Aspect-Oriented Connectors for Coordination[*]

Jennifer Pérez
Technical University of Madrid (UPM)
E. U. Informática, Carretera de Valencia, Km. 7
28031 Madrid, Spain
jperez@eui.upm.es

Carlos E. Cuesta
Dept. Computer Languages and Systems II
Rey Juan Carlos University
Móstoles 28933 Madrid, Spain
carlos.cuesta@urjc.es

ABSTRACT

Coordination has become a key concept in the industrial systems as it leads to a better understanding of the interactions that take place in complex and distributed systems. In the last few years, coordination has been introduced in two important fields of Software Engineering: Software Architectures, through the notion of connector, and Aspect-Oriented Software Development, through the notion of weaving and by considering coordination as an aspect. In this paper, we present how the PRISMA model orchestrates its aspect-oriented architectural models by using aspect-oriented connectors. Due to the complexity of a coordination process, they must be well founded and defined. This paper presents the formalization of this combination of connectors and aspects to obtain more consistent, reusable and maintainable coordination models.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*connectors, interaction*; D2.2 [**Software Engineering**]: Design Tools and Techniques—*aspect-orientation, model-driven engineering*

Keywords

Coordination, Connector, Aspect-Orientation, Symmetric Aspect, Software Architecture, PRISMA

1. INTRODUCTION

Currently, there is a great interest in coordination. Coordination orchestrates processes in order to achieve the correct functionality of software products. Good coordination management is essential and is a risk factor for the synchronization of difficult tasks that industrial systems must perform. As a result, several software development approaches have taken coordination into account. Two widely used are Component-Based Software Development (CBSD) [37] and Aspect-Oriented Software Development (AOSD) [16].

On the one hand, coordination is an important topic in CBSD and, by extension, in Software Architectures since it can be used to synchronize the components that form a specific architecture. In fact, Architecture Description Languages (ADLs) [22] could be classified according to the importance they give to coordination. Some of these ADLs have introduced the notion of connector, which is an architectural element that acts as a coordinator among other architectural elements (either connectors or components) [1, 7, 22]. However, other ADLs do not include connectors [4, 19]. Those that use the notion of connector give more relevance to coordination because they provide a specific architectural element to define it. In addition, they offer an architectural view of systems; whereas, an ADL without connectors has a more compositional view, as in object-oriented models [19, 18]. As a result, an ADL should provide connectors in order to separate coordination from computation and to provide an architectural view instead of a compositional view.

On the other hand, AOSD allows the separation of crosscutting concerns of software systems in a modular entity called *aspect*. Among the different crosscutting concerns that can be identified in software systems, coordination is perhaps one of the most common. But in addition to this characterization as a concern, coordination has also emerged as an important feature within AOSD itself, because the different aspects of a software system must also be synchronized. The need for aspect coordination has been identified as a key feature in this approach [15].

In this paper, the formalization of a model that combines Software Architectures and AOSD is presented. This model is called PRISMA [32]. PRISMA encapsulates properties and behaviour of different crosscutting-concerns into different aspects (coordination, safety, security, distribution, etc.). The definition of a connector is specified by importing different aspects, one of them must be a coordination aspect. As a result, coordination models of connectors are not tangled with the rest of concerns that affect the coordination process, have cleaner specifications, are more reusable and maintainable.

In order to illustrate the properties of PRISMA connectors and their coordination processes, a case study is going to be used throughout the paper. The case study is a connector

[*]This research has been partially supported by the Spanish Ministry of Education and Science through the Technical University of Valencia, in the context of the National Research Project META (MEC-TIN2006-15175-C05-01) and within the framework of the ISSI Research Group on Software Engineering and Information Systems.

that must coordinate two components, an actuator and a sensor. These three components belong to the software architecture of a robot, and their correct coordination allows the movement of a joint in this robot. The actuator sends the joint the order to do movements, and the sensor reads the results of those movements. In addition, the connector has to take into account safety constraints in order to be sure that the movements that it sends to the actuator are safe both for the robot and its environment.

The main contribution of this paper is the formalization of aspect-oriented connectors in order to define their coordination process in a formal way and thus, to avoid ambiguity. Since PRISMA connectors are observable processes that have state and behaviour, the formalisms which are used to formalize the PRISMA model are a variant of a Modal Logic of Actions [36], and a extension of the π-calculus [24] which provides priorities. The π-calculus is a process algebra which is used to specify and formalize the processes of the PRISMA model, and the Modal Logic of Actions is used to formalize the way in which the execution of these processes affects the state of architectural elements.

The paper is structured as follows: first an overview of the PRISMA concept model is provided, explaining the reasons behind its symmetric aspect model. Then the interest of using aspect-oriented connectors is discussed in detail, justifying the relevance of the PRISMA model and its merits with regard to other proposals. Once this interest has been stated, the concrete structure of *connectors* in PRISMA is described in detail, and the formalization of the relevant PRISMA concepts is provided and explained. Elements of the language are specified in terms of a dialect of the π-calculus, alongside with a real-world example of a connector within a robotic system. The notion of *weavings* is described with special care. After the structure of PRISMA connectors and its formalization have been described in detail, we briefly discuss how it compares to related work, and highlight the advantages of our model. We conclude by summarizing the results and the directions of further work.

2. AN OVERVIEW OF PRISMA

PRISMA provides a model for the description of software architectures of complex and large systems. It introduces aspects and connectors as first-class citizens of software architectures, and therefore is an aspect-oriented ADL.

This section present just a brief introduction to the PRISMA model; more detailed descriptions for it and the associated language can be found for example in [31, 32, 30].

From the aspect-oriented point of view, PRISMA is a symmetric model that defines functionality as an aspect. One concern can be specified by several aspects of a software architecture, whereas a PRISMA aspect represents a concern that crosscuts the software architecture. This crosscutting is due to the fact that the same aspect can be imported by more than one architectural element of a software architecture. In this sense, aspects crosscut those elements of the architecture that import their behaviour (see Figure 1).

A PRISMA architectural element can be seen from two different views: internal and external. In the external view, architectural elements encapsulate their functionality as black boxes and publish a set of services that they offer to other architectural elements (see Part A of Figure 2). These services are grouped into interfaces to be published through the ports of architectural elements. Each port has an associated interface that contains the services that are provided and requested through the port. As a result, ports are the interaction points of architectural elements.

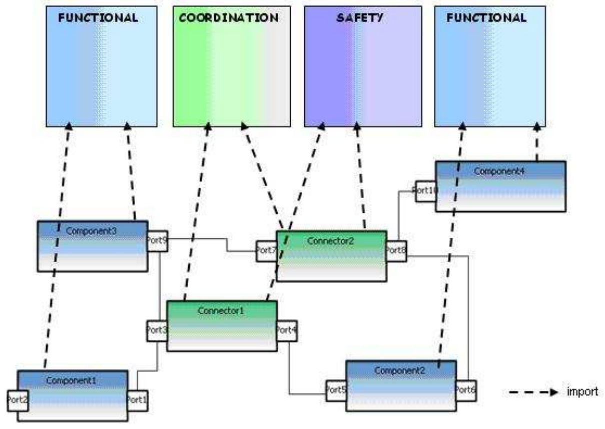

Figure 1: Crosscutting Concerns in PRISMA

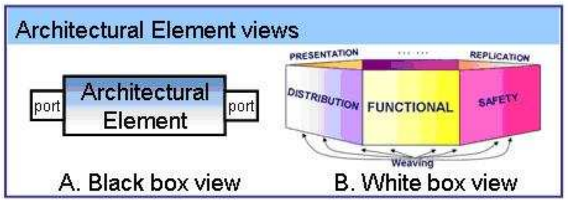

Figure 2: Views of a PRISMA Architectural Element

The internal view shows an architectural element as a prism (white box view). In this metaphor, each side of the prism is an aspect that the architectural element imports. In this way, architectural elements are represented as a set of aspects (see Part B in Figure 2) and the weaving relationships among aspects.

Since PRISMA uses a symmetric aspect-oriented model [13] that it is applied at the architectural level, the weaving process indicates that the execution of an aspect service can trigger the execution of services in other aspects. In PRISMA, in order to preserve the independence of the aspect specification from other aspects and weavings, those weavings are specified outside aspects and inside architectural elements. As a result, aspects are independent of the context of application and weavings coordinates the different aspects that form an architectural element. This way of describing the relationship between aspects provides greater flexibility, as different behaviours can be specified for the same architectural element by importing the same aspects and defining different weavings (coordination processes of aspects).

PRISMA has two kinds of architectural elements: components and connectors. A component is an architectural element that captures the functionality of software systems and does not act as a coordinator among other architectural elements; whereas, a connector is an architectural element that acts as a coordinator among other architectural elements.

Connectors do not have the references of the components that they connect and vice versa. Thus, architectural elements are orthogonal and unaware of each other. This is possible due to the fact that the channels defined between components and connectors have their references (attachments) instead of architectural elements. Attachments are the channels that enable the communication between components and connectors. Each attachment is defined by attaching a component port with a connector port.

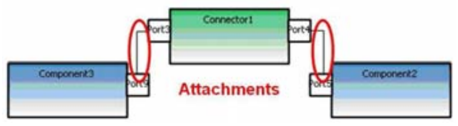

Figure 3: Attachments in Prisma

Since Prisma architectural elements are aspect-oriented, connectors import a set of aspects to perform the coordination process. As such, connectors have a coordination aspect. A coordination aspect defines how several architectural elements are synchronized while they communicate with each other. The coordination aspect is based on the notion of contract proposed by Andrade and Fiadeiro [3]. This allows us to define several coordination models as different coordination aspects. Since Prisma aspects are reusable types, a coordination model that has been specified as an aspect can be reused to create different connectors for the same architectural model or a different one.

2.1 Aspect-Oriented Connectors

It is important to keep in mind that current software systems perform complex coordination processes that have to take into account not only the coordination concern, but also other concerns such as: safety, distribution, security, etc. These other concerns are necessary in order to provide a correct coordination process. For example: The connectors that coordinate the actuators and sensors of teleoperated robots need to check that the movement is safe for the robot before sending the movement to the actuator. Prisma aspect-oriented connectors are presented as a solution for the specification of these complex coordination processes by improving the reusability and maintenance of software. This is improvement has been achieved by overcoming the disadvantages of the rest of ADLs. Current ADLs can be classified into three different kinds: non-aspect-oriented ADLs without connectors, non-aspect-oriented ADLs with connectors, and aspect-oriented ADLs. Next, we present how they specify complex coordination processes that have to take into account several concerns:

- **Non-aspect-oriented, connector-less ADLs**. There are ADLs that prefer the absence of connectors because they distort the compositional nature of software architectures. Some ADLs, such as Darwin [19], Leda [4], and Rapide [18] do not consider connectors as first-class citizens. However, these ADLs make difficult the reusability of components because they have the coordination process tangled with the computation inside them, and they are aware of the coordination process that has to happen in order to communicate with the rest. The notion of connector emerges from the need to separate the interaction from the computation in order to obtain more reusable and modularized components and to improve the level of abstraction of software architecture descriptions. Mary Shaw [34] presents the need for connectors due to the fact that the specification of software systems with complex coordination protocols is very difficult without the notion of connector. From her experience in the software architecture field, she demonstrates that the connector provides not only a high level of abstraction and modularity to software architectures, but also an architectural view of the system instead of the object-oriented view of compositional approaches. She also defends the idea of considering connectors as first-order citizens of ADLs. Figure 4 illustrates how two components (actuator and sensor) are communicated using an ADL without connectors. The coordination process is encapsulated in the components and tangled with the computation and other concerns.

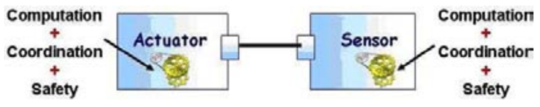

Figure 4: Sensor-Actuator Coordination by Using a Connector-less Adl

- **Non-aspect-oriented ADLs with connectors**. Most ADLs provide connectors as a first order citizens of the language such as: ACME [11], Aesop [10], C2 [20, 21], SADL [25], UniCon [34, 35], Wright [1], CommUnity [9], Pilar [5], ArchWare's π-ADL [27, 28], etc. All of these languages go a step forward with regard to the previous kind of ADLs. They improve the reusability of components and connectors by separating computation from coordination. However, their connectors are non-aspect-oriented and they specify their coordination processes by tangling the code inside them. For example: the coordination process between an actuator and a sensor of a robot will imply the specification of a connector with tangled concerns of coordination and safety (see Figure 5).

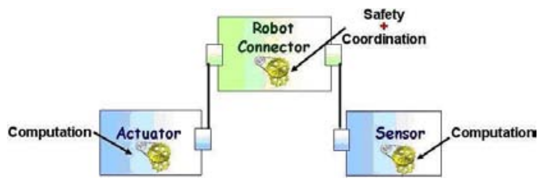

Figure 5: Sensor-Actuator Coordination by Using an Adl with Connectors

- **Aspect-oriented, connector-less ADLs**. Most aspect-oriented approaches applied to software architectures and their ADLs are based on an original ADL without connectors such as: PCS [14], DAOP-ADL [33], AspectLEDA [26], AOCE [12], etc. These ADLs introduce the aspect-oriented behaviour by means of connectors, i.e., aspects are connectors among components. However, when there are two components that are coordinated by several connectors (aspects), the connectors cannot be synchronized among them (weavings among aspects). And in those ADLs that could try to solve this problem by connecting both connectors they will lose the reusability of the concerns of those connectors, because they will be dependent to the connector (aspect) that are connected to. Figure 6 illustrates how two components are communicated using an aspect-oriented ADL without connectors.

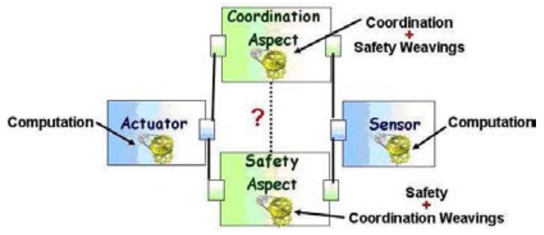

Figure 6: Sensor-Actuator Coordination by Using a Connector-less Aspect-Oriented ADL (AoAdl)

- However, in Prisma a new kind of ADLs is introduced, namely **aspect-oriented ADLs with connectors**. Prisma is based on an ADL with connectors, and aspects are introduced as a new concept in software architectures for concerns called aspects. As a result, each concern is specified in its aspect and the coordination rules among the different aspects are inside the connector being aspects reusable and independent one to each other. Figure 7 presents how Prisma coordinates the sensor and the actuator by separating the concerns or computation, safety and coordination. As a result, they are not scattered through the architecture and they are not repeated.

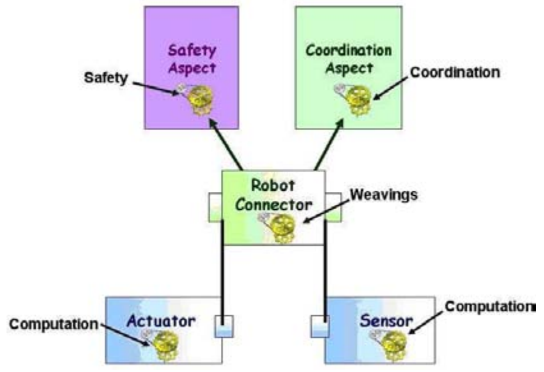

Figure 7: Sensor-Actuator Coordination by Using the Prisma Adl

In addition, Figure 7 shows that the coordination process among components, connectors and aspect is very complex. For this reason, this coordination process must be very well defined and formalized in order to guarantee that it coordinates all the pieces of software successfully. The formalization of this coordination process is the main contribution of this paper from previous presentations of Prisma. It is detailedly presented along the next section.

3. CONNECTORS IN PRISMA

A connector is an architectural element that acts as a coordinator between other architectural elements. As such, connectors have a coordination aspect. An example is the connector that synchronizes the Actuator and the Sensor of a robot joint. This connector imports a safety aspect and a coordination aspect to coordinate the movements of the robot in a safe way for the joint, the robot and the environment that surrounds it.

3.1 Architectural Element

Since a connector is an architectural element, a connector is formalized as an architectural element. An architectural element is formed by a set of aspects, their weaving relationships, and one or more ports. These ports represent interaction points among architectural elements.

Formalization: Architectural Element

An architectural element $\mathcal{AE}$ is built by composing a set of aspects $A_1, A_2, \ldots, A_n$, which are conceived as the smallest modules in our approach, and will be defined in section 3.3. The resulting element $\mathcal{AE}$ is in turn defined itself by the 4-tuple (A, X, Φ, Π), as follows:

A. The set of attributes in aspects $A_1, A_2, \ldots, A_n$.

X. The set of the services in aspects $A_1, A_2, \ldots, A_n$. See Definition 1, in section 3.4 below.

Φ. The set of formulae (in a modal Logic of Actions) providing constraints for aspects $A_1, A_2, \ldots, A_n$.

Π. The process P_{AE}, defined as follows:

$$P_{AE} ::= P_{P1} || \ldots || P_{Pm} \,||\, P_{A1} \,||\, \ldots \,||\, P_{An} \,||\, P_W$$

This means that the processes of the ports, weavings and aspects of the architectural element are executed concurrently. For this reason, P_{AR} is defined as their parallel composition, and therefore their dependencies are expressed and solved just as concurrency conflicts.

A brief comment about the role of the Modal Logic of Actions in Prisma is relevant here. Basically, the formulae in Φ are used for implementing obligations, prohibitions, and permissions, providing the concurrent equivalent of a deontic logic. As a result, it permits the analysis and formulation of assertions about processes that change the execution environment. A formula of this Modal Logic of Actions is written following the structure $\psi[a]\varphi$, where ψ and φ are well-formed formulae (wff) in conventional first-order logic, which characterize the state before or after the execution of the action a, respectively. As usual in modal logics, the construct $[]$ represents the necessity operator, and a represents an action. As a result, the meaning of formulae which are constructed following this pattern ($\psi[a]\varphi$) is the following: "if ψ is satisfied before the execution of a, φ *must* be satisfied after the execution of a".

To conclude, we provide an example for an architectural element (and particularly of a connector), namely the *RobotConnector* in charge of synchronizing the *Actuator* and the *Sensor* of a robot (see Figure 7). This connector imports the *SMotion* safety aspect and the *CoordJoint* coordination aspect as mentioned above and is formed by the follow set of ports and weavings (see Figure 8).

The formalization of this connector is therefore given by the following composite π-process:

$$\begin{aligned} P_{RobotConnector} \quad ::= \quad & P_{PAct} \,||\, P_{PSen} \,||\, P_{ASMotion} || \\ & P_{ACoordJoint} \,||\, P_W \end{aligned}$$

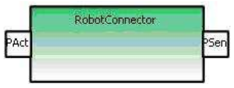

Fig.8(a). Black Box Representation

```
Connector RobotConnector

   Coordination Aspect Import CoordJoint;
   Safety Aspect Import SMotion;

   Weavings
      SMotion.DangerousCheck(NewSteps,
                                    Speed, Safe)
      beforeif (Safe = true)
         CoordJoint.movejoint(NewSteps, Speed);
   End_Weavings;

   Ports
      PAct : IMotionJoint,
         Played_Role CProcessSuc.ACT;
      PSen : IRead,
         Played_Role CProcessSuc.SEN;
   End_Ports
   . . . . . .
End_Connector CnctJoint;
```

Fig.8(b). Prisma **Specification**

Figure 8: The *RobotConnector* Connector

3.2 Ports

Ports are the interaction points of architectural elements (components and connectors). Every port has associated a process, which establishes the services that publishes, and how and when they can be executed.

Formalization: Ports

Let P be a port of an architectural element, such that its behaviour is specified by a process PR_1. Then its semantics are given by the process P_P, defined simply as follows:

$$P_P ::= PR_1$$

An example is the port *PAct* in the *RobotConnector* example (see Figure 8)), which has its behaviour specified as a process P_{PAct}, which in turn refers to the generic definition of another process ACT.

$$P_{\mathsf{PAct}} ::= ACT$$

3.3 Aspect

An aspect defines the structure and the behaviour of a specific concern of the software system. Examples of concerns are functionality, coordination, safety, distribution, among others.

Structure is defined by a set of attributes, each of which has a value in every state. The state of the aspect at any given moment is determined by the value of its attributes. An aspect defines a semantics for its services. This semantics captures when the services cannot be executed, how the execution of services changes the state of the aspect, and the order in which they can be executed. The behaviour of an aspect is defined by means of a protocol. The protocol describes how the different services of the aspect are coordinated.

Formalization: Aspects

An *aspect* is defined as follows by the 4-tuple (A, X, Φ, Π):

A. A set of attributes.

X. A set of services (see section 3.4).

Φ. A set of formulae in a modal logic of actions.

Π. A set of π-terms; this is, a set of concurrent terms describing partial processes in the π-calculus.

The contents of set Π are therefore a set of π-calculus processes. For instance, let α be an aspect whose behaviour is specified by by the $PRT1$ protocol. Then its semantics is the process P_α defined as follows:

$$P_\alpha ::= PRT1$$

Again, in the *RobotConnector* example of Figure 8, the *SMotion* aspect is similarly defined as:

$$P_{\mathsf{SMotion}} ::= \mathit{SMotionProtocol}$$

The dialect we use to describe terms in the Π set is a syntactic variant of the polyadic π-calculus. It also includes an extension to include priorities, which we will not describe nor use here. But apart from this extension, the language is largely standard, even in the choice of derived operators (such as **if** ... **then**). The main syntactic differences are the use of the arrow ($\rightarrow$) as the prefix operator to define a *sequence of actions*, instead of the dot (.), which is used here with its usual meaning at the programming level, to indicate scope nesting.

Finally, the dialect provides also support for vector-like tuples of channels, which are simply indicated as $\overline{v}$. We will assume an implicit indexing operator in this kind of vectors, so the name v_1 will refer to the first channel in the vector $\overline{v}$. This should be considered just as syntactic sugar.

3.4 Weavings

A weaving specification defines how the execution of a service of an aspect can trigger the execution of a service of another aspect. Of course, the same service can be involved in several weavings.

In order to preserve the independence of the aspect specification from other aspects and weavings, weavings in Prisma are specified outside aspects and inside architectural elements, including connectors. As a result, weavings specified inside connectors are the ones which coordinate the different aspects that a connector imports.

A weaving is defined by means of operators that describe the order in which services are executed.

A weaving that relates service s1 of aspect A1 and service s2 of aspect A2 can be specified using the following operators. Note the use of the dot (.) operator to indicate scope nesting, as indicated above.

- A2.s2 **after** A1.s1. A2.s2 is executed after A1.s1.
- A2.s2 **before** A1.s1. A2.s2 is executed before A1.s1
- A2.s2 **instead** A1.s1. A2.s2 is executed instead of A1.s1

- A2.s2 **afterif** (*Boolean condition*) A1.s1. A2.s2 is executed after A1.s1 if the condition is satisfied.
- A2.s2 **beforeif** (*Boolean condition*) A1.s1. If the condition is satisfied, A2.s2 is executed followed by A1.s1; otherwise, *only* A2.s2 is executed.
- A2.s2 **insteadif** (*Boolean condition*) A1.s1. A2.s2 is executed instead of A1.s1 if the condition is satisfied.

The invocation of A1.s1, the second argument of the weaving, triggers the execution of weaving. When a weaving is specified, the operator must be chosen from the point of view of the triggered service, depending on whether the triggered service needs the execution of a service before, after, or instead of it. Therefore the *before* and *after* weaving modifiers are not directly interchangeable.

Formalization: Weavings

The semantics of a weaving is a coordination process that intercepts the invocation of a service $A1.s1$ and either replaces it with, or executes it in relation to, another service $A2.s2$. $A1.s1$ and $A2.s2$ belong to different aspects.

The weaving must be executed each time that A1.s1 is invoked, upon which it executes either A2.s2 instead of A1.s1 or A1.s1 and A2.s2 in the correct order. This means that the invocation of a service does not automatically trigger the execution of its associated process. Taking into account that the formalization of a service in PRISMA is the following:

DEFINITION 1. *(Service) A* service *is a process that executes a set of actions to produce a result.*

Let S be a service. The semantics of S is a process in the polyadic π-calculus called P_S. This process has a channel C_S through which it is able to interact; or, conversely, it can be invoked for execution (see Figure 9). We shall see immediately that services are not invoked directly by other processes, but only through weavings that coordinate execution of services within architectural elements.

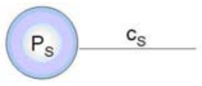

Figure 9: Formalization of a Service

Let's start by defining a service invocation. This will make much easier to understand later the way in which we will define the internal behaviour of a service.

DEFINITION 2. *(Service Invocation) Let $\overline{x} = x_1 \ldots x_n$ be the input parameters for a service S, and $\overline{y} = y_1 \ldots y_m$ be its output parameters. The invocation of S is formalized by means of a message sent through channel C_S.*

Moreover, each output parameter y_i must have a return channel r_{yi}, which is dynamically created for each invocation using the π-calculus restriction operator (ν). These channels are used to send the results of S and to indicate and acknowledge termination of the execution of S.

All this considered, a service invocation is described as the following process.

$$(\nu\,\overline{r_y})(C_S!(\overline{x},\overline{r_y}) \rightarrow r_{y1}?(y_1)\ldots r_{ym}?(y_m))$$

The structure of this process defines the different ways in which a service is able to interact; so, we are now able to define the behaviour of a service as a set of π-calculus processes, as indicated by the following definition.

DEFINITION 3. *(Service Process) The behaviour of a process P_S of a service S can be divided in three kinds of actions:*

- *Request Reception. The first action of P_S must be the reception of the messages that come through C_S. This reception is specified as follows.*

$$C_S?(\overline{x},\overline{r_y})$$

- *Service Execution. The execution of the service internal behaviour consists of processing a set of internal actions. We create the output parameters ($\overline{y} = y_1 \ldots y_m$) and assume that internal actions bind them with some useful value. Then this internal execution is specified as follows.*

$$(\nu\,\overline{y})(\tau)$$

- *Termination. The last action in P_S is always the sending of the output parameters ($\overline{y} = y_1 \ldots y_m$) through return channels ($\overline{r_y} = r_{y1} \ldots r_{ym}$). This way, the invoker is confirmed that execution of S has ended. This termination is therefore specified as follows.*

$$r_{y1}!(y_1)\ldots r_{ym}!(y_m)$$

As a result, the complete formalization of P_S is the replicated sequence of these three actions.

$$P_S ::= *(C_S!(\overline{x},\overline{r_y}) \rightarrow (\nu\overline{y})((\tau) \rightarrow r_{y1}!(y_1)\ldots r_{ym}!(y_m)))$$

This replication allows us to execute the service as many times as it is necessary.

In terms of our formalization in the π-calculus, and given a service S which is being controlled by the weaving, this means that the weaving process P_W interacts with P_S via the channel C_S defined in Definition 1. To do so, it must provide a channel C_{WS} which other processes can use to invoke S (see Figure 10).

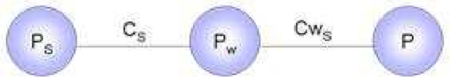

Figure 10: Formalization of a Service Controlled by a Weaving

Considering these two channels, the invocation of S by other processes is defined as the following π-term:

$$(\nu\,\overline{r_y})C_{WS}!(\overline{x},\overline{r_y}) \rightarrow r_{y1}?(y_1)\ldots r_{ym}?(y_m))$$

And then the invocation of S by the weaving process is therefore as follows:

$$(\nu\,\overline{r_y})(C_S!(\overline{x},\overline{r_y}) \rightarrow r_{y1}?(y_1)\ldots r_{ym}?(y_m))$$

After that, each weaving operator defines a different process with an specific behaviour, to provide the required semantics for each one of them. As an example, let's consider the process for the *beforeif* weaving operator, which involves two services belonging to two different aspects.

$$
\begin{array}{lll}
P_{1\ldots n} & ::= & (\nu\,\overline{r_y})\,(C_{WA1\_S1}!(\overline{x},\overline{r_y}) \rightarrow r_{y1}?(y_1)\ldots r_{ym}?(y_m)) \\[1em]
P_{BWIF} & ::= & *\,(C_{WA1\_S1}?(\overline{x},\overline{r_y}) \rightarrow (\nu\,\overline{r_{s2}})(C_{A2\_S2}!(\overline{x},\overline{r_{s2}}) \rightarrow r_{s21}?(s_{21})\ldots r_{s2m}?(s_{2m})) \rightarrow \\
 & & \quad \textbf{if}\ (\text{Boolean\_condition}) = \textbf{true})\ \textbf{then} \\
 & & \qquad (\nu\,\overline{r_{s1}})(C_{A1\_S1}!(\overline{x},\overline{r_{s1}}) \rightarrow r_{s11}?(s_{11})\ldots r_{s1m}?(s_{1m})) \rightarrow r_{y1}!(s_{11})\ldots r_{ym}!(s_{1m})) \\
 & & \quad \textbf{else} \\
 & & \qquad r_{y1}!(s_{21})\ \ldots\ r_{ym}!(s_{2m})) \\[1em]
P_{A1\_S1} & ::= & *\,(C_{A1\_S1}?(\overline{x},\overline{r_{s1}}) \rightarrow (\nu\,\overline{s_1})((\tau) \rightarrow r_{s11}!(s_{11})\ldots r_{s1m}!(s_{1m}))) \\[1em]
P_{A2\_S2} & ::= & *\,(C_{A2\_S2}?(\overline{x},\overline{r_{s2}}) \rightarrow (\nu\,\overline{s_2})((\tau) \rightarrow r_{s21}!(s_{21})\ldots r_{s2m}!(s_{2m})))
\end{array}
$$

Table 1: Translation set of π-processes for *beforeif* Weaving Pattern

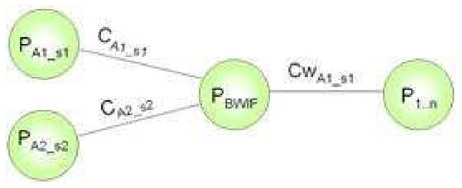

Figure 11: Translation for *beforeif* Weaving Pattern

$$A2.s2\ \textbf{beforeif}\ (\text{Boolean\_condition})\ A1.s1$$

According to PRISMA formal semantics, this weaving pattern will be translated to the π-calculus as a compound process P_{BWIF}, which has the context depicted in Figure 11.

This means that P_{BWIF} receives the invocation of A1.s1 from another process ($P_{1\ldots n}$) through C_{WA1_S1}. As $BWIF$ is a "before" weaving, P_{BWIF} starts by invoking A2.s2 using C_{A2_s2}. Then, P_{A2_s2} receives the invocation, executes a set of internal actions, sends the results, and notifies the weaving that execution has finished. Next, if the boolean condition in $BWIF$ is true, the first service of the weaving is executed; otherwise P_{BWIF} sends the results of A2.s2 to the process that invoked A1.s1. In the first case, when the condition is satisfied, P_{BWIF} invokes A1.s1 using C_{A1_s1} and P_{A1_s1} receives the invocation upon which it executes a set of internal actions, sends the results, and notifies the weaving that the execution has finished. Finally, P_{BWIF} sends the results of A1.s1 to the process that invoked A1.s1.

The semantics of the set of weavings defined inside a connector is therefore translated as the P_W process, the parallel composition of every individual weaving process.

$$
\begin{array}{lll}
P_W & ::= & P_{AW1}\,||\,\ldots\,||\,P_{AWn}\,||\,P_{BW1}\,||\,\ldots\,||\,P_{BWn}\,|| \\
 & & P_{IW1}\,||\,\ldots\,||\,P_{IWn}\,||\,P_{AWIF1}\,||\,\ldots\,|| \\
 & & P_{AWIFn}\,||\,P_{BWIF1}\,||\,\ldots\,||\,P_{BWIFn}\,|| \\
 & & P_{IWIF1}\,||\,\ldots\,||\,P_{IWIFn}
\end{array}
$$

This means that the weavings are executed concurrently, interacting as specified. In addition, the same service can be involved in several weavings of the same architectural element and there is an order for processing the different weavings that a service triggers. This ordering establishes that weavings are executed from more restrictive to less restrictive. The precedence is as follows: InsteadIf, Instead, BeforeIf, Before, After, AfterIf. Deadlocks and infinite loops that could appear when using these operators are avoided at the specification time.

An example of a weaving appears in the *RobotConnector* case study. This connector imports the *SMotion* safety aspect and the *CoordJoint* coordination aspect. The need for a weaving emerges due to the fact that the robot is moved only after the connector is sure that a movement is safe. The invocation of the *moveJoint* service (the second argument of the weaving) of the *CoordJoint* triggers the execution of the weaving (see the process $P_{BWIF\mathsf{SMotionCoordJoint}}$ in Figure 12). Specifically, the weaving of the connector receives the invocation of the *moveJoint* service (the term $C_{W\mathsf{CoordJoint\_moveJoint}}?(newsteps, speed, \overline{r_y})$ in the process) and afterwards it specifies that the DANGEROUSCHECK service of *SMotion* has to be executed, and it must answer before the *moveJoint* service of *CoordJoint* is even invoked, hence the term $(\nu\,\overline{r_{s2}})(C_{\mathsf{SMotion\_DANGER}}!(newsteps, r_{s21}) \rightarrow r_{s21}?(safe))$ in the process. Then the condition guarantees that the execution of the *moveJoint* service is only performed if the *safe* return parameter of the DANGEROUSCHECK service is set to *true* (hence the *if/then/else* construct in Figure 12, which encloses the invocation of the *moveJoint* service through the $C_{\mathsf{CoordJoint\_moveJoint}}$ channel). On the other hand, the processes defining the behaviour of each one of the services, which are in turn defined within the aspects, are ready to be invoked by the weaving at any time (see the definition for both $P_{\mathsf{CoordJoint\_moveJoint}}$ and $P_{\mathsf{SMotion\_DANGER}}$ as replicated, hence permanent, processes in the Figure).

4. DISCUSSION AND RELATED WORK

Both coordination and architecture are generic high-level abstractions of a software system; they provide different approaches to close concerns, and both have long and separate research traditions. At the same time, there is an obvious relationship between them. Both notions try to identify high-level patterns in the system, though their perspectives are slightly different. Architecture identifies structural patterns defined by inner interaction within a (mostly) compositional configuration, while coordination defines high-level interaction patterns shown by the resulting structure.

However when the relationship between them is considered, even their relative ordering has not always been clear. Different authors have considered their relationship in different ways, and this is the best proof of their intertwining and the intrinsic difficulty of their separation. For instance, Andrade *et al.* [2] consider that configurations (architectural

$$
\begin{array}{rcl}
P_W & ::= & P_{BWIF\mathsf{SMotionCoordJoint}} \\
P_{BWIF\mathsf{SMotionCoordJoint}} & ::= & *(C_{W\mathsf{CoordJoint\_moveJoint}}?(newsteps, speed, \overline{r_y}) \rightarrow \\
& & (\nu\,\overline{r_{s2}})(C_{\mathsf{SMotion\_DANGER}}!(newsteps, r_{s21}) \rightarrow r_{s21}?(safe) \rightarrow \\
& & \mathbf{if}\,(safe = \mathbf{true})\,\mathbf{then} \\
& & \quad (\nu\,\overline{r_{s1}})\,(C_{\mathsf{CoordJoint\_moveJoint}}!(newsteps, speed, \overline{r_{s1}}) \rightarrow \\
& & \qquad\qquad r_{s11}?(s11) \rightarrow r_{y1}!(s11)) \\
& & \mathbf{else} \\
& & \quad r_{y1}!(s21))) \\
P_{\mathsf{CoordJoint\_moveJoint}} & ::= & *(C_{\mathsf{CoordJoint\_moveJoint}}?(newsteps, speed, \overline{r_{s1}}) \rightarrow (\nu\,\overline{s_1})((\tau) \rightarrow r_{s11}!(s11)) \\
P_{\mathsf{SMotion\_DANGER}} & ::= & *(C_{\mathsf{SMotion\_DANGER}}?(newsteps, r_{s21}) \rightarrow ((\tau) \rightarrow r_{s21}!(safe)))
\end{array}
$$

Figure 12: Translation for the Weaving in the *RobotConnector* example

structures) are built on top of a coordination layer which guarantees a shared behaviour. On the contrary, Eisenbach and Radestock [8] conceive coordination as the higher level abstraction, which is built on top of a configuration layer, which guarantees a substrate for shared interaction. At the same time, many authors present these two abstractions at the same level and provide a common support for it; in particular, many coordination languages have also been presented as ADLs, provided that their particular abstractions are equally good for describing both [19, 29]. In particular, connectors and special-purpose components bear many similarities to some constructs in several control-driven coordination proposals.

Probably among the most important reasons for the success of the architectural approach is the implicit separation of concerns it provides; the designer is just concerned with the functionality of components (and possibly some relevant non-functional requirements), but he is now relieved of describing compositional and coordination issues, which have become the architect's responsibility. Though connectors are not the only way in which an ADL can describe interaction and coordination abstractions, their existence and the emphasis on them is probably the reason why these languages are so apt in specifying these issues. And once they have been separated, we find that relevant high-level linguistic constructs in different approaches are similar.

We could conclude that coordination is an emergent property of some architectures; an architecture-level description has the means for providing the coordination concern, but of course it can also describe non-coordinated systems. In summary, architectures describe interaction structures; and coordination can be described as a higher-level abstraction on interaction, therefore supported by architecture [6].

Connectors alone do not provide a global coordination policy, but only local coordination groups; therefore the use of connectors (as discussed in section 2.1 and above) eases the description of a coordinated system, but it is not a sufficient condition. Shaw's original identification of connectors [34] tried not to provide a coordination, but an interaction abstraction. However, subsequent work has defined ever more complex connectors, which were grouped in types and categories, tending towards the definition of much more complex abstractions, even higher-order connectors [17]. Mehta provided an initial taxonomy for connectors [23], which could have provided a basis for later developments in this direction, but this thread has not had a continuity.

The locality of the connector approach justifies still the definition of generic coordination language proposals, which provide the means to describe general policies. However an aspect-oriented alternative is also possible. Instead of providing a complete language from scratch, it is also possible to define an aspect-oriented extension of some existing language. More than that, this would ease the integration of this "coordination aspect" with other concerns in the architecture. Consider also that early proposals for aspect-orientation [16] defined specific-purpose languages to deal with aspects, rather than aspectual extensions, so this evolution towards an architectural extension is also within the tradition in the field.

Therefore we can provide coordination by means of pure compositional ADLs, but connectors make it easier. Then, an specific coordination language provides general policies, but an aspect-oriented extension makes integration easier. Thus, providing an aspect-oriented, connector-based ADL would gather the benefits of different proposals. We refer the reader again to section 2.1 for a detailed discussion of the different approaches for providing coordination in ADLs, including connector-based and aspect-oriented alternatives.

Our own previous work on the reflective ADL $\mathcal{P}i\mathcal{L}ar$ has explored the way in which a very general architecture language is able to describe coordination as a separate concern, i.e. as an *architectural aspect*. In [5] this was made by exploiting the reflective capabilities, thus proving that this is indeed possible, but very complex. Later research has explored also an aspect-oriented approach which makes a non-explicit use of these reflective capabilities in $\mathcal{P}i\mathcal{L}ar$, providing an aspectual layer and showing how coordination can be independently managed as an aspect [6]; but the relationship between this aspect and others, though possible, was complex, and is not explored in detail.

And this is the main benefit of the PRISMA approach, as highlighted in previous discussion. The aspect-oriented structure of the language itself, and the symmetry of its model, provide the basis to be able to relate coordination to other concerns, such as safety. The notion of *weaving*, which is required by the aspect model, provides also the means to reconcile the conflicts between aspects, whenever they appear. This, combined to the benefits of both connectors and aspects themselves for coordination, defines PRISMA as one of the most complete proposals in the field, gathering all the benefits provided by other approaches in a single, consistent and rigorous conceptual model.

5. CONCLUSIONS AND FUTURE WORK

In this paper, the advantages of combining software architectures and AOSD to define coordination have been presented. In addition, a detailed analysis about how to take more advantage of this combination has been done. From this analysis, this paper defines and formalizes PRISMA aspect-oriented connectors. They are specified in an elegant and novel way through the combination of AOSD and Software Architectures. As a result, PRISMA presents a coordination process that provides the following advantages:

1. Connectors to coordinate components: Reusability and maintenance of components and connector is improved by separating coordination from computation.

2. Aspects to specify the coordination process of connectors: Reusability and maintenance of different concerns is improved by separating coordination from other concerns that are necessary for the coordination process (safety, security, distribution, mobility, etc.). There are no tangled concerns inside complex connectors.

3. Weavings are inside connectors to coordinate their aspects: Reusability and maintenance of different aspect is improved by not specifying weavings inside aspects.

4. Formalization of the coordination processes among aspects (weavings) and architectural elements. Thus, non-ambiguity and proper execution of the different coordination processes is guaranteed.

Future work will exploit the results of this formalization to show the effects of combining several complex aspects, and will consider also the combination with the influence of assertions in the Modal Logic of Actions provided in architectural elements, as well as the possibility of extending this to a temporal logic such as the modal μ-calculus, which has already been made for recent work in $\mathcal{P}i\mathcal{L}ar$. Also, a detailed comparison with the formalization and capabilities of some other π-calculus-based ADLs, such as Leda [4], $\mathcal{P}i\mathcal{L}ar$ [5] or π-ADL [27] will be carried out, studing the extent in which our results can be provided as extensions to non-symmetric, non-aspect-oriented existing ADLs.

6. REFERENCES

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213 – 249, July 1997.

[2] L. F. Andrade, J. L. Fiadeiro, J. Gouveia, and G. Koutsoukos. Separating computation, coordination and configuration. *Journal of Software Maintenance*, 14(5):353–369, 2002.

[3] L. F. Andrade, J. L. Fiadeiro, J. Gouveia, G. Koutsoukos, and M. Wermelinger. 5th International Conference on Coordination Models and Languages. volume 2315 of *Lecture Notes in Computer Science*, pages 5 – 13, York, UK, April 2002. Springer Verlag.

[4] C. Canal, E. Pimentel, and J. M. Troya. Specification and Refinement of Dynamic Software Architectures. In *Software Architecture*, pages 107 – 126, San Antonio, Texas, February 1999. Kluwer Academic Publishing.

[5] C. E. Cuesta, M. P. Romay, P. de la Fuente, and M. Barrio-Solórzano. Reflection-based, Aspect-oriented Software Architecture. In *Software Architecture (EWSA 2004)*, volume 3047 of *Lecture Notes in Computer Science*, pages 43–56. Springer, Mayo 2004.

[6] C. E. Cuesta, M. P. Romay, P. de la Fuente, and M. Barrio-Solórzano. Coordination as an Architectural Aspect. *Electronic Notes in Theoretical Computer Science*, 154(1):25–41, Mayo 2006.

[7] C. E. Cuesta, M. P. Romay, P. Fuente, and M. Barrio-Solorzano. Architectural Aspects of Architectural Aspects. In R. Morrison and F. Oquendo, editors, *Software Architecture: Principles, Languages, Tools and Applications*, volume 3527 of *Lecture Notes in Computer Science*, pages 247 – 262. Springer, 2005.

[8] S. Eisenbach and M. Radestock. Component Coordination in Middleware Systems. In *IFIP International Conference on Distributed Systems Platforms and OpenDistributed Processing (Middleware'98)*, sep 1998.

[9] J. L. Fiadeiro and A. Lopes. CommUnity on the Move: Architectures for Distribution and Mobility. In F. S. de Boer, editor, *Fmco 2003*, volume 3188 of *Lecture Notes in Computer Science*, pages 177 – 196. Springer-Verlag, 2004.

[10] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *SIGSOFTŠ94: Foundations of Software Engineering*, pages 175 – 188, New Orleans, dec 1994.

[11] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, volume 68, pages 47 – 68. Cambridge University Press, 2000.

[12] J. C. Grundy, W. B. Mugridge, and J. G. Hosking. Static and dynamic visualisation of component-based software architectures. In *10th International Conference on Software Engineering and Knowledge Engineering*, pages 18 – 20, San Francisco, jun 1998. KSI Press.

[13] W. H. Harrison, H. L. Ossher, and P. L. Tarr. Asymmetrically Vs. Symmetrically Organized Paradigms for Software Composition. Technical Report RC22685 (W0212-147), Thomas J. Watson Research Center, IBM, 2002.

[14] M. M. Kande. *A concern-oriented approach to software architecture*. PhD thesis, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, 2003.

[15] G. Kiczales, E. Hilsdale, J. Huguin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer-Verlag, 2001.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, and C. V. Lopes. Aspect-Oriented Programming. In *11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220 – 242. Springer, 1997.

[17] A. Lopes, M. Wermelinger, and J. L. Fiadeiro. Higher-order architectural connectors. *ACM*

Transactions on Software Engineering and Methodology, 12(1):64–104, 2003.

[18] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717 – 734, sep 1995.

[19] J. N. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Fifth European Software Engineering Conference (ESEC)*, Barcelona, sep 1995.

[20] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *ACM SIGSOFTŠ96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pages 24 – 32, San Francisco, oct 1996.

[21] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *21st International Conference on Software Engineering (ICSEŠ99)*, Los Angeles, may 1999.

[22] N. Medvidovic and R. N. Taylor. A classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions of Software Engineering*, 26(1), 2000.

[23] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, volume 11, pages 178 – 187, Limerick, jun 2000.

[24] R. Milner. The Polyadic π-Calculus: A Tutorial. Technical report, Laboratory for Foundations of Computer Science, University of Edinburgh, oct 1993.

[25] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356 – 372, apr 1995.

[26] A. Navasa, M. A. Pérez, and J. M. Murillo. Aspect Modelling at Architecture Design. In *Software Architecture*, volume 3527 of *Lecture Notes on Computer Science*, pages 41 – 58. Springer Verlag, jun 2005.

[27] F. Oquendo. π-ADL: An Architecture Description Language based on the Higher-Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures. *ACM Software Engineering Notes*, 29(3), may 2004.

[28] F. Oquendo, B. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, and C. Occhipinti. ArchWARE: Architecting Evolvable Software. In *Software Achitecture (EWSA 2004)*, volume 3047 of *Lecture Notes in Computer Science*, pages 257 – 271, St Andrews, 2004. Springer.

[29] G. A. Papadopoulos and F. Arbab. Configuration and dynamic reconfiguration of components using the coordination paradigm. *Future Generation Computer Systems*, 17(8):1023–1038, June 2001.

[30] J. Pérez. *PRISMA: Aspect-Oriented Software Architectures*. PhD thesis, Department of Information Systems and Computation, Polytechnic University of Valencia, 2006.

[31] J. Pérez, N. Ali, J. A. Carsí, , and I. Ramos. Dynamic Evolution in Aspect-Oriented Architectural Models. 3527:59 – 76, 2005.

[32] J. Pérez, N. Ali, J. A. Carsi, and I. Ramos. Designing Software Architectures with an Aspect-Oriented Architecture Description Language. In I. Gorton, G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. A. Szyperski, and K. C. Wallnau, editors, *Component-Based Software Engineering*, volume 4063 of *Lecture Notes in Computer Science*, pages 123–138, Västeras, Sweden, 2006. Springer Verlag.

[33] M. Pinto, L. Fuentes, and J. M. Troya. DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development. In *Generative Programming and Component Engineering (GPCE 2003)*, Lecture Notes in Computer Science, Erfurt, sep 2003.

[34] M. Shaw. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *Workshop on Studies of Software Design*, jan 1994.

[35] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314 – 335, apr 1995.

[36] C. Stirling. Modal and Temporal Logics. In *Handbook of Logic in Computer Science*, volume II. Clarendon Press, 1992.

[37] C. Szyperski. Component software: beyond object-oriented programming. 1998.