An ADL dealing with aspects at software architecture stage

Amparo Navasa *, Miguel A. Pérez-Toledano, Juan M. Murillo
Quercus Software Engineering Group, Department of Computer Science, University of Extremadura, Spain

A R T I C L E   I N F O
Article history:
Received 6 November 2007
Received in revised form 19 March 2008
Accepted 29 March 2008
Available online 7 April 2008
Keywords:
Software architecture
AOSD
Aspect-oriented architecture
Aspect-oriented ADL

A B S T R A C T
Managing complex software systems is one of the most important problems to be solved by software engineering. The software engineer needs to apply new techniques that allow for their adequate manipulation. Software architecture is becoming an important part of software design, helping the designer to handle the structure and complexity of large systems, and AOSD is a paradigm proposed to manage this complexity by considering crosscutting concerns throughout the software’s life-cycle. The suitability of the existence of an Aspect-Oriented (AO) architectural design appears when AO concepts are extended to the whole life-cycle. In order to adequately specify the AO design, aspect-oriented architecture description languages are needed. The formal basis of these will allow architects to reason about the properties of the software architecture. In this paper, a new architecture description language – AspectLeda – is formally described in order to adequately manipulate AO concepts at the software architecture stage. The AspectLeda translation process is also described. A toolkit assists the architect during the process. Finally, a prototype of the system can be obtained, and the correctness of the architecture obtained can be checked.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction: complex systems, AOSD and software architecture

Managing complex software systems is one of the most important problems to be solved by software engineering. The software engineer needs to deal with ever more complex software systems, having ever stricter requirements. Modularity is an essential feature in being more reusable and maintainable. Moreover, the need to support software modularity and non-functional requirements are essential challenges to be faced in software development. The application of software engineering principles is necessary in order to cope with these challenges. In addition, the real world changes and software systems have to adapt to it to remain useful. For this reason, software systems need to be easily adapted (or reconfigured) and software engineers should design tools which allow the maintenance of complex systems easy. Nevertheless, traditional approaches (possibly structured objects or the object oriented paradigm – OOP) only allow partial reconfigurability and reuse of the system.

OOP is a programming paradigm that performs a functional decomposition [41] but has the limitation that some concerns can not be located in a single modular piece but cross several structural pieces. Separation of concerns which was introduced in [41] is a principle of software engineering which encourages dealing with the different concerns of a system individually. Dijkstra in [21] demonstrated that this division provides better results and offers many other advantages, including reducing software complexity and improving the modularity, reusability, and maintenance of software artefacts. In 1996, Kiczales proposed Aspect-Oriented Programming – AOP – [30] as a way to apply Separation of Concerns in software development to deal with the limitations of OOP. This proposal is different from previous ones: AOP introduces a new notion of concern: “those interests which pertain to the system’s development, its operation or any other aspects that are important to stakeholders”. From then on, AOP and other techniques and technologies focused on the modularisation of crosscutting were grouped under the name of Advanced Separation of Concerns. The idea behind it is to extract the code related to the crosscutting concerns (a concern that captures requirements that cross multiple modules in a system [2]), which is spread through the system code in independent single modules. This separation avoids tangled code and allows the reuse of the same aspect in different software units (objects, components, modules, etc.). AOP applies the notion of aspect to structuring software systems, in order to easily develop, understand, evolve, and maintain them [30]. In several research works, aspects have been considered as an important part of the complex systems development process, and they propose techniques that extract them in an effective way: composition Filters [12], AspectJ [29], Adaptive Programming [31]. These Aspect-Oriented (AO)
New approaches in software engineering consider software architecture as an important part of the design stage. Software Architecture (SA) is an activity that helps developers not only to control a system's complexity but also to define its structure in order to obtain easy maintenance and evolution. This is because during the software architecture stage, the structure of a system is considered as the decomposition of interconnected modules. Systems are defined as a set of components describing their high level functionality, which are connected through a set of architectural connectors.

The following paragraphs describe the suitability of relating the two concepts (aspect orientation and software architecture) to manage the system's complexity: AOSD and complexity, software architecture and complexity, and finally, we consider system complexity from an architectural point of view in the AOSD paradigm.

- In order to develop large systems, software engineers need to apply techniques which reduce their complexity. One of them is AOSD. This discipline helps to improve the software development process, promoting the use of the concept of aspect throughout the life-cycle. It provides approaches to allow an early identification of the aspects: their extraction, representation, and composition are the main goals. AOSD considers aspects as first-class entities which can be manipulated during the system development process. In addition, aspect definition is an abstraction mechanism which can be added to an existing system in such a way that the added elements are not spread over several modules of the system, but remain independent of them. AOSD allows engineers to model as software artefacts those properties which cross the system. Because of this, large systems developed under this paradigm can be adapted to changes and to the evolution of their properties more easily than under traditional ones, in which the transversal code becomes scattered over several modules or system components.

- Software Architecture and complexity can be related by taking into account the following: SA is generally considered to play a fundamental role in coping with the inherent difficulties of the development of complex systems. SA represents a point of view from which the software architect can conceive large scale abstractions to facilitate the communication between different stakeholders, to enable the parallel development of the system, to provide a means of directing design decisions and their evaluation, and to facilitate their reuse. The following definition can be applied to the context of software systems: “the software architecture of a system represents its structure, which comprises software components and its externally visible properties and the relationships among them” [8]. This is because during SA, the system structure is defined as both a set of components doing computations and a set of connectors defining the interaction between them. In addition, SA embodies the earliest design decisions allowing the assurance of quality factors. Architectural designs can be represented in several ways, one of which is by way of an architectural description language (ADL).

Finally, system complexity can be managed from an architectural point of view in the AOSD paradigm. Considering SA and AOSD jointly is really interesting because both approaches make the management of the system's complexity easier [23] than using traditional approaches:

- On the one hand, SA considers system structure when it is being defined. Thus, it is possible to say that SA is a discipline which studies system modularisation and composition, and their consequences. This is close to the principle of the separation of concerns, which states that every concern should be dealt with in a separate module. After modules have been defined, complex structures combining them appear, and they must be studied together. Hence, the need for SA.
- On the other hand, AOSD allows systems to be generated with clear code and make their evolution easier: AOSD facilitates the aspect replacement of particular aspects and the introduction of new aspects [23]. Crosscutting concerns code can be implemented in a single modular unit. This improves reusability and adaptability. The evolution of the systems implementation is also made easier.

The concept of aspect at the architecture level introduces a new kind of modularisation and composition in software, which defines new structures that must be studied by SA as well as determining the architectural features of aspects.

During architecture design, the system’s structural definition is made. The suitability of the definition of an AO architecture design appears when it is observed that crosscutting concerns cross architectural components, contaminating components and connectors, and the final design becomes complex. Besides, considering AO concepts at this phase can help with the management of the system’s evolution. Recently, numerous studies, whose results show the beneficial effects of an architectural approach, have been presented (AOSD [2]. Early-Aspects [22] sites). From them, one can deduce that aspects, during the architecture design level, can be considered as software artefacts. For this reason, mechanisms to identify and specify aspects during architectural design as well as to manage their interaction with other architectural elements are needed. As a consequence, using the two concepts jointly enhances the characteristics of each. Different proposals consider architectural aspects in different ways: as components, connectors, or architectural views, and with either a symmetric or an asymmetric approach to dealing with aspects (Section 5). Our model considers architectural aspects as architectural components in an asymmetric approach.

The aim of this paper is to formally describe AspectLEDA, an Aspect-Oriented Architecture Language (AO-ADL). To facilitate the understanding of its structure, the description of the Aspect-Oriented Software Architecture Model (AOSA Model) [37], on which the new language is based, is presented briefly. The method proposes giving a structural specification of an AO system by considering aspects as first-class entities as well. It is based on the combined use of a conventional Architecture Description Language (ADL) and a coordination model. An ADL is a language for modeling software systems in terms of components, connectors and configurations [51]. Unfortunately, regular ADLs do not give support for AO concepts. Moreover, the extension of a system with aspects previously designed should be done by considering the obliviousness principle [24]. In this way, components constituting the system do not change when an extension is applied. In order to solve the problem of executing system components and aspects in a coordinated way, an exogenous control-driven coordination model [4] is considered. The one selected was Coordinated Roles (CR) [35] based on the Event Notification Protocols.

1 In [22] an aspect is defined as “a modular unit designed to implement a concern”.
1.1. Putting Aspects into ADLs

In a general way, software engineering provides designers with tools to express systems adequately over the life-cycle. This can be done by using graphic models (such as UML) or formal models (such as specification languages or architectural languages). When AO systems are going to be developed, tools are also necessary to express them in a graphic or a formal way.

Several proposals and models to obtain AO systems have been developed through different stages in the life-cycle. However, only some of them have linguistic support. We consider that it is necessary to endow AO systems with formal support, the same as in the development of regular systems, particularly when the architectural definition of a system is made. The architectural structure of regular systems can be expressed by means of ADLs. Similarly, when the architectural structure of an AO system is determined, it should be expressed by means of an adequate ADL. In this paper, we show how it is possible to endow the development process of an AO system with linguistic support from the architectural design phase.

Sometimes, extending or evolving existing software systems (without changing the components constituting them) can be interesting. In this case, extensions should be considered as aspects and thus the AOSD approach can be considered. In addition, to formally express these extensions at the architectural level, an ADL should be used. But regular ADLs do not allow the designer to define the extended system because of the special characteristics of the aspects. For this reason, the definition of an AO-ADL is proposed: AspectLEDA, which is defined based on the ideas and formal support given by another previously defined language (LEDA).

AspectLEDA is then defined by means of a small set of instructions (Section 4), and the AspectLEDA specification will be finally translated into an equivalent pure-LEDA architecture so that the specification of the evolved system can enjoy all the advantages of using it, particularly its formal basis. Moreover, an internal architecture is created in order to support aspect addition. How to use these instructions is explained in Section 4.2 after describing the model which defines the elements used in this AO-ADL.

As a consequence, AspectLEDA (based on the AOSA Model and on a regular ADL) allows for the integration of aspects into previously designed systems. Hence, our study proposes endowing the architectural definition of AO systems with linguistic support maintaining the obliviousness principle after defining an architectural model and an AO-ADL.

The article is structured as follows. In Section 2, the motivations are described for defining the methodological approach and the ADL presented in subsequent sections. In Section 3, our Aspect-Oriented Software Architecture Model (AOSA Model) to develop AO systems at the architecture design stage is briefly presented together with associated methodological considerations. The way in which AO systems can be endowed with linguistic support is also discussed. In Section 4, the detailed characteristics of the new language are presented. It is explained how an extended system is obtained from that language, and a set of primitives is described which allows the AO concepts to be represented. In Section 5, some related works are presented. Some conclusions and future work are presented in Section 6. Finally, Appendix A introduces LEDA ADL and Appendix B shows the automatically generated equivalent architecture for an AspectLEDA architecture.

2. Motivation

In this section, motivations for defining an architecture model supporting the design of AO systems are presented. In addition, a review of the characteristics of the development process taking into account AO concepts over the life-cycle is briefly described too.

The treatment of crosscutting concerns at the architectural design level and aspect modeling in architecture have captured the interest of the research community. Proof of this is the existence of books [23], and the specific workshops on Early Aspect at the main Software Engineering conferences. One of the interests of Early Aspects research is that of AO-ADL proposals. The motivation for this comes from the important roles that languages of this type must play:

1. **Descriptive tool.** Aspect-Oriented Programming considers the code modularization of the crosscutting concerns. Also, AOSDs have led to major improvements in reusability, maintainance, evolution, quality, etc. However, in order to exploit all the potential benefits of AOP it is necessary to communicate AO designs to the construction team. It is expected that AO-ADLs will provide the linguistic support to express those designs.

2. **Provider of formal support.** AO-ADLs should provide the formal support needed for the correctness of the aspect-oriented architecture to be checked. In the same way that a conventional ADL provides support for reasoning about the properties of the architectures, AO-ADLs must provide support for reasoning about the properties of AO architectures in terms of the components, the connections, the aspects, and their links with the system.

Since aspects can be added to an existing architecture modifying its behaviour and since this can be done in a transparent way to components, the second role acquires special relevance in the present work. In fact, when an aspect is added to an existing architecture, one can be sure about the change of behaviour it will produce in the affected components. However, one is left unsure of how pre-existing aspects may interfere with the new one, so that the final behaviour of the system is unpredictable.

Our research proposes using an AO architecture model to define the structure of AO systems. After the architectural structure has been obtained, it must be formalized by means of an ADL supporting AO concepts: AspectLEDA. Before summarizing our model and describing the language, we want to point out the increasing interest in developing complex systems by using AO concepts throughout the life-cycle. As mentioned before, aspects can be considered during the development process, from requirement specification to the detailed design and programming level. In [16], there has been presented an overview of the main aspect-oriented requirements engineering, architecture, and design approaches, and some comparisons among them are made. Also, Cuesta in [19] includes the enumeration and classification of different approaches and models developed in the context of SA. Below, it is briefly shown how the different phases of software engineering can be considered from an aspect-oriented point of view. Some of the most interesting AO models are mentioned.

Aspects-Oriented Requirements Engineering techniques recognise the importance of considering crosscutting concerns (functional or non-functional) during the requirements specification. From this, Aspect-Oriented Requirements Engineering Approaches (AORE) arise. Obtaining a modular representation of crosscutting requirements allows developers to ensure the traceability of crosscutting concerns throughout the life-cycle. AORE approaches adopt the separation of concerns principle at the analysis phase (early separation of concerns) as well as providing a representation of crosscutting concerns as requirements artefacts. These approaches also promote the composition of concern/requirement with other elements of the system under construction, where concerns are defined as any matter of interest in a software system. As an example we can mention the following: Arcade [3,50], MDSoC [38,39], Aspect-Oriented Component Engineering (AOCE) and Aspect-Oriented Component Requirement Engineering (AOCRE) [26,27], the Theme

In Aspect-Oriented Architecture Design approaches, an architectural aspect is an architectural module that has a broad influence on a number of other architectural modules. Security, authorization, authentication, and encryption/decryption are examples of architectural modules. Aspectual architecture design approaches provide mechanisms to identify architectural aspects, with them being explicit in the system architecture. Examples are: CAM/DAOP-ADL [48], PRISMA Model [43], FAC model [45] extending Fractal [46] with aspect concepts, the TransSAT method [6,7], and the Aspectual Software Architecture Analysis Method. ASAAM [55].

The objective of Aspect-Oriented Design (AOD) is to characterise and specify the behaviour and the structure of systems. Concerns of a system that are scattered and tangled in traditional approaches can now be modularised. Module cohesiveness is enhanced, and module coupling reduced. AOD also supports the specification of concerns modularisation and composition, and focuses on the explicit representation of crosscutting concerns using design languages. AOD languages are a way to specify aspects and how aspects can be composed, and include a set of composition semantics to describe how aspects are going to be integrated. AOD models can be considered as UML extensions (this kind of model can be considered as part of Aspect-Oriented Modeling [1]). One of these extensions is Aspect-Oriented Design Modeling – AODM [52]. Another that is independent of UML is CoCompose [56].

Finally, there are several Aspect-Oriented Programming languages at the programming level. AspectJ, which is an extension of Java, is the most important of them. Also particularly worthy of note are JasCo [53], JAC [42], and JBOSS AOP [13], which are proposals based on managing crosscutting concerns on CBS, and DemeterJ [32].

3. A model to develop aspect-oriented systems at architecture level

In this section the model proposed to deal with the insertion of aspect at architectural level is presented briefly. Then, the methodological approach defined in association with the model is outlined.

3.1. AOSA Model in a nutshell

AOSA is a model to describe the software architectures of complex systems. Aspects are first-class citizens. In particular, they are considered as architectural components. In the AOSA Model, concerns crosscutting architectural modules are considered as architectural aspects. They can be functional or non-functional requirements identified at early stages or during system evolution and maintenance.

In the AOSA Model, system components – containing the system functionality – are like black boxes having required and provided interfaces. Aspects are architectural components characterized by their provided interfaces. The interaction between aspects and system components is made in an oblivious way, with aspects being context interdependent, thereby increasing their reusability. Aspects are injected into a software system by means of the pointcuts located in public interfaces of system components. Pointcuts referring to internal behaviour of system components are not considered.

Interactions in the system architecture are represented in two ways. On the one hand, this is by the connectors defined between system components (regular connectors). When aspects are injected into a system, these interactions do not change, except those connectors defined in association with the identified join points (to insert aspects) which need to be redefined. On the other hand, it is necessary to consider some new connexions which need to be established to make it possible to connect regular components and the aspectual ones. The AOSA Model proposes specifying this complex interaction to define the weaving by including new architectural elements: some coordinator components deal with the composition and coordination tasks considering the pointcuts and application conditions.

The weaving process is defined as follows: (i) A join point is a service provided by a system component; (ii) A pointcut describes a set of join points. It is also a predicate that matches join points. In this way, pointcuts represent the conditions to be satisfied at join points so that the aspects are applied. And (iii) advices are the services to be executed by aspectual components. Weaving conditions are specified externally to system components and aspectual components to preserve the context independence. Weaving can be executed after, before, or around the action associated with each pointcut. Also, an event can trigger the execution of more than one aspect.

The AOSA Model defines, in general, a coordinator component in association with each aspect. This component defines the weaving semantics determining, at run time, whether an aspect should be executed or not, depending on the condition values. Coordinators are context dependent and the software architect determines their structure during the architectural specification of the extended system. To reduce the coordinators’ workload – if complex systems are being considered – a new component has been defined: the supercoordinator (SC). This component is invoked by coordinators to check whether its corresponding aspect should be executed. To obtain better results a Supercoordinator Generator (SC_Gen) component is defined in such a way that when supercoordinator is invoked by a coordinator component, a new instance of SC is created by SC_Gen which only focuses on this call, and then this instance is destroyed.

Finally, composition actions are performed dynamically at run time, depending on the rules defined in association with aspect behaviour. If more than one aspect acts on the same point, the software architect ought to specify the proper execution priority.

3.2. AOSA methodology

In this section, we outline our methodological approach [37] to designing AO systems. Its main characteristics are based on the following premises:

- From an initial software system, one considers the possibility of carrying out its design taking into account the specifications of the problem. However, only the basic specifications to determine the base system design are studied. Those specifications which can be considered as aspects (because they cross-cut the base system components) are not examined at this stage (Section 3.2.1).
- When the basic specifications have been considered, the method proposes designing the base system and representing it by means of UML diagrams, particularly by use case and sequence diagrams. The architectural specification of the system should also be expressed by means of an ADL.
- After that, the specifications not previously considered are incorporated into the base system by following the Jacobson proposal [28]. This means that the aspects will be included in the base system diagrams. As a consequence, when the use case diagram is extended, the corresponding sequence diagrams are modified too (Section 3.2.2).
- The last step is to generate the architectural design for the new system. We provide an AO-ADL which makes this possible (Section 4).
Taking this into account, when the AOSA Model is followed, the method leads the software architect through the tasks of specifying and designing AO systems, and thus it is possible to extend existing (or new) systems – the base system – with new features which can be considered as aspects. The extensions are (or can be) concerns crosscutting several components of the base system. And AOSD concepts can be applied.

The aspects are identified at early stages (or at the maintenance phase) and are considered as design artefacts that describe their high-level behaviour. The aspects also become incorporated in an oblivious way. Therefore, one can manipulate them throughout the development process. The base system is built from independent components. Aspects will later be connected to the base system through a set of architectural connectors. Also, the model allows us to obtain an extended system from the architectural design phase, without changing the design components which constitute the initial one. As a consequence, the designs obtained remain clear, and reusability and evolution are easy.

Before an aspect can be considered as a design component and then incorporated into a system, it is necessary to identify and specify it by the description of its interfaces as well as its conditions of application. Then, interactions (architectural connectors) between the aspects and the components of the system can be defined taking that information into account.

In previous work [36], we proposed that aspect separation in a system can be treated as a coordination problem during software architecture. For this purpose, exogenous control-driven coordination models [4] can be considered to add these new requirements. The final goal is to acquire the execution of a prototype of the extended system (including the added aspects).

The AOSA Method is briefly presented here, it being the steps of the proposal defining our model to be explained in the following subsections. Fig. 1 represents a framework which functions in this sense. Ellipses with a light line represent UML specifications; ellipses in bold represent formal architectural representation (in an ADL). Ellipses having bold text represent the AO system both in the UML specification and in the architecture description. Arrows represent actions executed by the framework. The steps when using the framework are:

(1) The system is represented by a set of functional specifications expressed by its use case diagram and the associated sequence diagrams. In this way, the external point of view of the system is obtained. This is the base system. The base system architecture is also described by means of an ADL (1'). After that, in a second step, the behaviour of the base system has to be modified by adding the new functionalities (considered as aspects) which extend the base system. To do this, the aspects to be incorporated into the system need to be described. They are represented by Use Case Extension following the Jacobson proposal in [28]. Aspects are described as architectural components (1a) and also expressed in the ADL (1a').

(2) The specification of the AO system (extended system) is obtained by mixing the base system specification with the new requirements. The initial use case diagram is extended with the use cases containing aspectual specifications. These are incorporated as use case extensions. Consequently, the sequence diagrams become changed. In addition, extra information needs to be considered to enable aspectual insertion (application conditions, when, how, and where they will be applied, ...).

(3) Then, the AO system architectural description is achieved. It is expressed in AspectLEDA. This will allow the architect to check the consistency of the extended system. Then, the three inputs (1, 1a, and 2 in Fig. 1) are needed.

(4) Finally, this aspectual representation is translated into a regular ADL, which will allow a prototype to be executed.

To clarify these concepts, the following is a case study presenting interferences between aspects. The case study is inspired by the one used by Jacobson in [28]. We chose it with the aim of referencing a widely known case study, instead of one that would be closer to reality and hence more complex.

Consider a system which provides functionality to a Hotel Management. Specifically, consider the requirements for a Room Reservation facility which will be used by the hotel customers as well as its Restaurant Management facility that allows hotel customers to book a table in the hotel restaurant.

When a customer wants to reserve a room, its availability is checked; then, if it is available, a reservation is made. In addition, when a hotel customer wants to book a table in the restaurant, a query is made by means of the client utility which allows him or her to make the desired reservation. Two additional system requirements have been modeled as aspects. The first (Counter Aspect) is responsible for counting the number of rooms requests that could not be satisfied because the hotel was fully booked. Counter Aspect can also be used to count the tables requested but not booked. The second (Find Room Aspect) will perform a secondary search for rooms in other hotels belonging to the same hotel chain. This action must be performed whenever a room request cannot be satisfied in the hotel.

In order to achieve correct system behaviour, the Find Room Aspect must operate before Counter Aspect, thereby avoiding counting room requests that are finally satisfied (in a different hotel). However, the software architect may be unaware of this fact – either because they were forgotten (or unnoticed) when specifying the systems requirements or because the two requirements were introduced at different times and the obliviousness principle con-

---

**Fig. 1.** The aspect-oriented architectural framework.
tributed to hiding the issue. It would thus be desirable to be able to
detect this problem as early as possible and avoid carrying it over
to the implementation phase.

A waiting list management can be used in association with the
two initial requirements. Applying this new requirement is the
same as including an aspect (Waiting List Aspect) associated with
the Room Reservation and Restaurant Management facilities. This
new feature allows clients to be included in a waiting list when
they make a query and there is no availability.

Given this type of scenario, there are two fundamental rea-
sons for our proposal of AspectLEDA: first, to provide support
for architectural descriptions in which aspects are considered,
and second, to provide a basis on which to check whether the
described architecture will support the behaviour expected in
the system.

3.2.1. Specifying the system and creating the architectural model

The first step of the AOSA Model proposes considering the develop-
ment process of a system from its initial phases of: requirements
specification and design at high level, without considering aspects.
A detailed specification of the system is obtained using the use case
diagram. A full description of each use case is made to provide the
system documentation. For each use case, a sequence diagram is
defined. Components constituting the system appear in it, and
the interactions among them are also represented.

After the detailed specification of the system, its architecture is
defined. Design components need to be specified by describing
their interfaces. The interactions are defined by considering the
relationships among components and must be established during
the system specification and expressed in the sequence diagrams.

In order to formalize the design model, considering the archi-
technical decomposition in which components and their interac-
tions are identified, a regular ADL is used. As was mentioned
above, the ADL considered was LEDA due to its strong formal base
(pi-calculus) and because it enables the system prototype to be
executed from the architecture design, although other languages
could also be used. A base system description was obtained, in
which the crosscutting concerns are not yet considered. It is pre-
seated in Appendix B.

3.2.2. Adding aspects by redefining the base system architecture

After defining the base system, aspects can be added to obtain the
extended system, including the crosscutting concern, as follows.

The base system architecture is redefined. For this, the new as-
pects should be included in the system specification. The aspects
need to be described and then their descriptions are added to the
base system use case diagram. First, each aspect to be added is consid-
ered as a special kind of use case: a use case extension, each of which
extends the initial system specification. In the new diagram, the
interaction points, where the aspects crosscut with the base system
use cases, must be determined. They are the extension points.

Also, how each aspect interacts with the system must be de-
defined (its application conditions). For each extension point, some
information is needed, and will be stored in order to be used later.
This information is the name of the extended use case and its
description, the name of the extending use case, the extension points
list, the event type triggering the aspect execution, the conditions
of execution to be satisfied and when they will be executed (before,
after, or around the action triggering the event).

For each use case extended with an aspect, the sequence dia-
gram is redefined. This new diagram contains components in the
initial use case to be extended as well as the aspect component/s
to be added. Fig. 3a represents the use case diagram for the ex-
ample when the aspects are incorporated into the system. Fig. 3b rep-
resents the sequence diagram for the main success scenario of the
Reserve Room use case extended with the Counter aspect, when a
non-AO-approach is followed. In this case, one observes that the
ResRoomHandler component accesses the Counter component. If
one acts in this way, the ResRoomHandler code will need to be mod-
ified to make it possible to access the Counter component. In this
case, the obliviousness principle is not observed. It would be better
to apply an approach that allows the system behaviour to be chas-
ged without its components being modified. AO-approaches allow
this. Fig. 3c shows the sequence diagram for the same scenario.

However, now, the interaction between the aspect component

---

**Fig. 2.** (a) Initial system use case diagram. (b) Sequence diagram associated to reserve room use case. (c) Sequence diagram associated to restaurant handler use case.
and the system is not defined yet. Before specifying the nature of the new interactions, it is necessary to define a new element, which we call Aspect-Manager. This new element is now considered as a component that is not defined in detail, but will later be the Aspect Level in the model, containing the aspect components and manages their interaction with the base system.

The information associated to the extension points identified in the use case diagram should be propagated to the corresponding sequence diagrams. For this, a table is defined for each tuple: extension point, aspect, and the corresponding design component. These three elements jointly define one point in a design component in which an extension point is detected. It is called an Insertion Point. This concept is close to the join point concept, but at a higher abstraction level. For each extension point, aspect, and design component, one or more insertion points can be defined. In the following sections, details of how this information is obtained have been omitted in order to make the explanation as clear as possible. See [37] for more details.

It can be observed that when the AOSA Model is followed, an architectural component (Aspect-Manager) which contains the aspectual component and manages its interactions is included. The Aspect-Manager component intercepts the call methods defining the Join Points, expressed as Insertion Point. The detail of the interactions between aspects and components is specified. As a result, a structure which contains the required information is obtained – the Common Items structure. Table 1 presents its contents for the example, with a description of each field. In Fig. 3c, the retrieve() call method – invoked by the ResRoomHandler component – is intercepted by the Aspect-Manager1 component. It analyses the application conditions and when the aspect should be applied (before, after, or around the call method). In the case of the example, the aspect application is after the intercepted call method. Consequently, Aspect-Manager1 resend the call method to the server component and then checks whether the answer (availability) is positive or negative. In the case of a negative answer, the aspect condition is satisfied and it should be executed, i.e., the counter will be increased. After that, control is returned to the ResRoomHandler component with the result of the operation started by it (Retrieve). In Fig. 3c, aspect invocation is represented by means of the auto-delegation symbol at the life-line of the Aspect-Manager component. Note that the aspect invocation goes after the request of the retrieve() operation.

Finally, both components defining this piece of the system behaviour are oblivious of Aspect-Manager1 component activity.

Now one can consider extending the system with the Waiting List aspect, when it is defined in association with the Restaurant Handler utility. Studying the corresponding use case allows one to obtain a sequence diagram similar to the previous one (Fig. 3d). However, now a different Aspect-Manager component is included (Aspect-Manager2). It will manage the Waiting List aspect to be inserted here. Table 2 presents the Common Items structure for this aspect.

The same process should be followed for the third aspect. Fig. 4a represents the sequence diagram when a non-AO approach is followed. Fig. 4b shows the corresponding sequence diagram following the AO approach. As can be seen in the sequence diagram and exploring the corresponding Common Items structure (Table 3), the two aspects affect the same point. However, they must operate in some particular order. In the example, they will operate in the same order as they were included. In this case, the same Aspect-Manager component manages both aspects.

On conclusion, when an AO-approach in general, and the AOSA Model in particular, is followed, an aspect can be injected into a system observing the obliviousness principle. Including more aspects and use cases involves applying the same considerations.

---

Footnote: 2 If more than one aspect is applied at the same Insertion Point (or Join Point), it is necessary to consider the order of operation. The software architect has to assign some kind of priority among them.
3.2.3. Architectural support to deal with aspects

The definition of the Aspect-Manager components allows design components of the base systems not to change when the aspects are added. To manage this during architectural design, a two-level architecture is proposed: the Base Level in order to define the base system, and the Aspect Level in order to include the new components supporting the added aspect. This makes it possible to define a new workspace independent of the base system architecture. The Aspect Level represents the components inside each Aspect-Manager component and their relationships (Fig. 5). Thus, the Aspect Level is constituted by a set of aspects. Each aspect is defined as an architectural component, with an interface providing the operations that determine its operability. In addition, and in order to make the complex interactions between the two levels possible, it is necessary to manage some information which has been represented as a new kind of architectural connector. The information managed is the following:

- The Common Items (C-I) structure, which contains all the information deduced from previous steps (Table 1). This detailed specification of the system allows its structure to be obtained entirely defined. This information is deduced by the designer when studying the aspects to be included, and takes the sequence diagrams into account.

- A set of rules specifying the characteristics of the interactions between the Aspect Level and the components in the underlying level, providing the actions to be executed when applying the aspects. An insertion point (IP), defined above, can be affected by more than one aspect (as in the case study). As a result of this, more than one rule or one complex rule (con-
executing an aspect, the base system behaviour can change. In order to solve the problem of coordinating aspect and component execution, an exogenous control-driven coordination model is considered. The one selected was Coordinated Roles (CR) [35], based on the Event Notification Protocols. CR was selected because it adapts to the initial concepts of our model: using coordinator to monitor system components in a way that is oblivious to them. Other coordination models could be used, but CR is the closest to the AOSA Model specifications. When an event is detected by a CR coordinator, this executes the coordination actions associated to the events. As the extended system can become complex because of the number of aspects to be included, Control Process consists of [36]:

- Several “Coordinator” architectural components. Each is defined for each design component having one insertion point. A coordinator can manage several aspects associated to the same IP. Its behaviour is defined by the rules which are checked in another component: SuperCoordinator.
- SuperCoordinator is defined as an architectural component. It is unique for the whole system. It is a coordinator of coordinators. Its definition is considered necessary to facilitate the coordinated execution of the coordinators when several aspects are being included and are applied at different Insertion Points.

Fig. 6 represents the new connector structure which contains the elements needed to interconnect the base system components and the aspects to be injected. In Fig. 6a, only one new connector structure is represented in which Coordinator1 is defined to enable the FindRoom and Counter aspects to have coordinated execution. Both are associated to the same IP as can be seen in the corresponding Common Items structures (Tables 1 and 2). Fig. 6b shows all new interactions for the case study. There, Coordinator3 is defined to coordinate the Waiting List aspect when this is applied to the Room Handler component. Likewise, Coordinator2 and Coordinator4 are, respectively, defined in association with the Waiting List and Counter aspects to provide service to the Restaurant Handler facility.

- At runtime, a dynamic structure (BackBoard) is used to store the triggered events (waiting to be treated) during the system execution. The information stored for each event is the event type, the insertion point name (IP), the name of the component affected by the IP, and the values of the conditions. The events and their associated information are written to it by the Coordinator component when they are detected, then read by the SuperCoordinator which analyzes this runtime information searching for a match with the value of the conditions defining the coordination policies (rules).

The most interesting elements in the Aspect Level are the Coordinator components, which are defined in association with both a component (performing server functions) and one or more aspects. When an event (call method) is triggered, the aspect is executed if certain conditions are satisfied. A Coordinator acts only if the detected event has been associated to the event coordinated by it. Its behaviour is defined through the information deduced after applying the steps of the model and stored in the Common-Items structure. A more detailed Aspect Level description can be found in [37].

3.2.4. Architecture description of aspects

An aspect is an element representing extra behaviour to be added to a system, which is executed at a specific moment dur-
In order to summarize what has been described until now, Fig. 7 represents the elements of the activity diagram when synchronous messages are received. One observes that Coordinator manages the interactions between a server component and the aspect components.\footnote{The case is similar for asynchronous messages.} Finally, our architectural model enables an aspect to be eliminated from the extended system when it was inserted by means of AOSATool/LEDA.

```plaintext
1 component Extndsyst {
2   composition
3     baseSystem: System;
4     counter: Aspect;
5     findroom: Aspect;
6     waitlist: Aspect;
7     priority counter/findroom;
8     attachments
9     baseSystem.RoomHandler.RetrieveRoom("room")
10    <<roomroom, RMA, NoAvail, after>>
11    counter.Counter();
12    baseSystem.RoomHandler.RetrieveRoom("room")
13    <<roomroom, RMA, NoAvail, after>>
14    findroom.Findroom();
15    baseSystem.RoomHandler.Query("room")
16    <<roomroom, RMA, NoAvail, after>>
17    waitlist.Waitliststop();
18    baseSystem.RestaurantHandler.Query("room")
19    <<customerapp, RMA, NoAvail, after>>
20    waitlist.Waitliststop();
21    baseSystem.RestaurantHandler.RetrieveRoom("room")
22    <<customerapp, RMA, NoAvail, after>>
23    findroom.Findroom();
24 }
25 component System {
26   interface
27     none;
28 }
29 component Aspect {
30   interface
31     role: RoleAspect;
32 }
33 instance extsys: Extndsyst;
```

Fig. 9. Case study architecture in AspectLEDA.
of the latter. In order to remove an aspect, the corresponding elements and the associated information must also be removed from the Aspect Level. In this case, it is necessary to study how the extension needs to be modified.

4. Describing the extended system by using AspectLEDA

After the aspects have been added, the next step of the method is to describe the extended system in terms of the ADL. We define the new A0-ADL (AspectLEDA) in order to express AO concepts at the architecture stage in a formal way, which is also based on a regular ADL, LEDA [14], which is briefly overviewed in Appendix A. In this section, the characteristics of the new language are presented, as well as how an extended system can be obtained.

Describing the extended system in terms of the ADL implies enlarging the base system description by adding the Aspect Level elements and the specification of the new interactions. The new elements to be included lead us to consider a set of high order instructions to define the new Aspect-Oriented Architecture Description Language: AspectLEDA.

In order to design the extended system, the architect has to define the system architecture by means of the information deduced when the AOSA Model is followed. This information is stored in the C-S structure and in the rules. In addition, the elements managing the interactions (Coordinators and SuperCoordinator) with the base...
**Fig. 12.** (a) Regular Coordinator structure in LEDA. (b) A Coordinator structure in LEDA, for more than one aspect.

system are also included in the architecture of the new system. AspectLEDA is supported by the Aspect Oriented Software Architecture Tool for LEDA (AOSATool/LEDA). Fig. 8 shows its main screen. The architecture can be obtained in two ways. The first allows the architect to obtain the extended system under guidance by the tool (Section 4.1) and then it is translated into the (extended) equivalent LEDA architecture (5), and then it is translated into the (extended) equivalent LEDA architecture (6). Appendix B presents the full extended system code in LEDA for the case study after following the process.

Because the ADL allows it, the resulting extended system can be executed in this way to acquire the prototype of the new system (Section 4.4).

4.2. AspectLEDA: Detailed description

When the architect wants to define the architectural description of a system extended with aspects, he or she can follow the AOSA Model to obtain the information of the C-I structure. Also known is the name and the architecture description of the system to be extended and the name of the aspects which extend the system. After that, the architect can define the architecture of the extended system by using AspectLEDA. This language has the following elements and characteristics:

(A) The elements constituting an AspectLEDA architecture are the following (Fig. 9):

(i) A system in AspectLEDA is defined as a composed component – composition section – constituted by two kinds of element: system (describing the base system architecture) and aspects (specifying the aspects which are to be applied extending the base system). Both are defined as architectural components:

- basicsystem is the name of the base system to be extended. The basicsystem component will be an input file when the extended system is going to be generated and it contains the architectural description.

- aspect (counter, findroom, waitinglist) is the name of each aspect to be inserted.

(ii) The interaction between the system element and the aspects is described in the attachments section. Some parameters describing the new system architecture have been included between the attached elements representing the needed information (from the C-I). Its general format is:

```plaintext
Basename_Component_name_Provided_operation << Client_comp, Event, Cond, When >> 
Aspect_name_Aspect_operation;
```

where:

- Basename is the name of the system being extended. It is obtained from the base system description.
- Component_name is the name of the component providing the Provided_operation (Component field in C-I).

The first action is for the architect to select the architectural description of the base system expressed by means of the ADL – it can be stored in a previously created file (1) – as well as the necessary parameters (2). This information is known by the architect because it is in the C-I structure.

After that, the aspect information should be inserted (3). The name of the aspect/s, the triggering event type, and when the aspect should be applied, also need to be introduced. Then the code defining the external behaviour of the aspects is inserted.

The third step to be followed is to include the external behaviour of the coordinators (4). This information has previously been included in a file, and should now be inserted by the tool in the corresponding place of the architectural description of the extended system which is being built. Then AOSATool/LEDA automatically generates the extended system first in AspectLEDA (5), and then it is translated into the (extended) equivalent LEDA architecture (6). Appendix B presents the full extended system code in LEDA for the case study after following the process.

In this section, it is shown how to obtain the extended system by using AOSATool/LEDA. Its architectural description in a selected general purpose language (LEDA) can be automatically generated from the base system description considering the information in the C-I structure. The tool leads the architect through the actions to be taken in order to obtain the architectural description of the extended system (Fig. 8).
• Provided_operation is the "IP" element in the Common Items structure.
• Client_comp is the name of the component requiring the Provided_operation ("Client Component" in C-I).
• Event, Cond, and When, respectively, represent: the kind of event associated with the Provided_operation, the condition to be satisfied, and when the aspect should be executed. This information is taken from the C-I structure.
• Aspect_name, and Aspect_operation are the names of the aspects attached to Component_name in the new interaction, and the operation provided by it. This information is taken from the C-I too.

In the figure are represented the attachment instruction for coordinator1 (lines 9–14) and coordinator3 (lines 15–17) from Fig. 6. Then, two more sentences are necessary to fully describe the extended system in AspectLEDA: that associated to coordinator2, lines 18–20, and that associated to coordinator4, lines 21–23.

(iii) Finally components constituting the extended system and their interfaces are included in the AspectLEDA architecture description.

(B) These instructions allow the injection of aspects into an existing or a new system. As was shown, aspects are considered as first-class entities during architecture design. They are also added without the affected components to the base system being modified, maintaining the obliviousness principle. When two or more aspects need to be injected into the same IP, the architect should use the priority clause to force a specific order between aspects (line 7 in Fig. 9). This is an optional clause and its general format is:Priority aspect_HPrior > aspect_LPrior; (for two aspects)
(C) The architectural description of the extended system in AspectLEDA is translated into an equivalent LEDA architecture by parsing the AspectLEDA file with the corresponding lexical and syntactical analyzer. The translation process is described in following section. Appendix B presents the resulting code.
(D) Likewise, the extended system behaviour can be known during the architecture design phase by simulating the execution of a prototype. This is described in Section 4.4.

Finally, it is possible to demonstrate the properties of the extended system architecture due to the LEDA language's formal basis (π-calculus). Thus, a formal system specification can be obtained which can be used as the input for model checking tools such as the ABD tool that will explore the system’s properties allowing the detection of problems. This feature allows one to perform more exhaustive system correctness checking than with just the prototype execution. For the case study, the formal π-calculus specification of the AO architecture (expressed in LEDA) is shown in Fig. 10.8

4.3. Description of the AspectLEDA translation process

The internal process of translating the architectural description of a system expressed in AspectLEDA into LEDA by means of an interpreter – the "LEDA generator" – can be described by following two phases (Fig. 11). In order to help the architect to use this utility, a GUI has been developed.

The LEDA generator (1 in the figure) is made up of two main elements: the “AspectLEDA Interpreter” (1a), which scans AspectLEDA files and recognizes the AspectLEDA code inside; and the “AspectLEDA-LEDA translator” (1b), which translates (generates) the extended system LEDA code from the “AspectLEDA Interpreter” output into LEDA code. Each element does one translation process phase.

(1) AspectLEDA interpreter (1a) operation. This element has two inputs:
(a) A set of parameters which define the extended system in a particular way. They are provided by the user interface and are the C-I structure proposed in the AOSA Model. The AspectLEDA interpreter fills out an AspectLEDA architecture template by using the introduced parameters, and then an AspectLEDA config file is created.
(b) The software architect can complete this phase without using the AspectLEDA interpreter by editing him or herself the AspectLEDA architecture template to obtain the AspectLEDA config file, thus creating the AspectLEDA architecture.

(2) The second phase of the AspectLEDA translation process is carried out by the AspectLEDA-LEDA translator (1b). This utility has four inputs:
(a) A LEDA file having the architectural description of the system which is being extended (base system).
(b) A LEDA file containing the elements defining the new interactions (Coordinator and Super-Coodinator).

Fig. 12a shows part of the Coordinator code in LEDA for a regular coordinator which manages just one aspect. Fig. 12b shows a part of the coordinator code when more than one aspect (two in this case) are applied at the same point.
(c) A LEDA file having the architectural description of aspect components which are extending the base system.
(d) The AspectLEDA config file obtained after executing the first phase of the translation process. It contains all the useful data to generate the LEDA extended system.

During the first phase, the AspectLEDA generator determines the filenames and the parameter values needed to generate the extended system architectural description in the second phase. To do so, AspectLEDA-LEDA translator appropriately combines the code in the LEDA files in order to obtain the new system architectural code in correct LEDA. An error report is created to inform the software architect if any errors appear when AspectLEDA interpreter or AspectLEDA-LEDA translator are executed. Fatal errors stop the execution; warning errors allow the process to continue.

4.4. Obtaining and evaluating a prototype

After AOSATool/LEDA translates the AspectLEDA architecture into a LEDA equivalent, Java code can be obtained if the LEDA interpreter is executed. In this way a prototype of the extended system execution is obtained. This is simply because the LEDA language can be translated into Java. After LEDA-to-Java translation, LEDA components and roles become Java classes. Consequently, components in base system are now Java classes, and the added aspects (defined as LEDA components) are now Java classes too. Also, as was noted above, aspects are defined context independent. Context dependent information is kept in the elements defined by the model (coordinators) and the new interactions. These elements were coded first in LEDA and then translated into Java, in order to obtain new Java classes.

8 Detailed information on LEDA ADL and its π-calculus expression can be found in [15].
A graphic tool (Translator/Prototyping LEDA-Java Tool) helps the architect to perform this task. Fig. 13 shows, by way of example, one of its windows, showing a part of the automatically generated Java code obtained after the translation process. Also, a simulation of the execution of the prototype generated is shown at the bottom of the figure. Although methods belonging to classes representing components have no code in that Java program, a non-deterministic return is produced from all their possible outputs. Code will be injected at the implementation stage.

The execution of the architecture will provide the architect with information about the interactions between the aspects and the system, allowing a check as to whether its final behaviour matches the expected one. In the Room Reservation case study, the execution of the architecture would show that the behaviour of the system is not correct since the reservation request satisfied by a room in a different hotel is counted as not being satisfied as well. This is because the two aspects affect the same point and are applied in the incorrect order. Once aware of this mistake, the architect can perform the opportune corrections, making the aspects operate in the correct order, before the mistake is carried over to later steps in the development process.

5. Related works

Aspect orientation is a new discipline which is attracting growing interest. The whole life-cycle is considered by the AO paradigm; and each model or research line has its own perspective. The closest to our research line are those proposals mentioned in the AO requirements engineering and in AO architecture paragraphs, in which aspects are defined as relevant elements, but in a different way. In this section, some studies performed at the AO Software Architecture stage will be described. In particular, the section is focused on the AO-ADLs associated with those proposals, and they are studied in comparison with the AspectLEDA language. Some of them are extensions of regular ADLs, but others are new languages. All the proposals deal with the introduction of aspects into ADLs in different ways: as a component, as a connector, or as architectural views, and with either a symmetric or an asymmetric approach.

In these terms, the alignment of AspectLEDA with those criteria is as follows. First, it is based on a regular ADL (LEDA), which is extended with aspect support. Second, its model is an asymmetric one, which provides the advantage of supporting system evolution by the addition of aspects in a natural way. And third, aspects in AspectLEDA are components which remain separated (not weaved) throughout the process, facilitating the traceability of properties at any stage of the design task. As additional properties it must be mentioned that:

- It is supported by a tool that assists the architect during the design by providing him or her with the possibility of generating an executable Java simulation of the architecture.
- A pi-calculus specification of the architecture can be generated, allowing AO-Architecture checking and facilitating the evolution of a system while preserving its features.
- It allows the generation of Java code to obtain a prototype of the designed system, and incorporates dynamic reconfiguration by means of the reflective architecture model.
- A special feature is that coordination is provided by a special kind of component (coordinator) which is used to manage the coordination tasks as well as the aspectual composition.

The CAM/DAOP-ADL [48] method is a component and aspect-based approach that supports the separation of concerns from design to implementation. This approach defines the Component-Aspect Model (CAM), the DAOP-ADL [47], and is interpreted by the DAOP platform, but it is independent of any programming language. The information provided by CAM during the design is expressed in terms of the DAOP-ADL language. This is a new language which does not extend any existing ADL. Like AspectLEDA, it takes the asymmetric approach and aspects are described as components. DAOP-ADL is based on the definition of XML schemas. The architecture can then be generated instead of the pi-calculus specifications of AspectLEDA. Indeed, the pi-calculus specifications of AspectLEDA provide a stronger formal basis that can be used with a wide range of tools to perform model checking tasks.

Prisma ADL [44] is a highly evolved language created to describe systems developed in the framework of the Prisma architectural model [43], in which aspects are abstractions different from connectors and components, and are used to define the structure and behaviour of the architectural elements. It takes the symmetric approach in which a component is described as a set of aspects, an explicit weaving and a common interface. While this symmetric approach provides Prisma with a very natural way of dealing with crosscutting concerns, the asymmetric approach of AspectLEDA facilitates system evolution through the addition of aspects.

AspectualAcme [5,9] is an AO-ADL extension of the Acme language [25] in which the concept of aspectual connector is defined to connect aspects to base components, instead of using components to play the role of aspects. AspectLEDA, based on LEDA, has no connectors, allowing simpler architectures to be produced. Also, in contrast with AspectLEDA, AspectualAcme does not support code generation and no system prototype can be obtained.

Pilar [18], proposed in the Marmol model [20], is a dynamic and reflective ADL which can be extended by inserting aspects as components (meta-components) at a meta-level, and then binding them to the base architecture by adding reification over the base system. This model has great expressive power and flexibility. Pilar also provides a strong formal basis for checking system properties. Since the LEDA architecture produced for an AspectLEDA description is based on a reflective schema, it has some similarities with the model proposed by Pilar. However, AspectLEDA provides a special syntax for aspects which facilitates the architect’s work by hiding the description details of how the execution of the aspect functionality must be weaved with the execution of the base system functionality.

AO-Rapide [40] is an architectural model based on the same principles as AspectLEDA. It is an extension of Rapidé ADL [33], taking an asymmetric approach with aspects being components. In contrast to AspectLEDA, AO-Rapide is based on the Reo [4] channel calculus, which also provides it with a strong formal basis. It is less evolved than AspectLEDA, however, and has no tools supporting code generation.

The TransSAT method [6,7] is a framework for managing the evolution of software architecture using AOP principles, allowing systems to be described incrementally. Concerns are modeled as components and weaving is defined by means of adapters and weavers which, respectively, define integration rules and coordination rules. Consequently, adapters and weavers describe the integration of the core architecture and their extensions by identifying the join points in the core architecture and defining the pointcuts at adapters and weavers. This framework uses an XML-based ADL. As a result, and although like AspectLEDA it can generate the code implementing the architecture, no formal specifications are generated.

FAC [45] is an extension of Fractal [46], integrating ADL, CBP (Component-Based Programming), and AOP. It introduces the aspect component concept and aspect binding which connects a component and an aspect. Unlike in AspectLEDA, the existing bindings
are not modified. Special weaving interfaces are included to handle a component’s exposed join points. An XML-based ADL is used to describe software architectures with crosscutting concerns. While this is a similar approach to ours, AspectLEDA allows one to obtain a pi-calculus specification of the architecture to use as the basis for model checking.

Asbaco [34] is based on a “microcomponent” model which allows a component’s non-functional features to be modeled. A microcomponent designates a component that implements control functionality. Asbaco microcomponents are related to the notion of regular components: they have client and server interfaces allowing bindings between components. Based on this microcomponent model, Asbaco defines the concept of component aspect as a set of controller part extensions. Unlike AspectLEDA, this model does not modify the existing architecture. However, it extends the component controller to perform interception and advice, which AspectLEDA does by means of independent elements (coordinator components).

FuseJ [54] is a programming language for component-based software architectures on a Java Beans component model. There are no separately defined aspects in this language. Instead they are implemented as regular components. Coordination tasks are performed using AO primitives to define the pointcuts and advice (to specify the weaving process). The purpose of FuseJ is programming aspect-oriented applications on a Java Beans basis, whereas that of AspectLEDA is the development of AO software architectures.

In comparison with the foregoing:

- Our approach refers to the use of conventional components to play the role of aspects. That is, aspects are defined as regular components, instead of providing some additional abstractions (like views or slices). This has the advantage of maintaining a uniform model.

- The proposed architectural model follows an asymmetric model (on the contrary, Theme, MDSoc, and Prisma follow a symmetric model). The model defines a two-layered coordination structure to support aspect weaving.

- We propose the definition of an AO-ADL with a strong formal base, which permits the generation of a system prototype (on the contrary, MDSoc uses HyperJ, Theme/UML extends the UML meta-model, and CAM/DAOP defines an ADL in terms of XML).

- AspectLEDA allows the architecture obtained to be evaluated and checked. However, not all the above mentioned works allow this.

6. Conclusions and future work

In this paper, AspectLEDA, an AO Architecture Description Language, has been presented. Aspect separation is managed at the architectural level by defining a two-layered architectural structure to support aspect weaving and by treating aspect separation as a coordination problem. The AOSA Model, on which AspectLEDA is based, provides systems with an architectural description allowing their behaviour to be changed at the design phase by adding (or eliminating) logical restrictions without changing the components that constitute them. Only some interactions need to be modified. The AOSA Model gives architectural dynamism to the system in order to add (and remove) components and interactions. Systems are built from design components describing their functional behaviour, and from a set of design components (Coordinators and SuperCoordinator) executing aspect policies. It is then also necessary to define interactions between new and old components. They are transparent to the base system. This is possible due to the reflexive capabilities of the model which defines a base level containing the initial system (base system) and a meta-level which contains the new elements (aspects and coordinators in the model). Moreover, interactions between the two levels are transparent to the base system.

When following the steps of the model, systems designed in AspectLEDA have a clear design and highly cohesive components, because functional and aspectual components can remain separated since the generated LEDA code allows it.

From the Model, an extended system architectural description in AspectLEDA is obtained. Then, this description is translated into a...
pure LEDA architecture by means of a two-phase process. Next, this LEDA code can be translated into Java, and thus a prototype simulating the extended system execution can be obtained. The prototype also has dynamic and reflexive capabilities because so does Java.

In order to facilitate the entire extension process, the defined design framework provides the architect with a toolbox. The tools are the following ones:

- A user interface, which leads the software architect through the translation process from an AspectLEDA architecture to a LEDA architectural description. Information needed to generate the extended system is easily obtained by means of this tool.
- A graphic tool that allows Java code to be generated from the previously obtained LEDA architectural description.
- A prototyping facility helps the designer to simulate the execution of the Java code obtained, and thus a prototype of the extended system is obtained to verify the extended system's behaviour at design time.
- Another windows environment tool allows the architect to edit and modify the LEDA files in order to introduce changes in the system generated in LEDA. It is also possible to edit and change the LEDA grammar if the designer wants (or needs) to do so.

We are currently working on integrating these tools into a unified environment.

AspectLEDA provides the definition of join points intercepting the behaviour of components, and permits the access to components through its public interfaces. The method defines the capacity for aspects to crosscut the behaviour of components before, after, or around the actions associated to the events triggering their execution.

Finally, the characteristics of the extended system are expressed by means of the use of both an ADL, with a strong formal base, and a coordination model, so that the system's properties can be analyzed and verified.

One of the main conclusions of this research is that existing systems can be adapted to new situations by adding new requirements or unanticipated changes by considering them as aspectual features. This is because the incremental development process, proposed here to design extended systems from initial ones, allows architects to enable a system with new requirements by considering them as aspects. Particularly, at the architectural design level, the added requirements can be treated as aspectual components, and then it is possible to apply the steps of the model. The inclusion of new requirements means a minimum impact on the architecture of the existing system, and implies no change in the current design components. Thus, they will have easy evolution and adaptation to unanticipated changes, and low-cost maintenance: The tasks of evolution of a system are applied to its design specification and software architecture, instead of to the implemented classes. To evolve a system, it is proposed to act at the architecture level. The corresponding code is then automatically generated (by the tool) and classes become redefined. Consequently, system redesign is easy. Finally, the proposal is platform independent.

On the basis of our experience, we can conclude that using AspectLEDA and the architectural model on which it is based to define the system architecture helps designers to manage better structured systems. As a consequence, dealing with changes in stakeholders' requirements specifications is not as hard as with conventional architectures. We have also experienced the relative ease of extending an existing system with AO architectures.

After developing AOSA Model, the Methodology and AspectLEDA language, we currently are doing a comparative study and evaluating the obtained results (Java code) after applying our model with the results obtained when other methodologies or approaches are applied. We began this comparative considering the associated code to the extended system which we obtained from the translation of AspectLEDA with the obtained results applying another AO-ADL developed in our research group too [49].

Acknowledgments

We thank to the referees for the useful comments on this paper and the suggestions that they gave.

Appendix A. An introduction to the LEDA language

LEDA [14] is an ADL for the description and verification of structural and behavioural properties of software systems. It is based on pi-calculus, a process algebra that can express mobility in a natural way, allowing the specification of dynamic architectures. In [15], it is demonstrated that the formal basis of the language makes it possible to simulate the execution of the specifications and their analysis, in order to determine the compatibility among the components (their interfaces), ensuring that the system is free of blockades. The characteristics of the language also facilitate generating the implementation of a system from its architectural specifications, preserving the aforementioned properties. ADLs based on formalisms do not usually allow one to obtain an implementation conserving the properties of the architecture, although they enable designing it without errors. On the contrary, LEDA provides the automatic generation of components, from the description of its interfaces in LEDA. As a result, a prototype of the system is acquired, which can be used to obtain the final system. The prototype is implemented on Java, although any OO language could have been used.

The LEDA language does not distinguish, at language level, between architectural components and architectural connectors. That is, architectural connectors are specified as regular components in the language. Thus, the language is simpler. It is structured in two levels: components and roles. The components represent modules or software units. Each one provides a certain functionality, and they can be simple or composed by other components. The roles describe the behaviour of the components and the interaction protocols established among them. Each role is a partial abstraction, which represents the behaviour offered by the component to its environment as well as the behaviour required by those connected to it. The specification of the roles is carried out by means of pi-calculus. The communication is also carried out in pi-calculus by means of connections, which are bidirectional channels connecting some processes with others (connections can be shared among several processes). Roles are also used for validating, prototyping, and executing the architecture.

In addition, the LEDA language has mechanisms to dynamically create and interconnect components, as well as to obtain components by means of the inheritance and the instantiation of architectures. Other mechanisms in LEDA are: an adapter mechanism to adapt a component to an interface non-compatible with it; and role and component parametrization. This characteristic makes it possible to consider LEDA specifications as generic architectural frameworks which can be extended and reused, adapting them to new requirements.

LEDA follows a generative proposal allowing code generation from a language of a higher level to an implementation language: the designer works at the specification level in an architectural language and, from it, the implementation code is generated. Details not defined during the specification should be completed during the implementation phase. The generation scheme followed by
LEDA enables building a prototype of the system from its components and roles described in the language. For this, an association between each component and its roles is defined, and these are interconnected following the architectural definition of the system. The generative approach allows the designer to obtain the most appropriate architecture of the system. Its specifications expressed in LEDA are translated, by means of a generator, into Java as an executable prototype of the system, which is also the base for its complete implementation. The properties of the original specification are conserved and the generated code is clear and readable, being modifiable by the designer to complete the component computation code.

In order to obtain the final system, an evolutionary development is followed. First, to produce an initial prototype only the specification of the component roles is necessary, with the description and implementation of the computations being added later by the designer. Incomplete versions of the final system can also be tested and executed when the implementation of methods is being completed.

Components and roles are defined as implementation classes. The former encapsulate data and computations, and the latter implement the interaction protocols in which each component is involved. Components are translated into passive classes, in the sense that they do not invoke methods of other objects, but rather their methods are invoked by the roles. Roles in LEDA, which describe the behaviour of the components, carry out the functions of connecting a component with the rest of the system. They also carry out the operations of reading and writing by the appropriate connections, and in the correct order. They invoke the functions which are implemented by the components as well.

In LEDA, the architecture of a system is defined as a compound (complex component). Architecture is expressed by means of the relationships between the components constituting it. This is expressed by a set of connections (attachments) among the roles of the components. The interconnections are carried out by means of shared connections. According to this, the components of a system only read or write in a set of connections without knowing who has access to these connections at that moment. One advantage of this way of carrying out the interconnections is that the components do not have information on the rest of the system, which facilitates their reuse. The connections, which define the communication among roles, are also specified by means of an implementation class when the prototype of the system is generated. This class is independent of the implementation of the components and roles.

Appendix A.1 LEDA syntax at a glance

Components and roles are specified in a modular way. The Figs. A1 and A2 represent the code associated with them.

Several kinds of connections can be defined in LEDA: static, multiple, shared, and reconfigurable. The corresponding syntax is:

**Static Connections:**
```
compl.role1(names1) <> comp2.role2(names2);
```

**Multiples:**
```
compl[].role(names) <> comp'.role'[]([names']);
```

**Shared:**
```
compl*[].role(names) <> comp'.role'[ ]([names']);
```

**Reconfigurable Connections:**

(a) compl.role1(names1) <> if condition then comp2. role2(names2);  
    else comp3.role3(names3);  
(b) compl.role(names) <>  
    case condition1 then compl.role1(names1)  
    case condition2 then comp2. role2(names2)  
    default compl.role(names)
```

The Refinement of an architecture can also be expressed. This allows for the definition of a derived component as an instance of the class ClassComponent.
Appendix B. Basic and extended system representation

Fig. A3 represents the code of the base system architectural description for the example, in LEDA. No aspects have been considered yet. The code in frame A4 represents the automatically equivalent LEDA architecture generated for the AspectLEDA architecture of the reservation system after both aspects have been included.

As one can see, new code lines have been included. This will permit the execution of the prototype of the extended system without components constituting the basic one changing. Lines after attachments sentence are defined associated to roomhandler component (as client) and roomhandler (as client). In this case two aspects (counter and findroom) are associated to the same point.

```
component Extnddysyt {
    interface none;
    composition
        resroomhandler, customerapp: Client;
        roomhandler, restauranthandler: Server;
        counter: Aspect;
        findroom: Aspect;
        waitinglist: Aspect;
        ccoor: Coordinatortwice;
        coor2: Coordinator;
        coor3: Coordinator;
        coor4: Coordinator;
        sgcn: #generator;
        attachments
            resroomhandler.require(retrieve, typeroom, cant, total) => ccoor.intercept(retrieve, typeroom, cant, total);
            ccoor.act advance(counttop, resp) => counter.modify(modifop, resp);
            ccoor.act doslo(locate, resp) => findroom.modify(modifop, resp);
            restauranthandler.send(retrieve, typeroom, cant, total) => roomhandler.provide(retrieve, typeroom, cant, total);
            ccoor.connector(conexion) => sgcn.connector(conexion);
            ccoor.calltosc(ocoperation, match) => sgcn.replytoco(scoperation, match);
            //the same for waitinglist aspect associated to roomhandler (as client) and restauranthandler (as server) by coor3.
            //the same for counter aspect associated to customerapp (as client) and restauranthandler (as client) and restauranthandler (as client) by coor4
            //---------
    }
    component Client{
        interface
            require: Require;
    }
    component Server{
        interface
            provide: Provide;
    }
    component Aspect{
        interface
            modify: Modify;
    }
    component #generator{
        interface
            connector: Connector(conexion){
                spec is connexion{.newSc[Connector(conexion)};
            }
            replytoco { spec is Implicit;
        }
    }
    composition
        sc[]: Sc;
        attachments
            sc[*].replytoco>>replytoco;
    }
    //----------
    component Coordinator twice{
        interface
            connector: Connector(conexion){
                spec is connexion{.Connector(conexion)};
            }
            act: Actuno;
            actdos: Actdos;
            send: Send;
            intercept: Intercept;
    }
}
```

Fig. A4. Extended system code in LEDA for the case study.