Integrating Generative and Aspect-Oriented Technologies

Uirá Kulesza, Alessandro Garcia, Carlos Lucena, Arndt von Staa
Software Engineering Laboratory, SoC+Agents Group, Computer Science Department
Pontifical Catholic University of Rio de Janeiro - PUC-Rio
e-mail: [uira, afgarcia, lucena, arndt]@inf.puc-rio.br

Abstract

Over the last years, two new software engineering approaches have been proposed: generative programming and aspect-oriented software development. Generative programming addresses the study and definition of methods and tools that enable the automatic production of system families from a high-level specification. Aspect-oriented software development has been proposed as a technique for improving separation of concerns in the construction of OO software and supporting improved reusability and ease of evolution. The use of aspect-oriented techniques in a definition of a generative approach can bring benefits to the modeling and generation of crosscutting features since early development stages. This paper presents our experience in the definition of an aspect-oriented generative approach. The proposed approach explores the multi-agent systems domain to enable the code generation of agent architectures.

1. Introduction

Over the last years, generative programming and aspect-oriented software development have been proposed aiming at increasing maintainability and reusability of software systems. While several research works have focused on the investigation of the individual use of each of these software engineering approaches, less attention has been paid to the integration of these two techniques.

Generative Programming (GP) [8] has been proposed recently as an approach based on domain engineering [21, 27, 28]. It addresses the study and definition of methods and tools to enable the automatic production of software from a high-level specification. GP promotes the separation of problem and solution spaces, giving flexibility to evolve both independently. Problem space models concepts and features existent in a specific domain. Solution space consists of the components that are used to build particular software systems. Code generators represent the configuration knowledge in a generative model. They define how specific feature combinations in the problem space are mapped to a set of software components in the solution space.

Aspect-Oriented Software Development (AOSD) [23, 33] is an evolving approach to modularize crosscutting concerns that existing paradigms (e.g.: object-oriented) are not able to capture explicitly. Crosscutting concerns are concerns that often crosscut several modules in a software system. AOSD encourages modular descriptions of complex software by providing support for cleanly separating the basic system functionality from its crosscutting concerns. Aspect is the abstraction used to modularize the crosscutting concerns.

The use of aspect-oriented techniques in the definition of a generative approach can bring additional benefits for the development of system families, such as: (i) clear separation of orthogonal and crosscutting features in the problem and solution space; and (ii) direct mapping of crosscutting features in aspects. Despite these advantages, we believe that the integration of GP and AOSD techniques is not a trivial task. Interesting questions arise and need to be considered when

developing an aspect-oriented generative approach, including: How to model crosscutting features in the problem space? How to design aspect-oriented architectures that address the crosscuting and non-crosscutting features modeled? Which technologies (domain-specific languages, frameworks) are appropriate to implement these aspect-oriented generative approaches?
Recent work explored the use of GP and AOSD together [18, 26, 31]. However, these reports neither cover nor describe in detail the typical phases (domain analysis, domain design and domain implementation) found in the definition of a generative approach.
In this context, this paper describes systematically how we have developed an aspect-oriented generative approach to the context of families of multi-agent systems. Following the guidelines presented by Czarnecki and Eisenecker [8], we have organized the development of the generative approach into three phases: (i) domain analysis; (ii) domain design; and (iii) domain implementation. The use of aspect-oriented technologies required the adaptation of modeling notations used in domain analysis and design, such as: (i) the extension of feature models to represent crosscutting features; and (ii) the extension of a current aspect-oriented modeling notation [6] to represent aspect-oriented architectures. In the domain implementation, we illustrate the use of different mainstream technologies to implement the central components of a generative approach, such as: (i) XML-Schema [34] to specify domain-specific languages; (ii) Java and AspectJ [9] programming languages to implement the agent architecture and components; and (iii) Eclipse technologies [11, 30] to build the code generator.
The remainder of this paper is organized as follows. Section 2 introduces the basic concepts of generative programming and aspect-oriented software development. Section 3 presents an overview of our aspect-oriented generative approach and details the process of domain analysis and design. Section 4 describes the steps to implement the generative approach. Section 5 synthesizes some of the lessons learned during the definition of the aspect-oriented generative approach. Section 6 discusses some related work. Finally, section 7 provides some conclusions and directions for future work.

2. Background

2.1 Generative Programming

Generative Programming (GP) [8] addresses the study and definition of methods and tools that enable the automatic generation of software from a given high-level specification language. It has been proposed as an approach based on domain engineering [21, 27, 28].
GP promotes the separation of problem and solution spaces, giving flexibility to evolve both independently. To provide this separation, Czarnecki and Eisenecker [8] propose the concept of a *generative domain model*. A generative domain model is composed of three basic elements: (i) *problem space* – which represents the concepts and features existent in a specific domain; (ii) *solution space* – which consists of the software architecture and components used to build members of a software family; and (iii) *configuration knowledge* – which defines how specific feature combinations in the problem space are mapped to a set of software components in the solution space. GP advocates the implementation of the configuration knowledge by means of code generators.
The fact that GP is based on domain engineering enables us to use domain engineering methods [1, 8] in the definition of a generative domain model. Common activities encountered in domain engineering methods are: (i) *domain analysis* – which is concerned with the definition of a domain

for a specific software family and the identification of common and variable features within this domain; (ii) *domain design* – which concentrates on the definition of a common architecture and components for this domain; and (iii) *domain implementation* – which involves the implementation of architecture and components previously specified during domain design.
According to Czarnecki and Eisenecker, two new activities need to be introduced to domain engineering methods in order to address the goals of GP:

- development of a proper means to specify specific members of the software family. Domain-specific languages (DSLs) must be developed to deal with this requirement;
- modeling of the configuration knowledge in detail in order to automate it by means of a code generator.

In this work, we have adopted the common activities – domain analysis, domain design and domain implementation – encountered in a domain engineering method to define the generative approach (such as described in [8]). However, we have also considered the other two activities presented above by implementing a domain-specific language and a code generator.

2.2 Aspect-Oriented Software Development

Aspect-oriented software development (AOSD) [23, 33] is an evolving approach aiming at modularizing 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. Crosscutting concerns are concerns that often crosscut several modules in a software system.
AOSD has been proposed as a technique for improving the separation of concerns in the construction of OO software, supporting improved reusability and ease of evolution. When developing OO software one faces an architectural dilemma: no matter how the OO system is factored, frequently there will be concerns that are handled in different classes. Hence, such concerns crosscut these classes. AOSD supports the modularization of crosscutting concerns by providing abstractions to extract these concerns and later compose them back when producing the overall system.
AOSD proposes the notion of *aspect* as a new abstraction and provides new mechanisms for composing aspects and *components* (classes, methods, etc.) together at specific join points. AspectJ [22] is an aspect-oriented extension to the Java programming language. The aspect abstraction in AspectJ is composed of *inter-type declarations*, *pointcuts* and *advices*. Pointcuts have a name and are collections of join points. Join points are well-defined points in the dynamic execution of system components. Examples of join points are method calls and method executions. Advice is a special method-like construct attached to pointcuts. Advices are dynamic crosscutting features since they affect the dynamic behavior of components. Inter-type declarations specify new attributes or methods to be introduced in specific classes.
As already mentioned, in this work we will focus on aspect-oriented abstractions to capture crosscutting concerns encountered in multi-agent system implementations. Examples of such concerns are interaction, autonomy, adaptation and collaboration [17].

3. An Aspect-Oriented Generative Approach

The aspect-oriented (AO) generative approach aims at exploring the horizontal domain [8] of multi-agent systems (MASs) to improve their quality and productivity. The purpose of the generative approach is threefold: (i) to uniformly support crosscutting and orthogonal (non-crosscutting) features

of software agents starting at early development stages [9, 29]; (ii) to abstract the common and variable features; and (iii) to enable the code generation of AO agent architectures.
Figure 1 depicts our generative approach that is composed of:

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

(ii) an AO architecture modeling a family of software agents. It is centered on the definition of *aspectual* components to modularize the crosscutting agent features;

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

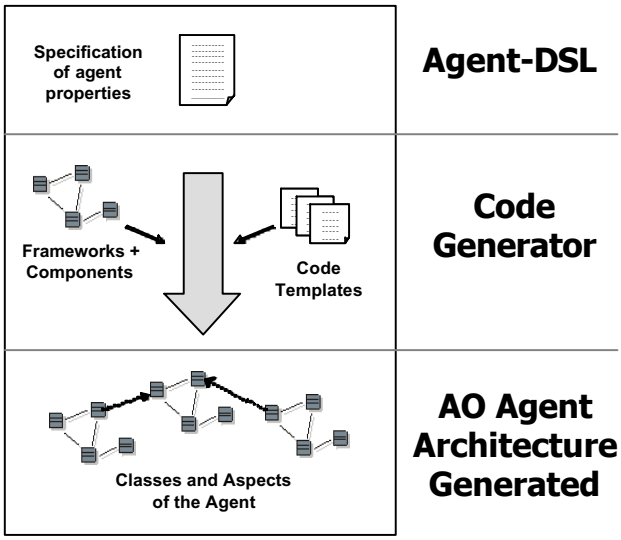


Figure 1. The Aspect-Oriented Generative Approach

The definition of our generative approach encompassed a typical domain engineering process. The steps followed in the development of the generative approach were:

1. Domain Analysis
   a. Definition of the domain
   b. Identification and modeling of common and variable features of the domain
   c. Identification and modeling of the crosscuting features of the domain
2. Domain Design
   a. Specification of the generic AO architecture
   b. Identification and specification of the DSLs
   c. Specification of the configuration knowledge
3. Domain Implementation
   a. Implementation of the DSLs
   b. Implementation of the AO architecture and additional components
   c. Implementation of the code generator

The following sections describe in more detail most of these steps. Section 3.1 describes the domain analysis phase by presenting the resulted feature model and the proposed notation to represent crosscutting features in a feature model. Section 3.2 presents the AO agent architecture and the proposed notation to represent aspectual and non-aspectual components. Section 4 describes the steps to implement the elements of the generative approach.

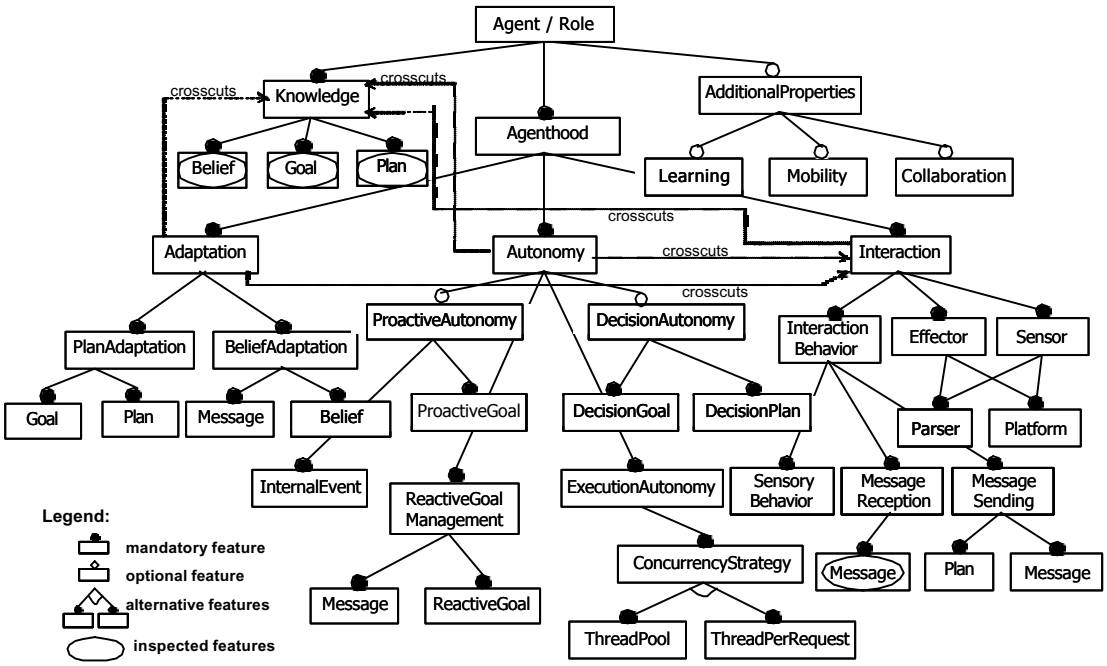


Figure 2. Partial Feature Model of a Agent / Role

In the domain design we have defined an AO architecture that was implemented using as base an AO framework. Because of this decision, the step 2(b) that involves the identification and specification of the DSLs was simplified. It was necessary a definition of a sole configuration DSL used to instantiate the AO framework. Section 4.1 presents the specification and implementation of this DSL.

Due to limited space the step 2(c) of the domain design is not described in this paper. The specification of the configuration knowledge was accomplished in our work by defining a pattern language [12]. A pattern language defines how a set of interrelated design patterns can be used together to address a larger problem. Our pattern language shows how the domain features of MASs can be mapped to specific design structures of classes and aspects. A complete description of this pattern language can be found in [13].

3.1 Domain Analysis

During the domain analysis, recurring agent concerns of multi-agent systems (MASs) were modeled using feature models [21]. Feature models are used to represent common and variable features of system families. Our domain analysis was supported by experience gained from our extensive previous work on the development of several multi-agent systems [13-17], and by surveys of different MAS modeling languages, architectures and platforms [17, 32]. We captured the different features associated with the agent concept, including orthogonal and crosscutting agent features. Figure 2 depicts a partial feature model produced during this phase.

The agent concept is composed of its *knowledge* and its basic properties, which we termed "*agenthood*". The knowledge feature encompasses *beliefs*, *goals* and *plans*. Agent beliefs describe information about the agent itself and about the external environment with which the agent interacts. To achieve a goal, an agent executes a specific plan. During the execution of a plan, the agent manipulates its beliefs. The agenthood feature is composed of three subfeatures: *interaction*, *adaptation and autonomy.*

The interaction feature is the agent capacity to communicate with the environment. The agent can receive or send *messages* to the environment by means of its *sensors* and *effectors*, respectively. External messages are translated to the agent ontology using specific *parsers* in its sensors. Effector parsers translate internal messages to a specific external representation.
The adaptation feature is formed by *belief adaptation* and *plan adaptation*. Belief adaptation is responsible for interpreting received messages from the environment and for manipulating its beliefs based on the message contexts. Plan adaptation determines the plan the agent must execute whenever a new goal needs to be achieved.
The purpose of the autonomy feature is to instantiate and manage the agent goals. It deals with three types of goals: *reactive goals*, *proactive goals*, and *decision goals*. Reactive goals are instantiated when the agent receives an external request from other agents or environment components. Proactive goals are instantiated due to internal events that occurs, such as, the end of a plan execution or the achievement of a specific agent state. Finally, the decision goals are instantiated due to external or internal events and are used to decide if special reactive or proactive goals could be instantiated. The autonomy property is also responsible for monitoring the adopted *concurrency strategy*. It supports the goal achievement by implementing a mechanism for executing concurrently agent plans.In addition to the agent knowledge and the agenthood features, an agent can incorporate additional properties. Additional features include *collaboration*, *mobility*, and *learning*. The current version of the generative approach just provides support for the collaboration feature. An agent collaborates with other agents by playing different roles. A role gives to the agent extra capacities of knowledge, interaction, adaptation and autonomy. Each agent can play different roles during its execution.
To support the representation of crosscutting features in feature models, a new kind of relation between features, called *crosscuts* relation, has been introduced. We say that a feature A crosscuts a feature B, when either A or one of its subfeatures depends and inspects B or one of the subfeatures of B. In the feature model of the Figure 2, for instance, *Adaptation* is characterized as a crosscutting feature because it is composed of two features (*BeliefAdaptation* and *PlanAdaptation*) that inspect common features of the *Agent Knowledge* (*Goal* and *Plan*) and the *Agent Interaction* (*Message*). As a result, the *Adaptation* feature crosscuts the *Knowledge* and *Interaction* features.
The *Interaction* feature is also characterized as crosscutting because it is composed of a subfeature (*MessageSending*) that inspects features of the *Agent Knowledge* (*Plan*). In addition, the *Autonomy* feature crosscuts the *Knowledge* and *Interaction* features.

3.2 Domain Design

Domain design consists of specifying a generic and flexible AO agent architecture for the domain at hand. Each feature modeled during domain analysis needs to be considered in the design. The AO agent architecture is a refinement of a previous work [16, 17]. It uses two kinds of components: (i) a central component that modularizes the orthogonal features associated with the agent knowledge; and (ii) the *aspectual components* that separate the crosscutting agent features from each other and from the `Knowledge` component. Aspectual components represent crosscutting features at the architectural level.

Figure 3 depicts the components of the AO agent architecture. We have used a new notation to graphically represent an AO architecture. It is an extension of the ASideML modeling language [6]. We developed this notation to enable the representation of aspectual components. An aspectual component may crosscut other aspectual or non-aspectual components using its crosscutting interfaces. A *crosscutting interface* may both add new state or behavior in other components and intercept (and modify) the existent behavior of components. Non-aspectual (normal) components are represented in a similar way to UML [3] and offer their services through the *normal interfaces*.

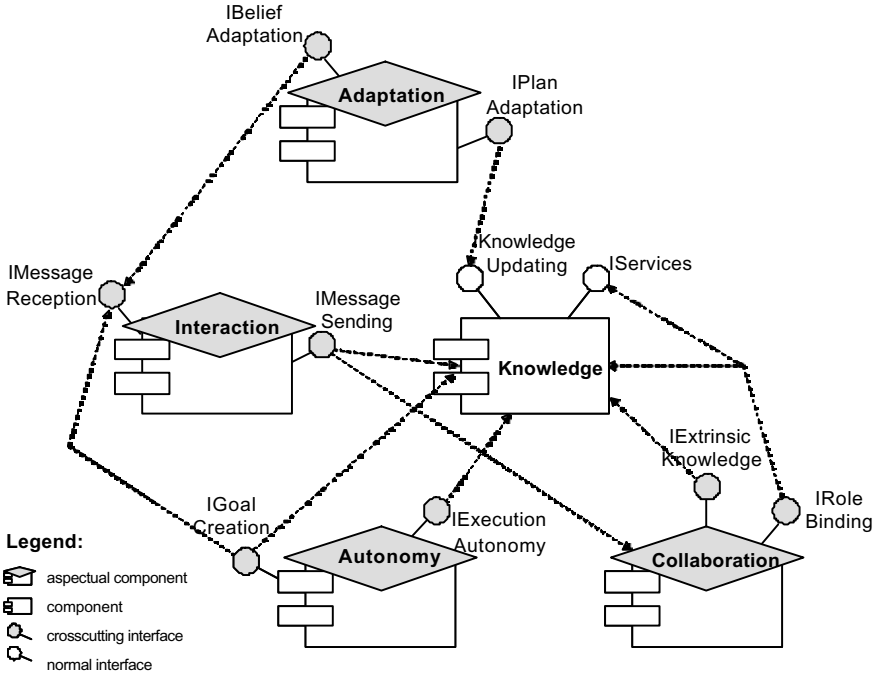


Figure 3. The Aspect-Oriented Agent Architecture

The `Knowledge` component models the orthogonal features (belief, goal, plan) related to the knowledge feature. It contains two normal interfaces: (i) `IKnowledgeUpdating` – to update the agent knowledge; and (ii) `IServices` – to offer agent services. In the domain implementation (section 4.2), this component is refined as a set of classes.

Each of the crosscutting agent features (interaction, adaptation, autonomy and role) are modeled as aspectual components in the agent architecture. Each aspectual component was refined during the domain implementation (section 4.2) as a set of aspects and auxiliary classes, which are also part of the crosscutting feature.

The `Interaction` aspectual component models the interaction crosscutting feature. It is composed of two crosscutting interfaces: (i) `IMessageReception` – which introduces the capacity to receive external messages into the `Knowledge` component; and (ii) `IMessageSending` – which crosscuts elements of the `Knowledge` component to define specific points where is necessary to send messages to the environment. It also crosscuts elements of the `Collaboration` aspectual component to specify specific points in collaboration plans where is also necessary to send messages to the environment.

The `Adaptation` aspectual component models the adaptation crosscutting feature. It is composed of two crosscutting interfaces: (i) `IBeliefAdaptation` – which intercepts the invocation of services provided by the `IMessageReception` interface of the `Interaction` component to

update agent beliefs when new external messages are received by the agent; and (ii) `IPlanAdaptation` – which intercepts the invocation of services provided by the `IKnowledgeUpdating` interface of the `Knowledge` component to instantiate new plans to be executed when the agent needs to achieve a specific goal.
Finally, the `Collaboration` aspectual component models the role crosscutting feature. It is composed of two crosscutting interfaces: (i) `IExtrinsicKnowledge` – which introduces new knowledge (state and behavior) associated with agent roles in the `Knowledge` component; and (ii) `IRoleBinding` – which defines specific points in the `Knowledge` component where agent roles are instantiated and bound to the agents.

4. Implementing the Generative Approach

This section describes the implementation of the generative approach elements: (i) the Agent-DSL, (ii) the AO agent architecture, and (iii) the code generator.

4.1 Agent-DSL

Based on the feature models defined in the domain analysis (section 3.1), we defined a configuration domain-specific language (DSL), called Agent-DSL. A configuration DSL allows to specify a concrete instance of a concept [8]. It can be directly derived from feature models. This language is used to specify the agent features that an agent instance could have to accomplish its tasks. It allows modeling the agent features, such as, knowledge, interaction, adaptation, autonomy and collaboration.
An XML Schema [34] was used to specify the semantics of the Agent-DSL. The feature models were translated to XML Schema complex types. For each specific agent of a MAS to be generated, it must be created an agent description XML document. This document must conform to the XML Schema that defines the Agent-DSL. The right side of Figure 5 depicts a partial specification of an agent type used in a case study developed by our research group [13]. Subsection 4.3 describes in more detail the case study.

4.2 The Aspect-Oriented Agent Architecture

The implementation of the generic AO agent architecture (section 3.2) was realized using Java and AspectJ [22] programming languages. The basis of the architecture implementation is an AO framework that contains hot-spots as classes and aspects [24]. Figure 4 presents a partial description of the AO framework. The ASideML modeling language [6] is used to represent visually the framework. This language extends UML with notations for representing aspects. The notations provide a detailed description of the aspect elements. In this modeling language, an aspect is represented by a diamond; it is composed of internal structure and crosscutting interfaces. The internal structure declares the internal attributes and methods. A crosscutting interface specifies when and how the aspect affects one or more classes [6]. Each crosscutting interface is composed of inter-type declarations, pointcuts and advices. The first part of a crosscutting interface represents inter-type declarations, and the second part represents pointcuts and their attached advices. The notation uses a dashed arrow to represent the crosscutting relationship, which relates one aspect to classes and/or aspects. Every class and aspect presented in the figure are hot-spots.
The *Knowledge* component (section 3.2) was refined as a set of classes – `Agent`, `Belief`, `Goal` and `Plan` classes. Each one of them represents a specific hot-spot that can be extended to define an

agent type. Agent beliefs are defined in our architecture as domain classes that `Agent` instances can aggregate. Each one of the aspectual components (section 3.2) was refined as a central aspect and a set of auxiliary classes. Figure 4 only presents the main aspects that refine the agent knowledge classes incorporating specific agent features.

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

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

The *Autonomy* component defines the `Autonomy` aspect, which enables the `Agent` class to instantiate and manage reactive goals and execute concurrently several plans (execution autonomy). However, for sophisticated agent types, the `Autonomy` aspect also allows to define proactive and decision autonomy. To instantiate reactive goals, the `Autonomy` aspect also intercepts the `receiveMsg()` method of the `Agent` class. This interception is used to verify if specific external events (for instance, a request of another agent) demand the instantiation of reactive goals. The execution autonomy is implemented in the `Autonomy` aspect by defining an Active Object [24], which monitors the list of plans to perform of the `Agent` class to execute them in separate threads. The proactive autonomy is implemented by specifying: (i) several pointcuts in agent knowledge classes that represent specific events of interest, and (ii) an advice associated with these pointcuts which is responsible for determining if a proactive goal must be instantiated in the occurrence of any of these events. Finally, the decision autonomy only defines a `makeDecision()` method in the `Autonomy` aspect that is invoked in the advices associated with the pointcuts of reactive and proactive goal instantiation. This method verifies whether it is necessary to execute a decision plan on the occurrence of a specific event or on the reception of a message. Autonomy subaspects can also be implemented to define specialized proactive, reactive and decision autonomy for each one of the agent types and roles defined in a MAS.

The *Collaboration* component is implemented by defining role aspects that introduce attributes and methods in an agent type (`Agent` class or subclass). These elements define respectively specific beliefs and behaviors of roles. Also, specific `Plan` and `Goal` subclasses must be defined for the roles. The plans defined for a role manipulate the attributes (beliefs) and invoke methods (behaviors) introduced by the role aspect. Goal classes specified for a role are instantiated by an `Autonomy`

subaspect which is specially created for the role. Specific interaction, adaptation and autonomy subaspects can be defined for an agent role. Next section exemplifies the definition of subaspects for agent roles of a specific case study developed by our research group.

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

- interaction feature: concrete sensors and effectors specially tailored to specific agent platforms (such as JADE [2]);
- autonomy feature: concrete concurrency strategies (such as thread pool and a thread per request) used by the active object to implement the agent's execution autonomy.

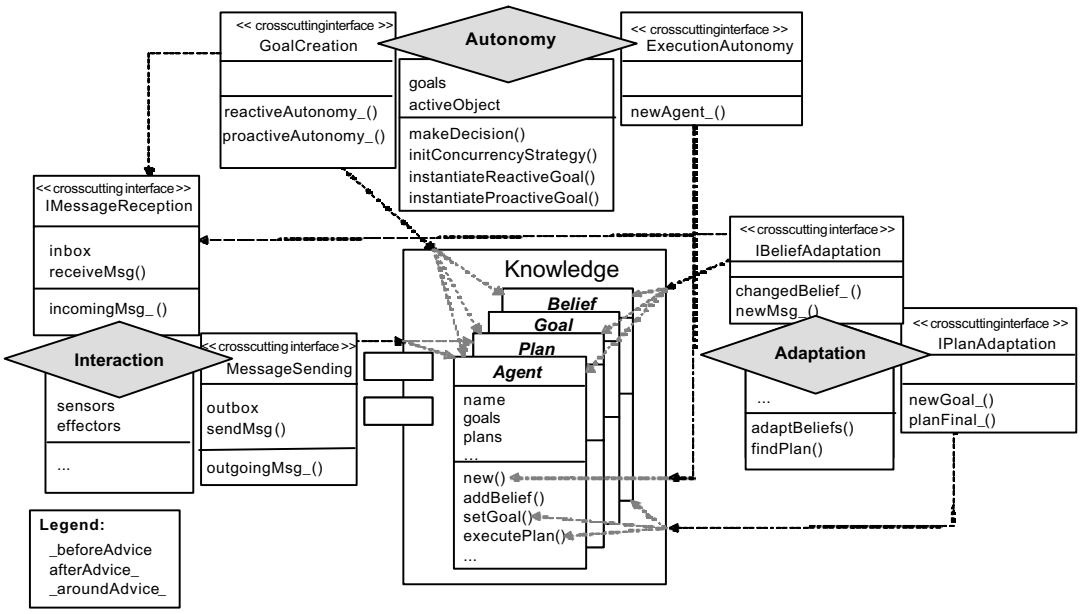


Figure 4. The Aspect-Oriented Agent Framework

4.3 Code Generator

In the configuration knowledge of the generative approach, we implemented a code generator as an Eclipse plug-in [30]. This generator maps abstractions in the Agent-DSL to normal and aspectual components of the agent architecture. The AO framework (section 4.2) is used as the basis of the agent architecture. The main task of the generator is to instantiate the framework, creating subclasses and subaspects for specific hot-spots of the framework. Depending on the agent descriptions provided, new types of agents (or roles) with their respective agent properties can be generated. We present below examples of classes and aspects generated for the context of a case study.

We have used the generative approach for the development of the ExpertCommittee (EC) system, which is a case study undertaken by our research group [13]. EC is an open system that supports the management of paper submissions and the reviewing process for a conference. Software agents have been introduced to EC in order to assist its users with time-consuming activities and automate repetitive user tasks. EC agents are software assistants that represent paper authors, chairs, PC members and reviewers and coordinate their activities. The EC system also includes information agents.

Figure 5 presents the elements of the generative approach applied to the EC system. The left side of the figure contains an agent description file for a specific agent type of the EC, called

`ResearcherUserAgent`. The chair role played by this agent type is also presented. We used the JAXB plug-in [20] to enable reading the agent description XML file by the code generator.
The center of the figure is the code generator. It is responsible for reading the agent description file and customizing code templates based on information collected by the Agent-DSL. Code templates allow us to represent structure and behavior of specific classes and aspects that we want to generate. They are used to represent each one of subclasses and subaspects of the framework hot-spots. Java Emitter Templates (JET), a generic template engine of the Eclipse Modeling Framework (EMF) [4], has been used to write the source templates. Examples of classes and aspects that we wrote as templates are: (i) concrete instances of hot-spots (classes or aspects), such as specific agent type classes, specific agenthood (interaction, adaptation and autonomy) subaspects; (ii) specific agent plans and goals classes; and (iii) specific role aspects.
Finally, the right side of the figure shows the specific elements (subclasses and subaspects) generated for the EC system. Several classes and aspects are generated based on its Agent-DSL description file and on the JET source templates. First, the `ResearchUserAgent` class, a specific agent type, is generated. The two roles played by instances of this class are also generated. They are called `Chair` and `Reviewer` aspects. Each one of them introduces specific beliefs and behaviors in the `ResearchUserAgent` class. Specific `Plan` and `Goal` classes are also generated to the two roles. Moreover, different `Interaction`, `Adaptation` and `Autonomy` subaspects are generated to each one of the roles. For instance, the `ChairInteraction`, `ChairAdaptation` and `ChairAutonomy` aspects are produced to agents playing the chair role. `ChairInteraction` initializes JADE sensors and effectors to be used by the agents playing the chair role. `ChairAdaptation` realizes specific belief and plan adaptation of the chair role. Finally, `ChairAutonomy` defines: (i) a reactive autonomy – to instantiate specific goals when receiving external messages from reviewer agents; (ii) a proactive autonomy – to instantiate specific goals when internal events occur; and (iii) an execution autonomy – which defines a "thread per request" concurrency strategy to execute agent plans.

5. Discussion and Lessons Learned

Based on the experience of development of an AO generative approach, we have already identified some important requirements and techniques that are useful and relevant during the integration of GP and AOSD technologies. Below, we synthesize these lessons learned.

- *support to crosscutting features in the domain analysis* – the modeling of crosscutting concerns in early design phases has been recognized recently as an important topic of research in AOSD [9, 29]. The extension of feature models to represent crosscutting features, presented in this paper (section 3.1), helps to become explicit the existence of crosscutting concerns in system families during domain analysis. More research need to be developed to understand better how to model crosscutting features and their interactions;
- *specification of aspect-oriented architectures* – a fundamental activity when constructing program families (and product lines) is the modeling of a software architecture that addresses common and variable features. We have presented in this paper (section 3.2) a new notation that allows representing aspectual components. The use of this notation enables us to represent in a concise way the main components (and their interactions) of AO architectures. The definition of principles and patterns to model AO architectures so that it can be possible to address different crosscutting concerns is an important topic of research that we intend to explore;

•

```
<MAS ... xsi:noNamespaceSchemaLocation=
                        "agent-dsl.xsd">
  <agent>
    <name>ResearcherUserAgent</name>
    <belief> ... </belief>
    <goal> ... </goal>

    <role>
      <name>Chair</name>
      <belief> ... </belief>
      <goal>

<name>PaperDistributionGoal</name>
        <type>Reactive</type>
      </goal>
      <plan>

<name>PaperDistributionPlan</name>
        <type>Reactive</type>

<communication>false</communication>
      </plan>

      <interaction>
        <sensor>
          <name>sensorAgent</name>

<type>AgentCommunication</type>
          <platform>JADE</platform>
        </sensor>
        <effector>
          <name>effectorAgent</name>

<type>AgentCommunication</type>
          <platform>JADE</platform>
        </effector>
        <message>

<id>REQUEST_DISTRIBUTE_PAPER</id>

<performative>REQUEST</performative>
          <service>
              SERVICE_DISTRIBUTE_PAPERS
          </service>
        </message>
      </interaction>

      <autonomy>
        <executionAutonomy>
          <concurrencyStrategy>
              ThreadPool
          </concurrencyStrategy>
        </executionAutonomy>
        <reactiveAutonomy>
          <messageToGoal>
            <message>
              REQUEST_DISTRIBUTE_PAPER
            </message>
            <goal>
              PaperDistributionGoal
            </goal>
          </messageToGoal>
        </reactiveAutonomy>
        ...
      </autonomy>

      <adaptation>
        <planAdaptation>

<goal>PaperDistributionGoal</goal>

<plan>PaperDistributionPlan</plan>
                                daptation>
                              Adaptation> ...
                          </beliefAdaptation>
        </adaptation>

    </role>
    ...
  </agent>
</MAS>
```

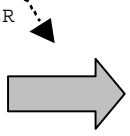
Agent Type Code

Adaptation Subaspect Code

Role Aspect Code

Interaction Subaspect Code

Autonomy Subaspect Code



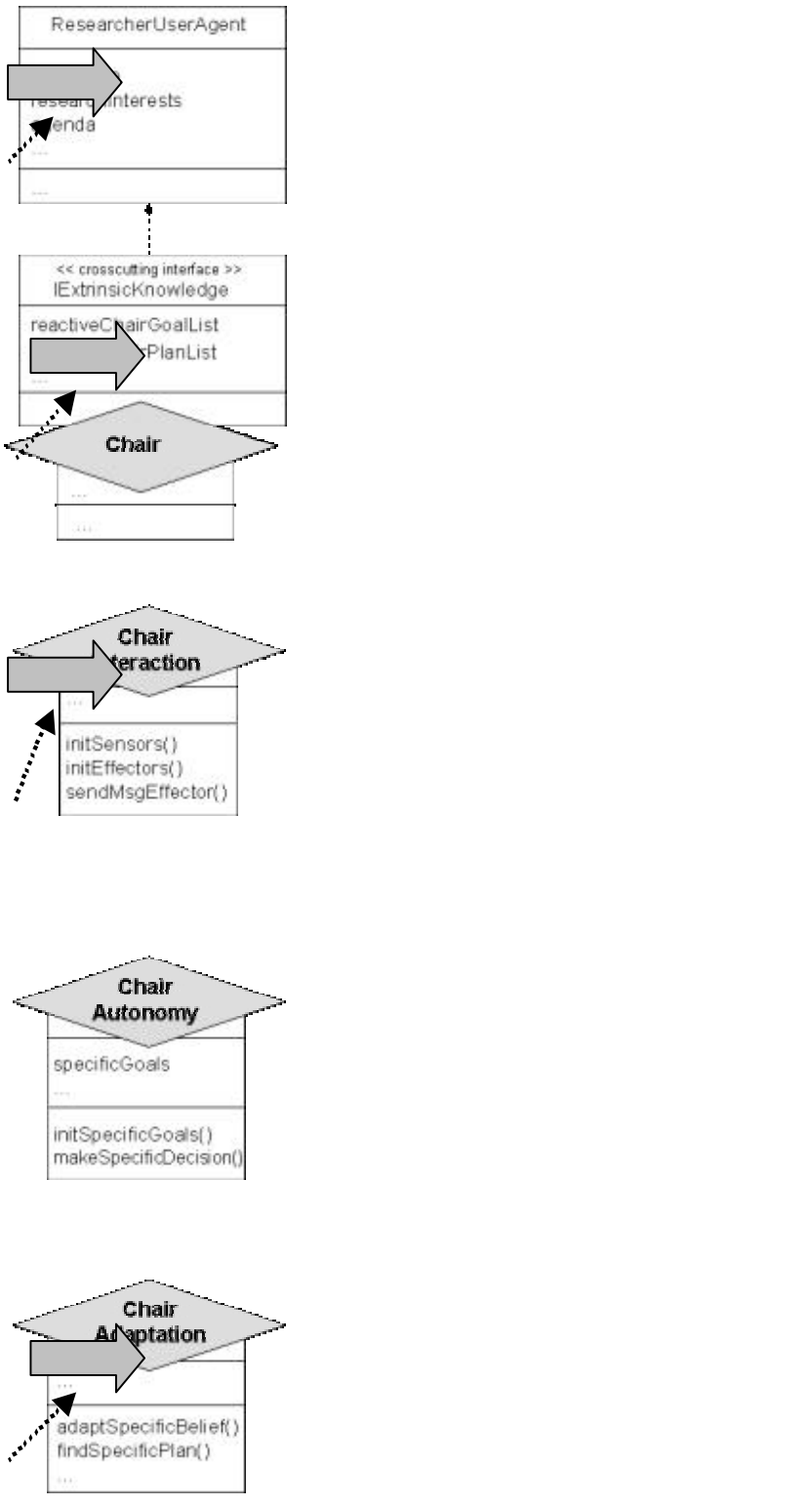


Figure 5. Problem Space | Configuration Knowledge | Solution Space (Agent-DSL) (Code Generator) (Agent Architecture Generated)

- *specification of the configuration knowledge* – the configuration knowledge in a generative domain model (section 2.1) defines how specific combinations of features in the problem space can be mapped to specific combinations of components in the solution space. The configuration knowledge was specified in our work by defining a pattern language [13]. A pattern language defines how a set of interrelated design patterns can be used together to address a larger problem. Each one of the components of the AO architecture was specified as a design pattern of the pattern language. The description of each pattern emphasizes: (i) a specific problem in the context of agent architectures; and (ii) a flexible design structure to address that problem;
- *construction of domain-specific languages* – during the implementation of our generative approach, we have defined a unique configuration domain-specific language to express orthogonal and crosscutting features of software agents (section 4.1). Configuration DSLs are recognized as a suitable alternative to define generative approaches that require to instantiate object-oriented frameworks [5, 11]. In our case, it was also suitable to represent the crosscutting features encountered in a definition of a software agent. An interesting work is the investigation of the combined use of a configuration DSL to express only the orthogonal features and aspectual DSLs to express each one of the crosscutting features (see related work in section 6) existent in a domain;
- *implementation of aspect-oriented frameworks* – object-oriented frameworks are a common and useful technology to implement architectures for system families [10]. They address the implementation of common (frozen-spots) and variable (hot-spots) features of system families. The use of the aspect abstraction in the definition of frameworks enables us to define common and variable behaviors of crosscutting features. An AO framework that models a set of crosscutting features encountered in agent architectures has been presented in this work (section 4.2). We believe that AO frameworks are a fundamental technology to be used during the implementation of AO generative approaches. In the implementation of our framework it was used some of the AspectJ idioms presented in [19] (such as, *Abstract Pointcut*, *Chained Advice*, *Pointcut Method*, *Template Advice*). We found that they represent basic and recurrent constructions used to define AO frameworks in AspectJ.

6. Related Work

Some recent reports explored the integration of GP and AOSD [18, 26, 31]. However, these reports have not covered or described in sufficient detail all the typical phases encountered while developing a generative approach.

Pinto et al [26] have proposed DAOP-ADL, an architecture description language used to describe software architectures. This language supports the concepts of components and aspects. DAOP-ADL is interpreted by DAOP (Dynamic Aspect-Oriented Platform), a specific component and aspect-based middleware platform [27]. DAOP platform uses the information presented in the ADL to compose dynamically components and aspects of a application. DAOP-ADL enables the specification of AO architectures independent of programming languages and component platforms. The notation (section 3.2) proposed in this paper can also be formalized as an ADL.

Shonle et al [31] have developed XAspects, an extensible system that allows to define aspectual domain-specific languages. An aspectual DSL allows to express specific crosscutting concerns into modularized constructs [8]. Examples of aspectual DSLs presented

by them are [31]: (i) a coordination language used to specify thread coordination concerns; and (ii) a traversal language used to express collaborative behavior between classesXAspects also provides support to generate Java and AspectJ code derived from one or more aspectual DSLs. In our work we have studied the domain of software agents to define a unique DSL that express orthogonal and crosscutting agent features.

Colyer et al [7] have presented some principles that guide the definition of flexible and configurable AO systems. They show the proper use of AOSD technologies to follow these principles. They also illustrate how the application of the principles can help in the configuration of different features in a product line. The work presented by these authors is an important step in the definition of guidelines for constructing program families in order to maximize configurability in AO architectures. This kind of principle is very useful to guide the specification of AO architectures that can be configured with different feature combinations in a generative approach.

7. Conclusion and Future Work

This paper reported our experience in the definition of an AO generative approach. The goal of this approach is to explore the horizontal domain that MASs represent in order to enable the code generation of agent architectures. We organized the development of the generative approach using typical phases encountered in domain engineering processes. During the development process of the generative approach, it was necessary to adapt modeling notations used in generative programming due to the adoption of AOSD. The feature model was extended to support the representation of crosscutting features (section 3.1). Also, a new notation was proposed to support the representation of AO architectures (section 3.2). Aspectual components have been used to model crosscutting features from the architectural point of view.

We believe that the definition of AO generative approaches can bring important benefits to the development of software families. GP allows: (i) to evolve the problem and solution spaces independently; and (ii) to define clearly the mapping between high-level features and implementation components. The integrated use of GP and AOSD techniques brings additional benefits, such as: (i) clear separation of orthogonal and crosscutting features starting at early design phases; and (ii) direct mapping of crosscutting features in aspectual components. This latter benefit simplifies the implementation of code generators, because the composition of crosscutting concerns is accomplished by the aspect weavers. Using only OO abstractions, crosscutting agent features need to be hand-coded in the code of classes.

This work aimed at identifying relevant techniques and requirements to be considered on the development of AO generative approaches. It represents a significant step in the definition of a method to develop AO generative approaches. In this context, our future work consists of the: (i) application of the same process presented in this paper to develop an AO generative approach for a different software domain; (ii) investigation of the benefits and drawbacks of the use of aspectual domain-specific languages; and (iii) definition of a set of principles and guidelines to specify artifacts in a generative domain model considering the use of AOSD technologies using the notations presented.

Acknowledgments. This work has been partially supported by CNPq under grant No. 140252/2003-7 for Uirá Kulesza, grant No. 141457/2000-7 for Alessandro Garcia, 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 and by the art. 1st of Decree number 3.800, of 04.20.2001.

References

1. A rrango, G. Domain Analysis Methods. In Software Reusability, Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds.), Ellis Horwood, New York, pp. 17-49, 1994.
2. Bellifemine, F., Poggi, A., Rimassi, G. JADE: A FIPA-Compliant Agent Framework. Proc. Practical Applications of Intelligent Agents and Multi-Agents, pp. 97-108, April 1999.
3. Booch, G., Jacobson, I., Rumbaugh, J. Unified Modeling Language - User's Guide. Addison-Wesley,1999.
4. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T. Eclipse Modeling Framework, Addison-Wesley, 2003.
5. Cechticky, V. et al. A Generative Approach to Framework Instantiation. Proceedings of the GPCE´2003, Erfurt, Germany, September 2003.
6. Chavez, C. A Model-Driven Approach to Aspect-Oriented Design. PhD Thesis, Computer Science Department, PUC-Rio, April 2004.
7. Colyer, A., Rashid,A., Blair,G. On the Separation of Concerns in Program Families. Technical Report, Computing Department, Lancaster University, January 2004.
8. Czarnecki, K., Eisenecker, U. Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
9. Early Aspect Home-Page, Available at URL http://www.early-aspects.net/
10. Fayad, M., Schmidt, D., Johnson, R. Building Application Frameworks: Object-Oriented Foundations of Framework Design. John Wiley & Sons, September 1999.
11. Fontoura, M. et al. Using Domain Specific Languages to Instantiate Object-Oriented Frameworks. IEE Proceedings - Software, 147(4), 109-116, August 2000.
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing, 1995.
13. Garcia, A. From Objects to Agents: An Aspect-Oriented Approach. PhD Thesis, Computer Science Department, PUC-Rio, April 2004.
14. Garcia, A., Cortés, M., Lucena, C. A Web Environment for the Development and Maintenance of E-Commerce Portals based on a Groupware Approach. Proceedings of the Information Resources Management Association International Conference (IRMA'01), Toronto, May 2001.
15. Garcia, A., Lucena, C. Software Engineering for Large-Scale Multi-Agent Systems. ACM Software Engineering Notes, Vol. 27, Number 5, September 2002, pp. 82-88.
16. Garcia, A., Lucena, C., Cowan,D. Agents in Object-Oriented Software Engineering. Software: Practice & Experience, Elsevier, V. 34, Issue 5, May 2004, pp. 489-521.
17. Garcia, A., Silva,V., Chavez, C., Lucena, C. Engineering Multi-Agent Systems with Aspects and Patterns. Journal Brazilian Computer Society, July 2002,1(8),pp.57-72.

18. Gray, J., Bapty, T., Neema, S., Schmidt, D., Gokhale,A., Natarajan,B. An Approach for Supporting Aspect-Oriented Domain Modeling. Proceedings of the GPCE´2003, pp. 151-168 Erfurt, Germany, September 2003.
19. Hanenberg, S., Unland, R., Schmidmeier, A. AspectJ Idioms for Aspect-Oriented Software Construction. Proceedings of the 8th European Conference on Pattern Languages of Programming (EuroPlop'03), Irsee, Germany, June 2003.
20. JAXB Eclipse Plug-in. Available at URL http://sourceforge.net/projects/jaxb-builder/
21. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A. Feature-Oriented Domain Analysis (FODA): Feasibility Study. Technical Report CMU/SE4-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
22. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. Getting Started with AspectJ. Communications of the ACM. October 2001.
23. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J. Aspect-Oriented Programming. In Proceedings of the ECOOP´97, LNCS (1241), Springer-Verlag, Finland, June 1997.
24. Lavender,R.,Schmidt,D. Active Object:an Object Behavioral Pattern for Concurrent Programming. In: Pattern Languages of Program Design, Addison-Wesley, 1996.
25. Pinto, M., Fuentes, L., Fayad, M., Troya, J. Separation of Coordination in a Dynamic Aspect Oriented Framework. Proceedings of the AOSD'02, April 2002, Enschede, Netherlands.
26. Pinto, M., Fuentes, L., Troya, J. DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development. Proceedings of the GPCE´2003, Erfurt, Germany, September 2003, pp. 118-137.
27. Prieto-Diaz, R. Domain Analysis for Reusability. Proceedings of the 11th COMPSAC - Computer Software & Applications Conference, Tokyo, Japan, October 1987, pp. 23-29.
28. Prieto-Diaz, R., Arango, G. (Eds). Domain Analysis and Software Systems Modeling. IEEE Computer Society Press, Los Alamitos, CA, 1991.
29. Rashid, A., Moreira, A., Araújo, J. Modularization and Composition of Aspectual Requirements. Proceedings of the AOSD´2003. 2nd International Conference on Aspect-Oriented Software Development, ACM, pp. 11-20.
30. Shavor, S., D'Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J., McCarthy, P. The Java Developer's Guide to Eclipse. Addison-Wesley, 2003.
31. Shonle, M., Lieberherr, K., Shah, A. XAspects: An Extensible System for Domain Specific Aspect Languages. In Companion of the OOPSLA´2003, pp. 28-37.
32. Silva, V., Garcia, A., Brandao, A., Chavez, C., Lucena, C., Alencar, P. Taming Agents and Objects in Software Engineering. In: Software Engineering for Large-Scale Multi-Agent Systems, LNCS 2603, Springer-Verlag, 2003.
33. Tarr, P., Osher, H., Harrison, W., Sutton Jr, S. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In Proceedings of the 21st International Conference on Software Engineering (ICSE'99), May 1999, pp. 107–119.
34. XML Schema. Available at URL http://www.w3.org/XML/Schema.