are typically tangled and scattered over the architecture artifacts in the same way as interaction-specific interfaces and functionalities.
<i>Inflexible design</i>	The rigidity in the composition rules to support alternative compositions between agent components in order to smoothly produce heterogeneous architectures. This inflexibility is visible both in the mediator and layered solutions.
<i>Limited evolvability and maintainability</i>	The adversities associated with including, changing or removing the design elements associated with an agent crosscutting concern as they crosscut several architectural components. For instance, this problem would occur recurrently in changing or replacing the collaboration and interaction components in Fig. 1a. In fact, in a previous study [6] we have observed a poor stability of a typical mediator-based agent architecture implementation in the presence of maintenance and evolution scenarios.

Table 1. Shortcomings of Existing Architectural Styles

The layered architecture imposes a bi-directional communication only between adjacent layers; i.e. the architectural components of a software agent. In order to make proper internal decisions and adapt the agent knowledge accordingly, the autonomy and adaptation components need to be aware of received messages and external stimulus coming from the surrounding environment. As illustrated in Figure 1a, this requirement forces the software architects to add interaction-related interfaces into the other layers so that the autonomy and adaptation components have access to the perceived stimulus and messages. As a result, the interaction concern (represented in blue) crosscuts the modularity of the kernel and adaptation components.

The situation is not different in the mediator-based solution. This architectural style requires all the inter-component communications being intermediated by a central component, which is also in charge of encapsulating the agent's basic functionality. As illustrated in Figure 1b, this architectural approach also fails to isolate the interaction concern and leads to the intermingling of the agent kernel functionality and the message-processing functionality. Note that a similar problem occurs with the mobility concern as the communication with the autonomy component needs to be settled by the agent kernel. Hence, the inability of traditional architectural styles and respective composition rules in capturing the crosscutting feature of agent concerns in heterogeneous architectures leads to several negative consequences, such as architectural tangling and bloated interfaces, which are described in detail in Table 1.

Spreading the Crosscutting to Subsequent Artifacts

The problems associated with heterogeneous agent architectures are typically disseminated from the agent architecture specifications to artifacts generated later in the software lifecycle. Hence, the crosscutting of agent concerns is inevitably spanned over the resulting artifacts in the detailed design level and implementation. It does not matter what kind of decomposition and abstractions the agent-based software developers are relying on; the problem is the same if you are using an agent-oriented design language [2], or an OO modeling language, such as UML.

For example, the architectural abstractions and composition rules are not directly supported in OO design and programming languages. They are not aligned with the composition and decomposition mechanisms of the object paradigm, which makes it hard to handle the heterogeneous agent types in a single system [6]. For instance, the inheritance mechanism usually leads to large inheritance trees with replication of code and an explosion of the number of classes [5, 6, 12]. Even in some simple agent architectures, bringing them into the detailed design and code also raises similar problems (Table 1), such as scattering and tangling.

Fig. 2a illustrates how the reification of heterogeneous agent architectures in the design tends to be scattered over many classes of the system design. It shows a partial representation of an agent-based system [6] composed of three agent types, each with a different internal architecture. Each set of classes, surrounded by a gray rectangle, has the main purpose of modularizing a specific agent concern, namely the agents' basic functionality and collaboration. The figure shows that agent-specific concerns crosscut

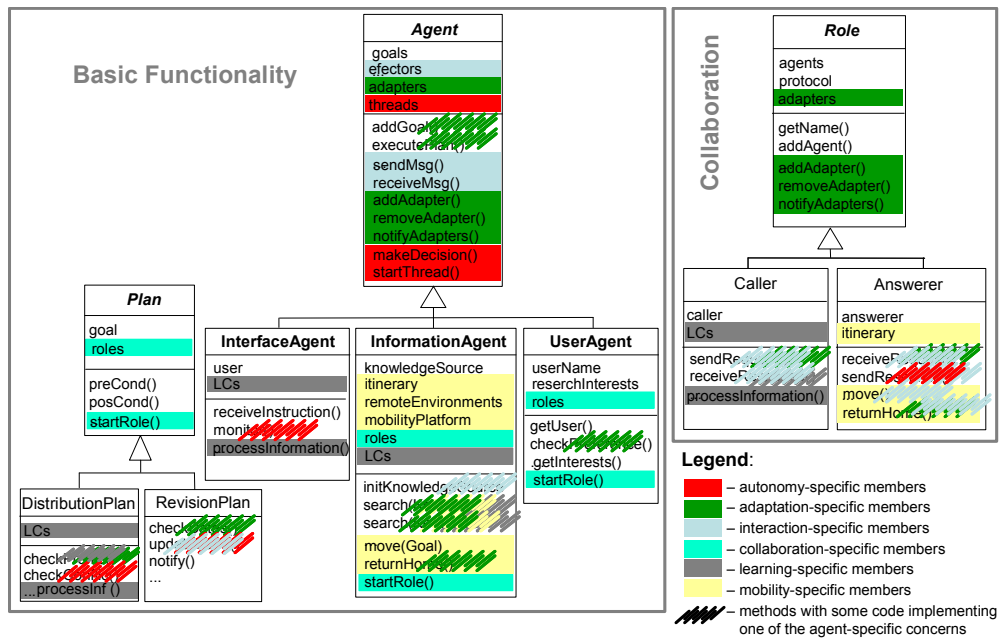
the classes implementing those surrounded classes, such as “Information Agent” and “Role”. The use of OO mechanisms breaks not only the class-level modularity, but also the operation-level modularity. Fig. 2b shows the partial implementation of a searcher agent, which consists of a class and respective operations with a confusing tangle of lines of code for different concerns. On the left, there is the code for a class as a non-agent entity. On the right, there is an equivalent class with code for implementing agent-specific concerns. After analyzing the code on the right, it is clear that such code has lost the functional encapsulation of the non-agent version (left side). Moreover, it is a confusing intermingling of lines of code for different agent concerns, and the modularization of several agent-specific concerns is lost across the system classes.

Architecting Software Agents with Aspects

Aspect-oriented software development [9] is an evolving paradigm to modularize concerns which existing software engineering abstractions and mechanisms are not able to capture explicitly. The notion of aspects encourages modular descriptions of complex software by providing support through new composition mechanisms for cleanly separating the system functionality from its crosscutting concerns. However, existing aspect-oriented approaches have not been explored in the context of heterogeneous agent architectures; they have focused on the context of classical crosscutting concerns, such as distribution [11], persistence [11], and design patterns [7]. These concerns have been mostly studied in an isolated way and from the implementation point of view.

Aspects and its new composition possibilities can be exploited at the architectural level to capture the multiple interacting agent concerns that are hard to modularize with existing architectural abstractions. An aspect-oriented architectural style brings to the software architects a new abstraction, the notion of *architectural aspects*, and new composition means for handling each crosscutting agent concern as an individual component at an early stage of design, as illustrated in Fig. 3a. Each architectural aspect modularizes a typical crosscutting agent property and separates it from the agent kernel. The “aspectization” of agent architectures allows the association of agenthood properties with the basic functionality in a way that is transparent to the agent kernel.

The key idea to enable adjustable compositions is the notion of *crosscutting interfaces*, which are modularity abstractions attached to the architectural aspects. A crosscutting interface is different from a module interface in the sense that the latter essentially provides services to other components. Crosscutting interfaces provide services to the system, but also specify when and how an aspect affects other architectural components. Opposed to interfaces in traditional architecture styles, they flexibly determine which external components and interfaces the architectural aspect of a software agent will be connected. With this dependency inversion, crosscutting interfaces overcome the problems associated with the rigidity implicit in traditional architectural styles (Figure 1). Each agent’s architectural aspect can be more flexibly composed with the agent kernel and with any agent aspects depending on the requirements of a specific agent architecture.



(a) scattering in design representation

```

public class Searcher {
    private KnowledgeSource ks;
    public SearcherAgent() {
        initKnowledgeSource();
    }
    private void initKnowledgeSource() {
        ks = KnowledgeSource.getInstance();
        if (ks.disconnected) { ks.connect(); }
    }
    public String search(String keyword) {
        Enumeration tuples = ks.elements();
        boolean foundKeyword = false;
        String tuple = new String();
        while ((!foundKeyword) &&
            (tuples.hasMoreElements())) {
            tuple = (String)tuples.nextElement();
            int response = tuple.indexOf(keyword);
            if (response != -1) {foundKeyword = true;}
        }
        if (foundKeyword) {return tuple;}
        else {return "Keyword not found!";}
    }
}

public class InformationAgent extends Agent {
    private Hashtable itinerary;
    private Hashtable remoteEnvironments;
    private Hashtable mobilityPlatform;
    private KnowledgeSource ks;
    private Hashtable roles;
    protected Learning learningComponent;
    public SearcherAgent(Goal initialGoal) {
        initKnowledgeSource();
        goals = new Vector(); plans = new Vector();
        inbox = new Vector(); outbox = new Vector();
        interactionComponent = new Interaction(this);
        autonomyComponent = new Autonomy(this);
        adaptationComponent = new Adaptation(this);
        learningComponent = new Learning(this);
        setGoal(initialGoal);
    }
    private void initKnowledgeSource() {
        ks = KnowledgeSource.getInstance();
        if (ks.disconnected) { ks.connect(); }
        Message msg = new Message("Manager", "Searcher is Available");
        sendMsg(msg);
    }
    public String search(String keyword) {
        learningComponent.processPreference(keyword);
        Enumeration tuples = ks.elements();
        boolean foundKeyword = false;
        String tuple = new String();
        while ((!foundKeyword) && (tuples.hasMoreElements())) {
            tuple = (String)tuples.nextElement();
            int response = tuple.indexOf(keyword);
            if (response != -1) {foundKeyword = true;}
        }
        learningComponent.processQuerySuccess(foundKeyword);
        if (foundKeyword) {return tuple;}
        else {
            InformationSearchGoal goal = new InformationSearchGoal(...);
            setGoal(goal);
            move(goal);
            return goal.getInfo();
        }
    }
    protected boolean move(Goal goal) {
        ...
    }
    protected boolean returnHome() {
        ...
    }
    protected boolean startRole(Role role) {
        ...
    }
}

```

- Legend:**
- Basic Functionality
 - Autonomy
 - Interaction
 - Adaptation
 - Collaboration
 - Learning
 - Mobility

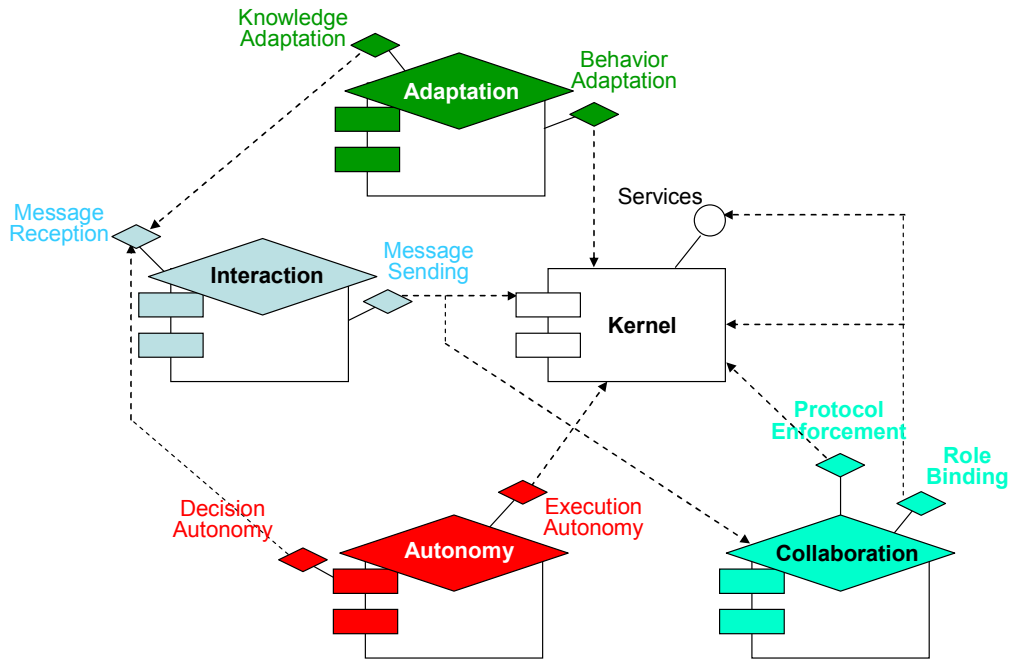
(b) tangling in a class representing an agent

Figure 2. Heterogeneous agent architectures: crosscutting the lifecycle artifacts

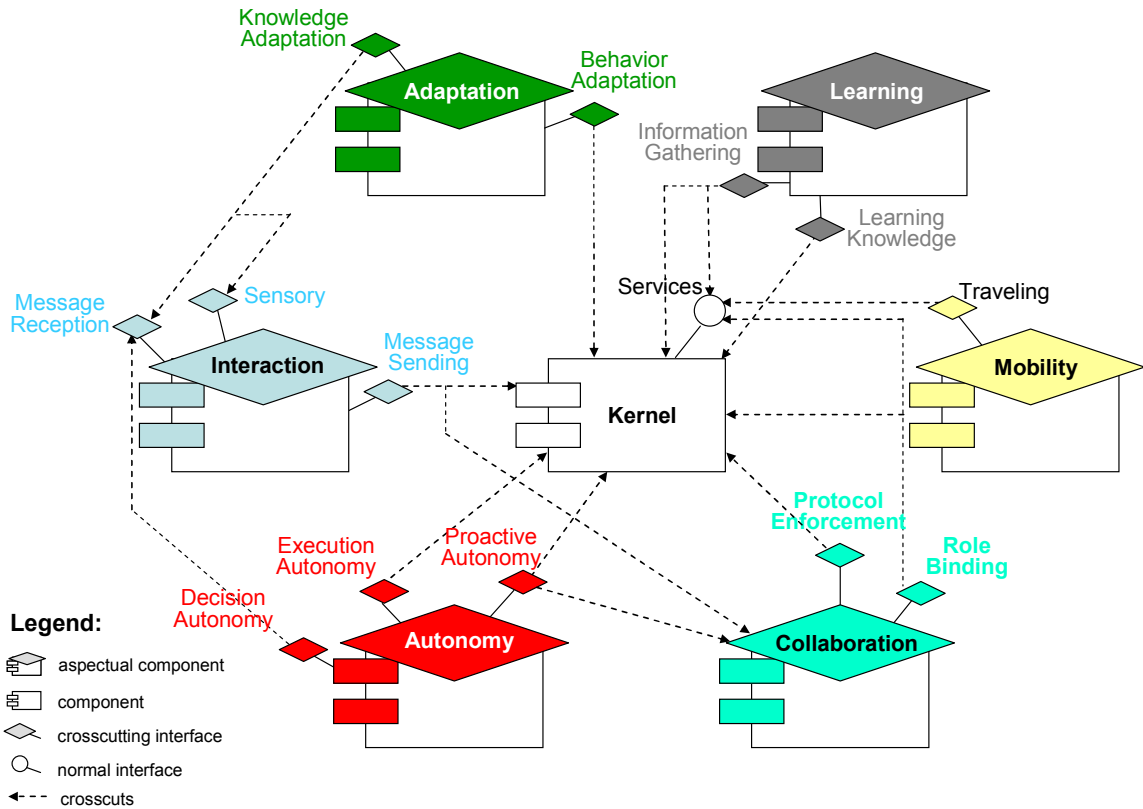
Each of the architectural aspects is related to more than one component, representing the crosscutting nature of agent properties in complex architectures. An agent's architectural aspect can realize more than one crosscutting interface since it can crosscut multiple agent components in different ways. The interface of an architectural aspect can crosscut the Kernel component and other architectural aspects. An aspect interface crosscuts either the internals 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 an agent component. The second case means that the aspect affects elements of an interface.

Figure 3 illustrates instances of aspect-oriented agent architectures for a user agent (3a) and an information agent (3b). Extensions of UML are used to represent the architectural aspects (components with diamonds on their top), crosscutting relationships (dotted arrow), and crosscutting interfaces (small diamonds). Their partial non-aspect-oriented detailed designs were represented in Figure 2a. The user agent is prominently reactive, while the information agent is a proactive entity. As a result, their architectures are very different. For example, the component in charge of implementing the autonomous behavior has different interfaces and relationships with each component in the agent architecture. Reactive agents make decisions only as a matter of deciding about external requests embedded in incoming messages. Proactive agents also need to inspect changes in their internal state and decide for starting new actions by its own initiative independently from requests from other agents and system users. In other words, the autonomy component also needs to observe events in the agent kernel and in the agent roles (collaboration component) in order to make decisions.

Figure 3 shows that the aspect-oriented style easily supports both architectural configurations. A partial architectural design is shown and some non-aspectual components, such as the ones for agent knowledge representation and message assembling, are left out for simplicity purposes. In the first case (Figure 3a), the autonomy aspect does not have an interface for enabling proactive behavior. It has two crosscutting interfaces which affect the kernel and interaction components. In the second case (Figure 3b), the autonomy component includes a third crosscutting interface which supports the observation of the agent kernel and the collaboration component in order to trigger the proactive behavior of the information agent. Notice that the architectures have other important differences in their configurations and compositions. The architecture of the information agent also includes the mobility and learning aspects and a third interface to enable sensory behavior. Moreover, the adaptation aspect crosscuts the sensory interface in order to adapt as new environmentally relevant events are sensed by the agent.



(a) aspect-oriented architecture of a user agent



(b) aspect-oriented architecture of an information agent

Figure 3. Aspectizing heterogeneous agent architectures: enabling flexible composition

What are Other Benefits?

The benefits of having new architectural design rules entailed by an aspect-oriented style are not limited to improved composability of agent architecture concerns. We have noticed a number of other positive consequences such as the ones discussed in the following.

Modular reasoning at the architectural level. The aspectization of agent architectures supports modular reasoning, since the architects are able to more independently treat each agenceness property. Different from other architectural styles, the agent concerns are modularized and do not affect the definitions and interfaces of multiple architectural modules. The notion of crosscutting interfaces allows to address tangling- and scattering-related problems typically found in the definition of heterogeneous agent architectures. As a result, software architects are able to make decisions about an agent property while only looking at its description and its interface with other concerns.

Smooth transition in software lifecycle phases. Aspect-oriented agent architectures are directly mapped to implementation abstractions using well-known aspect-oriented programming languages, such as AspectJ [1]. AspectJ is the most popular aspect language, which extends the Java programming language. Architectural aspects are decomposed into a set of AspectJ aspects and classes. The crosscutting interfaces are realized as pointcuts, advice, and inter-type declarations, because they define different ways an aspect affects other design and implementation modules. 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 a 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. We have implemented an AspectJ framework that supports this aspect-oriented architectural style at the implementation level [5]. It helps to guarantee a smooth transition from the specification of heterogeneous agent architectures to their detailed design and implementation. We have also developed several other aspect-oriented techniques to cope with agent aspects in different software development phases in order to facilitate the traceability of the software engineering artifacts.

Coping with Dynamic Adaptability and Customizability. Architecting software agents with improved separation of concerns is of paramount importance in open, dynamic agent-based systems. Dynamic reconfiguration of agent roles and collaboration protocols are often required as the agents move to different environments. In addition, with the growing number of applications for pervasive computing, the selection of learning and coordination strategies may depend on the context where the agent is being executed. As such, agent architectures need to be designed properly, and the dependency inversion gained with aspect-oriented agent architectures is a key factor to allow the agent adaptation and customization.

Conclusions

Nowadays numerous types of agent architectures are everywhere [8] and need to be developed in a way that meets the stringent modern requirements of evolvability, reusability, and dynamic reconfigurability. Existing architectural styles are rigid by its very nature and unable to cope with the crosscutting nature of agent properties as well as the complexity of heterogeneous agent architectures, which are often required in realistic modern systems. On the other hand, aspect-oriented software development is gaining wide attention both in research environments and in industry as a paradigm to promote improved modularity of complex software systems. The exploration of aspect-oriented techniques clearly seems a promising step forward to allow the construction of more flexible agent architectures and to foster enhanced quality of realistic multi-agent systems.

References

- [1] ASPECTJ Website. The AspectJ Guide. <http://eclipse.org/aspectj/>.
- [2] BERGENTI, F.; GLEIZES, M.-P.; ZAMBONELLI, F. (Eds.). Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook. Springer-Verlag, Vol. 11, 2004.
- [3] BORDINI, R.; DASTANI, M.; DIX, J.; SEGHROUCHNI, A. (Eds.) Multi-Agent Programming: Languages, Platforms, and Applications. Springer, New York, USA, 2005.
- [4] BUSCHMANN, F. et al. Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons, 1996.
- [5] GARCIA, A.; LUCENA, C.; COWAN, D. Agents in Object-Oriented Software Engineering. Software: Practice and Experience, Elsevier, Volume 34, Issue 5, April 2004, pp. 489 - 521.
- [6] GARCIA, A.; SANT'ANNA, C.; CHAVEZ, C.; SILVA, V.; LUCENA, C.; STAA, A. Separation of Concerns in Multi-Agent Systems: An Empirical Study. In: C. Lucena et al (Eds.), Software Engineering for Multi-Agent Systems II, Springer-Verlag, LNCS 2940, March 2004, pp. 49-72.
- [7] GARCIA, A.; SANT'ANNA, C.; FIGUEIREDO, E.; KULESZA, U.; LUCENA, C.; VON STAA, A. Modularizing Design Patterns with Aspects: A Quantitative Study. Proc. of 4th ACM International Conference on Aspect-Oriented Software Development (AOSD'05), Chicago, USA, 14-18 March 2005, pp. 1-11.
- [8] JENNINGS, N. An Agent-Based Approach for Building Complex Software Systems. Communications of the ACM, vol. 44, no. 4, pp. 35--41, 2001.
- [9] KICZALES, G. et al. Aspect-Oriented Programming. Proceedings of the European Conference on Object-Oriented Programming - ECOOP'97, LNCS (1241), Springer-Verlag, Finland., June 1997, pp. 220-242.

- [10] RAO, A.; GEORGEFF, M. BDI Agents: From Theory to Practice. Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS-95), V. Lesser and L. Gasser (Eds.), AAAI Press/MIT Press, San Francisco, June 1995; 312-319.
- [11] SOARES, S.; LAUREANO, E.; BORBA, P. Implementing Distribution and Persistence Aspects with AspectJ. Proc. of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02), pp. 174-190.
- [12] UBAYASHI, N.; TAMAI, T. Separation of Concerns in Mobile Agent Applications. Proceedings of the 3rd International Conference Reflection 2001, LNCS 2192, Kyoto, Japan, September 2001, Springer, pp. 89-109.