PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 13/07

Composing Architectural Aspects based on Style Semantics

Christina von Flach Garcia Chavez
Alessandro Fabricio Garcia
Thais Vasconcelos Batista
Awais Rashid
Cláudio Nogueira Sant'Anna

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 13/07 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena Julho, 2007

Composing Architectural Aspects based on Style Semantics

Christina von Flach Garcia Chavez[1], Alessandro Fabricio Garcia[2] Thais Vasconcelos Batista[3], Awais Rashid[2], Cláudio Nogueira Sant'Anna

[1] Computer Science Department - UFBA, Brazil
[2] Computing Department - Lancaster University, United Kingdom
[3] Computer Science Department - UFRN, Brazil

flach@ufba.br, garciaa@comp.lancs.ac.uk, thais@ufrnet.br,
marash@lancaster.ac.uk, claudios@inf.puc-rio.br

Abstract. An architectural style can be regarded as a paradigm of architectural modularity that encompasses syntactic descriptions, semantic models and constraints over them. In this paper, we analyze the influence exerted by architectural styles over the nature of architectural aspects. We propose style-based join point models that expose high-level join points based on style semantics, with the goal of enhancing composition of architectural aspects in hybrid software architectures. We present a real-life case study involving several styles to demonstrate the expressiveness of the style-oriented join point model. We also assess the scalability of our proposal in the presence of four stylistic composition categories, namely hierarchy, specialization, conjunction, and overlapping. Finally, we discuss the interplay of the style-based architecture composition model and the other conventional models.

Keywords: architectural aspects, architectural styles, style-based composition, semantics-based composition.

Resumo. Um estilo arquitetural pode ser visto como um paradigma de modularidade no nível arquitetural, que agrega descrições sintáticas, modelos semânticos e restrições. Neste artigo, nós analisamos influência exercida por estilos arquiteturais sobre a natureza de aspectos arquiteturais. Nós propomos modelos de pontos de junção baseados em estilo que expõem pontos de junção de alto nível, tendo como base a semântica de estilos arquiteturais, para melhorar a composição de aspectos em arquiteturas de software híbridas. Nós apresentamos um estudo de caso realista que envolve diversos estilos para demonstrar a expressividade de modelos de pontos de junção baseados em estilo. Também avaliamos a escalabilidade de nossa proposta no presença de quatro categorias de composição baseada em estilos, a saber: hierarquia, especialização, conjunção e sobreposição de estilos. Finalmente, discutimos a relação entre o modelo de composição arquitetural baseado em estilo e os outros modelos convencionais.

Palavras-chave: aspectos arquiteturais, estilos arquitetural, composição baseada em estilo, composição baseada em semântica.

In charge of publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

1 Introduction

Architectural aspects are expected to "modularize widely-scoped properties that naturally crosscut the boundaries of system components at the software architecture level" [2, 6, 14]. However, there is little consensus in the research community on what architectural aspects are and on the ways they are related to elements in a software architecture. For example, most aspect-oriented approaches at the architecture level tend to focus on object or component modularity hence mimicking the object and component call graph based join point models in aspect-oriented programming languages. They tend to neglect other well established architecture-level modularity mechanisms such as architectural styles. This means that the architecture specification is no longer a blueprint of systematic decisions and stylistic guidelines governing the design and implementation of the system. Instead an aspect-oriented architecture is reduced to a high-level model of the aspect-oriented program to be developed. Consequently the resulting architectures are brittle in nature making them difficult to evolve and maintain. Furthermore, ignoring well-established architecture modularity mechanisms such as architectural styles makes it difficult to deploy an "aspect-oriented thinking" within an existing architecture design process as architects need to understand the effects of crosscutting in the context of well-known architectural abstractions.

The selection of a particular *architectural style* [30, 35, 4] or combination of styles [28, 34] for a software system has a significant impact on system decomposition and system-wide properties. An architectural style prescribes a main kind of system decomposition and modularization, and adopts distinct component types and connector types, with style-specific semantics. Different architectural styles applied to the same problem can lead to designs with significantly different properties [33]. Since a style prescribes some kind of system decomposition and modularization, it is fair to make the assumption that some system concerns are well modularized while others are not, depending on the choice of architectural style(s). This raises two key questions. Firstly, how is aspect modularity related to style modularity? and, secondly, can style-based semantics of components and connectors serve as a basis of high-level architectural join point models?

In this paper we aim to answer these questions. We first revisit some classical examples of architectural styles from [18] to investigate how the benefits and drawbacks of each architectural style can improve our understanding of architectural aspects (Section 2). We then define a set of *style-based join point models*, each derived from our investigation of the structural and behavioral semantics of a specific architecture style and the nature of crosscutting in the context of that architecture style (Section 3). These style-based join point models facilitate aspect specification and composition in the context of real large-scale and complex systems which are seldom homogeneous. In such systems, different architectural styles may be used for different parts of the system. Hence, pointcut expressions in an aspect crosscutting various architectural elements in such a system are inevitably based on querying over multiple style-based join point models. The pointcut language for our style-based join point models is presented in Section 4. We then present a real-life case study involving several styles for which we have investigated the scalability of our style-based aspect specification and composition mechanism(Section 5). Section 6 discusses some related work before we conclude the paper in Section 7.

2 Architectural styles and aspects

This section presents architectural designs for the KWIC (*Key Word In Context*) index system expressed in three different architectural styles with the twofold goal of: *(i)* contrasting the criteria for system decomposition supported by each architectural style, and *(ii)* analyze the nature of the crosscutting concerns with respect to that style that may emerge at the architectural level. The KWIC has been used in different contexts to illustrate the benefits and drawbacks of design choices driven by modular decomposition [29] and architectural styles [18].

We also use the KWIC system to illustrate the client-server and the layered styles. Although it is not straightforward and perhaps even desirable to structure the KWIC using these styles, we use it as this would allow us to contrast the three styles and showcase different style-specific decompositions and crosscutting in the same context. This discussion supports a better understanding about to what extent a certain widely-adopted architectural pattern exhibits style-specific crosscutting features. Second, the three architectural designs will be also contrasted with respect to their ability to support modular treatment of a similar architectural intriguing concern. In particular, they are evaluated with respect to the ease of evolution to incorporate error handling, a widely-scoped influencing concern at the architectural level.

2.1 Pipe and filter style

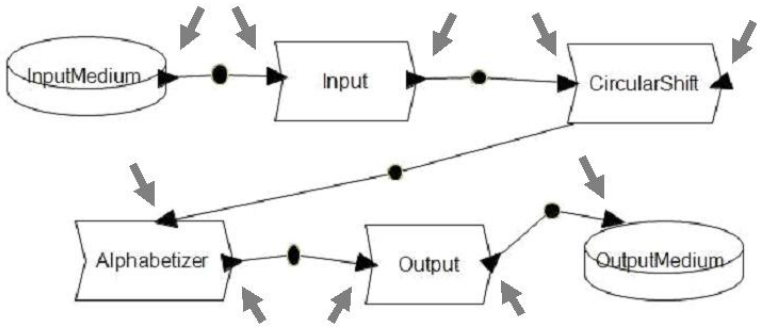

Figure 1: KWIC using a pipe and filter style [18].

Pipe and filter systems are used for processing or transforming data streams. As such, the pipe and filter style prescribes functional decomposition. Components are functional entities ("filters") that usually apply local transformation to their input streams, incrementally, so that output begins before input is consumed. Each filter has a set of inputs and a set of outputs. A filter reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order. The connectors of this style ("pipes") serve as conduits for the streams, transmitting outputs of one filter to inputs of another [18].

Figure 1 presents the KWIC system using the pipe and filter style. The KWIC system reads an ordered set of lines (`Input` filter), and any line may be "circularly shifted" (`CircularShift` filter) by repeatedly removing the first word and appending it at the end of the line. The KWIC system sorts all circular shifts of all lines in alphabetical order (Sort filter) and outputs a listing of these ordered lines (`Output` filter) [29].

There are several reported advantages associated with the use of pipe and filter systems, resulting in enhanced reusability and maintainability of filters [18]. These advantages are aligned with the functional decomposition prescribed by the style. One of its disadvantages is related to data manipulation. The pipe and filter style is function-oriented – therefore, data representation is treated as a secondary concern. Changing the data representation is typically difficult because data type information is spread across filter and pipe interfaces. For the pipe and filter style, the functional concern is well modularized while the data concern is not – it crosscuts several architectural elements, as pointed out by the gray arrows in Figure 1.

Error handling is reported as the Achilles' heel of the pipe and filter style [4]. Since pipes and filters do not share any global state, error handling is hard to address and is often neglected. Moreover, error handling on pipe and filter systems may require resynchronizing the elements when a filter or pipe crashes, and restarting the system.

2.2 Client and server style

The client-server style defines a service-oriented architecture where components can only be a client or a server. They are organized in terms of requesters and providers of services. Servers provide services to clients. A client can only be connected to a server and a server can only be connected to a client. A client requests a service provided by a server, which performs the service and returns the result to the client. Connectors specify the interaction between clients and servers.

Figure 2 presents a client-server version for the KWIC system adapted from the abstract data type (ADT) style [18]. `LineStore`, `CircularShifter` and `Alphabetizer` are servers that provide character-handling, shift and sort services, respectively. `Input` and `Output` are clients that read an input (a set of lines) and write output (ordered lines), respectively.

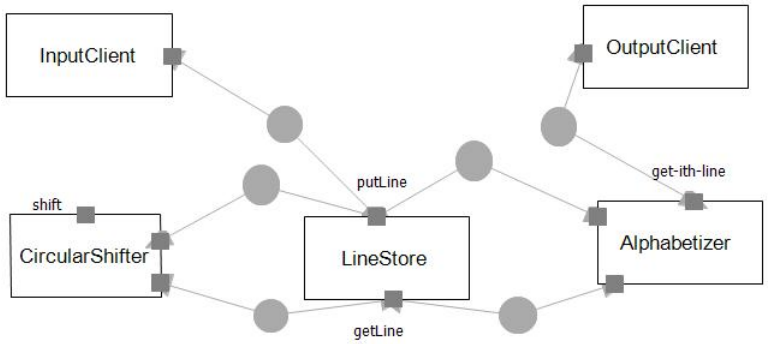

Figure 2: KWIC using the client and server style (adapted from the ADT style [18]).

The main advantages of the client-server style is the facility of their use in distributed systems. Distribution of tasks among servers is straightforward. In addition, it is easy to add new servers. Supporting error handling in the client-server style is a daunting task. There are several points of failures: client invocation at the client side, client invocation at the server side, server reply at the server side and server reply at the client side. For this reason, it is very difficult to apply a similar error handling strategy to all the servers in a modular fashion [4].

2.3 Layered style

The layered style defines a hierarchical organization of the system in layers where each layer provides services to the layer above it [35]. A typical topological constraint of layered systems is to restrict the interactions to adjacent layers and the design of layers is based on increasing levels of abstraction. Connectors specify the interaction between the layers. Network communication protocols are traditional examples of this kind of style.

Figure 3 depicts the KWIC system using the layered style. This example shows that the layered architectural style applies a functional modularization of concepts based on an increasing abstraction level. Each layer has a well-defined function providing services to the upper layers. The layered style presents some advantages: *(i)* it reduces coupling across multiple layers; *(ii)* it hides the inner layers from all except the adjacent outer layer; *(iii)* it allows the interchange of different implementations of the same layer. These features provide benefits to evolvability and reusability.

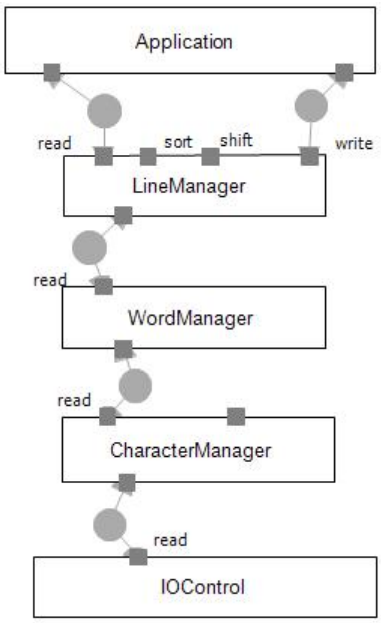

Figure 3: KWIC using a layered style

The main drawback of the layered style is the difficulty of defining the right levels of abstraction on the layers conception. In addition, not all systems are easily structured in layers, as we can see from the example in Figure 3. Communication between layers can lead to performance overhead. For example, to read from the input, the layers must delegate the read tasks to the lower layers until the `I_OControl` provides this service. Error handling can be rather expensive for layered architectures with respect to processing time and programming effort [4]. An error can either be handled in the layer where it occurred or be passed to the next higher layer. In the latter case, the lower layer must transform the error into an error description meaningful to the higher layer [4].

2.4 On the interplay of crosscutting and styles

The kind of decomposition supported by the three styles described may favor the clean modularization of some concerns while others may be not well-modularized. Therefore, styles may interfere with the nature of the crosscutting concerns at the architectural level. Architectural aspects can be style-dependent – e.g., data representation for the pipe and filter style – or not, such as error handling. In fact, error handling has been extensively referred in the literature as a classical crosscutting concern in systems following different kinds of architecture decompositions [22, 26, 37]. Error handling is widely recognized as: *(i)* a global design issue [15, 32], *(ii)* affecting almost all the system modules and their interfaces [7, 27], *(iii)* both in generic and context-specific fashions [7], *(iv)* an anti-modularity factor in several well-known architectural styles [4], and *(v)* exhibits intricate, heterogeneous relationships with the normal system behavior [19, 10].

An architectural style defines a set of properties that are shared by the configurations that are members of the style. These properties can be structural, including a topology, a common vocabulary, the restrictions on the ways this vocabulary can be used in configurations, as well as the semantics of architectural elements. Part of the semantics is also based on the style features. The semantics can be used in a behavioral composition with architectural aspects. In general, architectural composition is based on the structural information that are, in general, static. Behavioral composition goes beyond by introducing a more dynamic approach where the architectural composition is based on semantic information (object computation or interaction). In scenarios such as these ones, architectural aspects show up and may demonstrate their usefulness.

3 Style-based join point models

An architectural style provides distinct component types and connector types, with style-specific semantics. As shown by the examples in Section 2, in a style-based decomposition, crosscutting is driven by the semantics of the specific style. The behavior of the components and connectors employed by a style can, therefore, act as a basis to define a high-level *style-based join point model* for software architecture models expressed in that style. These style-based join points can be exposed and used in quantified expressions, so that *style-based composition* is supported at the architectural level.

In the following sub-sections we present a join point model for each of the three architectural styles from Section 2. The styles are described in Wright [1], an architecture description language based on a formal, abstract model of system behavior. Wright supports the description of architectural styles and the formal specification of the behavior of components and connectors using a notation based on CSP [21]. By using Wright, we make the semantics of the styles explicit so that our style semantics-based joint point models are derived from a formal understanding of the style rather than in an *ad-hoc* way. We first introduce Wright in Section 3.1 before discussing the three representative style-based join point models in Sections 3.2-4.

3.1 Wright specifications

Wright is an architecture description language (ADL) based on the formal description of the abstract behavior of architectural elements [1]. Wright provides explicit notations

for the basic architectural abstractions of components, connectors, and configurations, formalizing the general notions of component as computation and connector as pattern of interaction. In this paper, we use Wright because it supports two important features: (i) the formal characterization of architectural styles, and (ii) the ability to describe the behavior of components and connectors. By using Wright, we are able to provide abstract models of architectural description in which style-specific behavioral properties beyond simple structural descriptions can be specified for individual elements.

3.1.1 Structural specifications

The structure of *component* in Wright consists of a *set of ports* and a *computation*. Each port represents an interaction in which the component may participate. The computation section of a description describes what the component actually does. A *connector* in Wright defines a *set of roles* and the *glue*. A *role* specify the behavior of a single participant in the interaction. The glue of a connector describes how the participants work together to create an interaction. The computation of a component and the glue of the connector represent the behavioral specifications of these elements. A *configuration* is a collection of component instances combined via connectors. The *attachments* define the topology of the configuration, by associating a component's port with a connector's role.

An *architectural style* in Wright defines a set of properties that are shared by the configurations that are members of the style. These properties can include a common vocabulary and restrictions on the ways this vocabulary can be used in configurations. Common vocabulary is introduced by declaring a set of *component and connector types*. Restrictions are defined by means of *constraints* [1]. Wright provides several sets and operators for the specification of constraints, such as `Components` (the set of components in the configuration), `Connectors` (the set of connectors in the configuration), `Name(e)` (the name of element `e`, where `e` is a component, connector, port, or role), `Type(e)`, (the type of element `e`), `Ports(c)` (the set of ports on component `c`), `Computation(c)`, (the computation of component `c`), etc. These sets may be used in the style-based pointcut specifications provided in Section 4.

3.1.2 Behavioral specifications

In Wright, the behavior of components and connectors are specified using a notation based on CSP [21]. CSP components are called *processes*, which interact with each other and with the environment by communication. The basic unit of a CSP behavior specification is an *event*. An event represents an important action. A process is defined in terms of events. Wright adds a notation to CSP to distinguish between initiating an event ($\overline{event}$) and observing an event (written without the overline). Events can carry data. If a process supplies or outputs data the notation used is *event*!*data*. If a process receives or inputs data, the notation used is *event*?*data*. A data carry event does not have to have an overline if it is used by another process.

Processes are described by combining events and simpler processes. If a is an event and P is a process, $a \rightarrow P$ is a process that is initially ready to engage in a and when a occurs, the process will subsequently behave as P. $\surd$ is a special event in CSP that represents the act of terminating the entire system successfully. § is the *success process*, the process that successfully terminates immediately. The behavior of processes can be extended by

means of *choices*. There are two types of choice in CSP: *deterministic choice* ($\Box$) which is resolved by the environment and *nondeterministic choice* ($\sqcap$) which is decided by the process itself. Wright also uses this underlying CSP model, to permit *semantic constraints* on systems by providing sets and operators such as: α`P`: the alphabet of process `P`, α`iP`: the subset of α`P` that is initiated, α`oP`: the subset of α`P` that is observed, `Traces(P)`: the traces of process `P` and `Failures(P)`: the failures of process `P`. Other features from Wright can be found in [1].

3.2 Pipe and filter style

Style Pipe-Filter
  Connector Pipe
    Role Source = DataOutput
    Role Sink = DataInput
    Glue [*Sink will receive data in the same order delivered by Source*]
  Interface Type DataInput =
    $(\overline{\text{read}} \rightarrow (\text{read-data?x} \rightarrow \underline{\text{DataInput}}$
    $\Box\ \text{end-of-data} \rightarrow \overline{\text{close-port}} \rightarrow \S))$
    $\sqcap(\overline{\text{close-port}} \rightarrow \S)$
  Interface Type DataOutput = $(\overline{\text{write-data!x}} \rightarrow \text{DataOutput})$
    $\sqcap\ (\overline{\text{close-port}} \rightarrow \S)$
  Constraints
    $\forall c \in \text{Connectors} \bullet \text{Type(c)} = \text{Pipe} \wedge$
    $\forall c \in \text{Components; p:Port} \mid p \in \text{Ports(c)} \bullet$
      $\text{Type(p)} = \text{DataInput} \vee \text{Type(p)} = \text{DataOutput}$
End Style

Figure 4: Pipe and filter style in Wright [1].

Figure 4 presents a description of the pipe and filter style in Wright. The `Pipe-Filter` style specification introduces one connector type (`Pipe`) and two interfaces types (`DataInput` and `DataOutput`). The constraints of the style refer to these types by name and indicate that (i) all connectors must be pipes and (ii) all components in the system use only `DataInput` and `DataOutput` interfaces on their ports. The informal specification for `DataInput` is *read data repeatedly, closing at or before end-of-data* and for `DataOutput` is *write data repeatedly, closing the port to signal end-of-data.* A `DataOutput` port has two events with which it communicates, `write-data` and `close-port`. Both of these events are initiated by the component, and so they are written with an overbar. The informal description for the glue part (*Sink will receive data in the same order delivered by Source*) indicates the pattern of interaction for Source and Sink. We provide a specification in natural language.

From the example in Figure 4, several high-level, style-specific join points emerge (Table 1). The pipe and filter style allows us to state that "this system is a pipe and filter system, components `CircularShift` and `Sort` are filters". The use of style-specific join points allows the definition of quantified assertions over them. Architects can resort to these join points to describe the ways aspects affect components and connectors. At the

Style	Join point	Where
Pipe-Filter	read data	Filter, DataInput
	write data	Filter, Data Output
	close port	Filter
Client-Server	request service	Client
	receive result	Client
	invoke service	Server
	return result	Server
Layered	receive request	Layer
	delegate subtask	Layer
	return result	Layer
	receive result	Layer

Table 1: Style-based join points.

software architecture level, behavior can be composed with respect to these join points. For example, we can state that, "for all components and connectors in the system `S`, before "read data" or "write data", perform some conversion".

3.3 Client and server style

In client and server systems, components can be clients or servers. Often, a client sends requests to a server and waits for a resource to be sent in response to the request. A server waits for client requests to arrive. If such a request is received, the server processes it and may return a result.

Style Client-Server
  Component Client
    Port r = $\overline{\text{request-service}}$ → receive-result?x → r ⊓ §
    Computation = internalCompute → $\overline{\text{r.request-service}}$
      → r.receive-result?x → **Computation** ⊓§
  Component Server
    Port p = invoke-service → $\overline{\text{return-result!x}}$ → p□§
    Computation = p.invoke-service → internalCompute
      → $\overline{\text{p.return-result!x}}$ → **Computation** □§
  Connector Link
    Role c = $\overline{\text{request-service}}$ → receive-result?x → c ⊓ §
    Role s = invoke-service → $\overline{\text{return-result!x}}$ → s□§
    Glue = c.request-service → $\overline{\text{s.invoke-service}}$ → **Glue**
      □s.return-result!x → $\overline{\text{c.receive-result?x}}$ → **Glue**
      □§
  Constraints
    ∃!s ∈ Components, ∀ c ∈ Components:
    Type(s)=Server ∧ Type(c)=Client ⇒ connected(c,s)
End Style

Figure 5: Client-Server style in Wright.

Figure 5 presents a description of the client-server style in Wright. A component that plays the role `c` defined by the connector type `Link` is allowed to continuously make requests for a service and receive results. The *Glue* part of a connector describes how the participants work together to create an interaction or, in other words, how the partial computations of the components are composed to turn into a larger computation.

The following high-level, style-specific join points emerge from the description provided in Figure 5: "request service", "receive result", "invoke service" and "return result"(Table 1). We can express the need for auditing server operations by stating that "after a server `Serv` invokes a service `S` do some auditing on `Serv`"; we can also express that some data conversion is need in the client side by stating that "before the client `C` receives result `R` do some conversion on `R`".

3.4 Layered style

A layered system is organized hierarchically, each layer receiving requests for services from the layer above it (upper layer) and delegating tasks to the layer below (lower layer). In these systems, the components implement a virtual machine at some layer in the hierarchy and the connectors are defined by the protocols that determine how the layers will interact. Figure 6 presents an specification in Wright for the layered style.

Style Layered
  Component Layer
    Port sr = $\overline{\text{delegate-task}} \rightarrow$ receive-result?x $\rightarrow$ sr $\sqcap$ §
    Port rr = receive-request $\rightarrow \overline{\text{return-result!x}} \rightarrow$ rr$\square$§
    Computation = internalCompute $\rightarrow$
    $\overline{\text{sr.delegate-task}} \rightarrow$ sr.receive-result?x $\rightarrow$ **Computation** $\sqcap$
     rr.receive-request $\rightarrow$ internalCompute
     $\rightarrow \overline{\text{rr.return-result!x}} \rightarrow$ **Computation** $\square$§
  Connector InterLayer
    Role upperRole = receive-request $\rightarrow$
    $\overline{\text{return-result!x}} \rightarrow$ upperRole$\square$§
    Role lowerRole = $\overline{\text{delegate-task}} \rightarrow$
    receive-result?x $\rightarrow$ lowerRole $\sqcap$ §
    Glue = $\overline{\text{upperRole.receive-request}} \rightarrow$
          lowerRole.delegate-task $\rightarrow$ **Glue**
        $\square\overline{\text{lowerRole.receive-result?x}} \rightarrow$
          upperRole.return-result!x $\rightarrow$ **Glue** $\square$§
  Constraints
    $\forall$ c $\in$ Components: Type(c)=Layer
    $\forall$ c $\in$ Connectors: Type(c)=InterLayer
    $\exists$!layer $\in$ Components, $\forall$ c $\in$ Components:
    connected(c,layer)
End Style

Figure 6: Layered style in Wright.

Some of style-specific join points for layered systems are: "delegate subtask", "receive request", "return result" and "receive result" (Table 1). They allow us to specify statements

such as "for all layers in system `S`, before "delegate-subtask" perform some conversion".

3.5 Style-based semantics for architectural aspect composition

In the component-connector view of a system, components are runtime processes and data stores, and connectors are mechanisms that allow the specification of interaction protocols between components. System topology is an important issue but system behavior must be equally considered. Style-based semantics defines a high-level join point model that stands for an abstract model of system behavior at the architectural level. Style-based composition can be defined in terms of such an abstract model.

Formal descriptions of component and connector behaviors such as those provided in Figures 4, 5 and 6 embed the expression of partial ordering between the different types of style-based join points. The use of style-based join point models (Table 1) supports more expressive, fine-grained composition at the architectural level without breaking the encapsulation of components.

Sections 4 and 5 show that style-based composition can lead to more expressive composition capabilities at the architectural level than a conventional programming-oriented model permits.

4 Style-based composition

This section illustrates how the style-based join point models presented in Section 3 can be used for composing architectural aspects. Composition requires mechanisms for selecting join points and for weaving aspects, according to some strategy. In this context, we define a pointcut language that supports quantification over architectural elements and their style-based semantic join points (Section 4.1) and also special connectors that support style-based composition of aspects (Section 4.2).

4.1 A style-based pointcut language

At the software architecture level, we want high-level, style-based join points, that abstract away the semantic model used to specify the behavior of architectural elements. For example, we want to resort to joint points such as "points where a client requests a service", "points where a client receives a result from a server" or "points where a layer delegates a subtask".

Pointcuts are mechanisms for specifying and picking out join points [22] relevant to the modularization of a certain crosscutting concern. Accordingly, we define architecture-level pointcuts as mechanisms for specifying and picking out join points at the architecture level. Pointcuts can be primitive or composite. Composite pointcuts can be defined using the operators $\wedge$, $\vee$, $\neg$. Pointcuts can be defined and named in a Pointcuts section (Figure 7). Primitive pointcut designators can be style-specific or general-purpose, but in this paper, we are interested in style-specific pointcuts.

Several primitive, style-specific pointcut designators are defined to specify join points based on the style semantics. A set of style-specific pointcut designators are presented in Table 2. They are representative examples of designators for the three architectural styles discussed in Sections 2 and 3. For example, the `request-service` pointcut matches points where service requests from clients to servers are made in some client-server configuration.

Style	Designator	Points at which
Pipe-Filter	read-data	data is read
	write-data	data is written or delivered
	end-of-data	end-of-data is signalled
	close-port	input port is closed
Client-Server	request-service	clients request a service
	invoke-service	servers execute a service
	return-result	servers return a result
	receive-result	clients receive a result
Layered	receive-request	layers receive a request
	delegate-task	upper layers delegate task to lower layers
	return-result	lower layers return a result to upper layers
	receive-result	upper layers receive results from lower layers

Table 2: Style-based primitive pointcuts

Similarly, the `close-port` pointcut captures join points at which input port is closed in a piper-filter system.

These examples of style-based pointcuts are based on two key elements of an architectural style [28] *(i)* structure – the form of elements from which a system is composed, including the input and output interfaces of the elements, and *(ii)* behavior – the processing logic of elements by which input data is consumed and output data is produced. However, architectural pointcuts do not necessarily be limited to pick out sets of structural and behavioral join points. They can exploit other complementary style elements, such as: *(iii)* interaction – the means by which data and control are transferred among different elements of a system, and *(iv)* the architectural topology – the paths of interaction among different elements and the rules governing creation and/or removal of the paths.

One important issue concerning join points is the concept of *join point shadow* [20], that is, the static places where join points manifest. In this paper, style-based join points manifest in elements of component or connector interfaces. Therefore, we define that we will be able to map any set of join points to a set of client ports. These ports comprise architectural join point shadows that will be used in the attachments section (Section 3.1.1).

4.2 Composition mechanisms

A pointcut language allows us to specify joint points. The composition between components based on their style semantics is yet to be defined. In this case, we need to resort to some composition mechanism that combines them. In software architecture, there are two candidate places for expressing composition: the attachments section and connector' glue (Section 3.1.1). Since attachments address simple structural component-connector bindings, the connector and its glue part seem to be the most suitable place where style-based composition should be defined.

Figure 7 presents `System1`, a client-server system configuration based on structural properties. In this configuration the attachment statement `S*.p as Retrial.faultyService;` binds the server ports to the connector role `faultyService`. However, we cannot manage to bind the exception handling service w.r.t. the moment a server receives a request from a client or the moment it returns a result

Configuration System1
Style Client-Server
Connector Retrial = {
Role faultyService = {
Invariant: $\#retrials of faultyService \leqslant 3$;
}
Role localStateRecovery;
Glue *localStateRecovery after faultyService*
}
Instances
C1,C2,C3: Client; E: ExceptionHandling;
L1,L2,L3: Link; S1,S2: Server;
Attachments
C1.r as L1.c; S1.p as L1.s;
C2.r as L2.c; S2.p as L2.s;
C3.r as L3.c; S2.p as L3.s;
E.errorLoggingHandler as localStateRecovery;
S*.p as Retrial.faultyService;
End Configuration

Figure 7: Client-Server system configuration in Wright based on structural properties.

Configuration System2
Style Client-Server
Connector Retrial = {
Role faultyService = {
Invariant: $\#retrials of faultyService \leqslant 3$;
}
Role localStateRecovery;
Glue *localStateRecovery after faultyService*
}
Instances
C1,C2,C3: Client; E: ExceptionHandling;
L1,L2,L3: Link; S: Server;
Pointcuts
PortSet: Ports, `where` PortSet = receive-request;
Attachments
C1.r as L1.c; S.p as L1.s;
C2.r as L2.c; S.p as L2.s;
C3.r as L3.c; S.p as L3.s;
E.errorLoggingHandler as localStateRecovery;
PortSet as Retrial.faultyService;
End Configuration

Figure 8: Client-Server system configuration in Wright based on style semantics.

to a client. Figure 8 presents `System2`, an alternative client-server system configuration based on style semantics. We introduce the `Pointcuts` section to allow the specification of user-defined pointcuts to be reused in the Attachments section. The pointcut `PortSet` selects the points where servers receive requests from any client. The meaning of an attachment such as `PortSet as Retrial.faultyService;` is: "bind each server port in the set `PortSet` to the connector role `Retrial.faultyService`". The granularity of the composition (glue behavior w.r.t. receiving a request from a client) will be handled by the connector. At the configuration level, an architect should only deal with attachments between ports and roles.

Figure 9 presents a connector that may deal with the `Retrial` semantics from the example in Figure 8. It specifies the protocol in which a service is retried after service invocations. Figure 10 presents a connector that may be used for composing with respect to the points where servers returns results back to clients. Other examples of style-based composition will be presented in Section 5.

Connector Retrial-after-invoke-service
  Role bRole = invoke-service → $\overline{\text{return-result!x}}$
    → bRole□§
  Role cRole = invoke-service → $\overline{\text{return-result!x}}$
    → cRole□§
  Glue = bRole.invoke-service → $\overline{\text{cRole.invoke-service}}$
    → cRole.return-result → $\overline{\text{bRole.return-result}}$
    → **Glue** □§

Figure 9: Composition mechanisms

Connector Retrial-before-return-result
  Role bRole = invoke-service → $\overline{\text{return-result!x}}$
    → bRole□§
  Role cRole = invoke-service → $\overline{\text{return-result!x}}$
    → cRole□§
  Glue = bRole.invoke-service → $\overline{\text{cRole.invoke-service}}$
    → cRole.return-result → $\overline{\text{bRole.return-result}}$
    → **Glue** □§

Figure 10: Composition mechanisms

5 A case study

This section presents a case study in which we have evaluated the applicability of the style-based join point models (Section 3) and our pointcut language (Section 4.1). Their scalability will be discussed in the presence of style compositions (Section 5.2) in a real-life hybrid software architecture. The case study, called *Health Watcher* (HW), is a Web-based information system for the registration of complaints to the health public system [37],

implemented in Java and AspectJ [22]. We have selected this target application since the application of aspects in this system has shown to be effective to improve the modularization of three widely-scoped conventional crosscutting concerns: persistence [37, 31], distribution [37, 25], and exception handling [10]. In fact, the AO design of Health Watcher has exhibited superior architectural stability even in the presence of heterogeneous evolutionary changes [25], and when compared with non-AO design alternatives [25]. The HW architecture has been previously described [13] using the ACME ADL [17] with some minor extensions for expressing aspectual connectors and configurations [13] (Section 4.2).

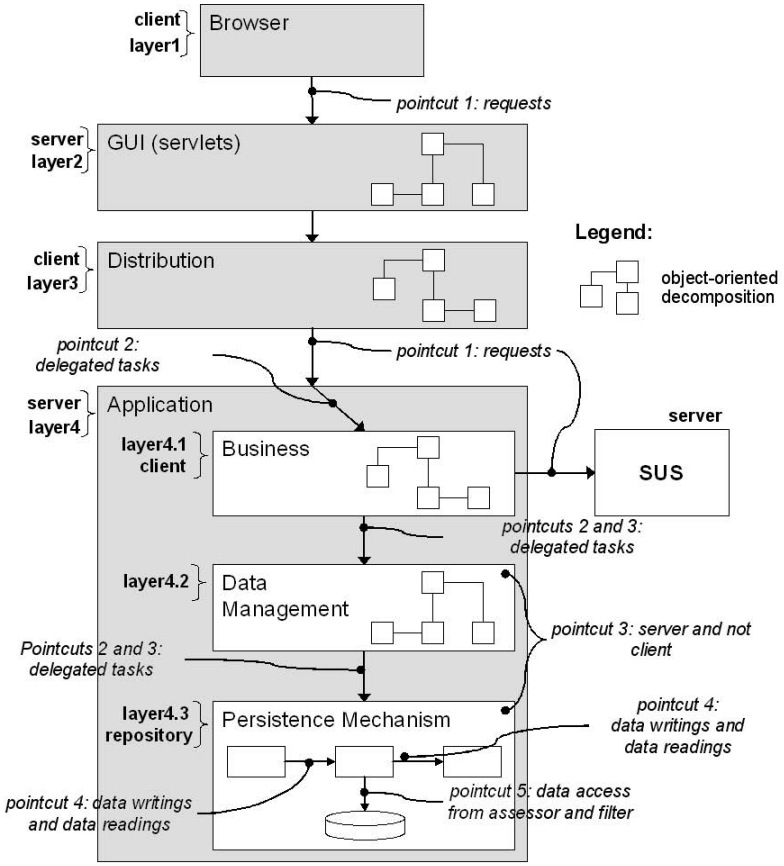

Figure 11: The topology of the Health Watcher system.

In addition, as indicated in Figure 11, the HW architecture involves the instantiation and composition of five different architectural styles: *(i)* client-server, *(ii)* layers, *(iii)* object-orientation, *(iv)* repository, and *(v)* pipe-and-filter. Each of the styles instantiated in the HW architecture provides an architectural-relevant set of join points based on the style semantics (Section 3). Some of those points were in turn explored in the definition of style-based pointcuts for the HW architectural aspects (Section 5.1). Figure 11 presents the overall architecture topology of the HW system. Browsers are located in client hosts which communicate with the servlets in the Web server. The Web server also works as a client by accessing the remote interface exposed by the Application server. The Business layer in the Application server is connected to an external service – the SUS component – that is a centralized system with updated information about the Brazilian Public Health

System. For simplicity, all the interfaces and connector details are omitted in Figure 11.

5.1 Architectural exception handling

The HW system encompasses a number of connectors to modularize the three architectural crosscutting concerns: persistence, distribution, and exception handling. The following sections specifically concentrate on the discussion about connectors employed to the modularization of exception handling policies in the HW architecture. Our choice is driven by the fact that the crosscutting nature of error handling is style agnostic (Section 2.4). Hence, this also allows us to demonstrate the applicability of style-based pointcuts based on different architecture decompositions and hybrid software architectures. The other reasons are that: (i) error handling has been extensively referred in the literature as a classical crosscutting concern [22, 26, 37], and (ii) the benefits and drawbacks of aspectizing error handling using aspect-oriented programming techniques have been well explored nowadays [10, 23, 37].

In the HW system, exception handling has been used as an error-recovery strategy that complements other techniques for improving the system reliability, such as the atomic transaction mechanism (data management layer). In the HW architecture, exception handling promotes the implementation of specialized forward recovery measures, and it is mostly realized by a `ExceptionHandling` component in the HW architectural specification [36, 13]. This component consists of the systemic exception handlers in charge of either putting the system back in a coherent state or recording some relevant information for later manual recovery. In the HW system, a number of exceptions are of architectural nature in the sense they are associated with its coarse-grained design structures, i.e., components and their interfaces. An exception is architectural if it is raised within an architectural component but not handled by the raising component [11, 9]. Such exceptions cut across the boundaries of architectural components. The architectural exceptions that flow between two components determine the "abnormal interaction protocol" to which the involved components adhere.

Figure 12 presents the definition of the main architectural elements to address concerns specific to exception handling in the HW system. Three connectors are used to capture the crosscutting nature in which the abnormal interaction protocols are employed in the HW architecture decomposition [36, 13]: *(i)* termination, *(ii)* retrial, and *(iii)* propagation. The exception handling connectors are special-purpose architecture modularity units that specify unidirectional point-to-point links through which only exceptions flow, i.e., they are "ducts" of exceptions. They denote different forms of composition from the point where they are raised and the point where the system returns to its operation after the handler is executed in the `ExceptionHandling` component. In other words, the exception connectors are ways that the `ExceptionHandling` subsystem collaborates with the rest of the architecture. Notice that such strategies are captured by distinct specifications in the glue clause of each connector. Figure 9 and 10 present two examples of retrial connectors.

Such abnormal interaction protocols are directly or indirectly supported by exception mechanisms in modern programming languages [15], such as Java and C++. The termination and propagation policies are directly supported by the Java and AspectJ programming languages, which were used to implement the HW system. Retrial-based recovery is implemented by explicitly re-invoking the target service from the exception handlers. Different error handling actions are provided by the `ExceptionHandling` component, such as stale

connection handler, server communication fault handler, and information appending handler.

Style-based pointcuts are defined in the HW architecture description to attach architectural elements to "aspectual" connectors modularizing the abnormal collaboration protocols. Figure 11 provides a big picture of the list of pointcuts in the light of the HW architecture topology. For example, the pointcut 1 exploits the join point model defined by the client-server style in order to pick out all the exceptional events not successfully handled by the *servers* before they are propagated as results to the *clients*. Several failures might happen in the execution of the services made available by all the system servers. This pointcut is formally described in Figure 12 (lines 29-30). The `Retrial-before-return-result` connector (Figure 10) is the protocol attached to such a pointcut. The goal is to: *(i)* activate a default handler in charge of logging relevant information associated with server faults, and *(ii)* retrying the service execution again before they are propagated as exceptional results to the *clients*. This systemic error handling strategy crosscuts all the *result returns* from the three *server* instances defined in the HW architecture.

Note that the `Retrial-after-invoke-service` connector (Figure 9) could be also applied at pointcut 1. On the other hand, we cannot specify a composition based on `request-service` join points (Table 1) as the exceptional events need to be handled at the server side in the HW architecture. We need to capture all the `return-results` (Table 1), exactly before the results are returned to the clients. The idea is that we have contextual information for logging details (e.g., time of the request and faulty service) related to specific service failures. Hence, a default handler is in charge of logging those relevant information associated with server faults before they are propagated as exceptional results to the clients. Note that this generic exception handling strategy crosscuts all the *result returns* from the three *server* instances defined in the HW architecture. Figure 12 describes a list of the pointcuts used to attach the affected architectural elements to the aspectual connectors involved in exception handling.

5.2 Composition of styles

Styles can be composed with each other in several ways. The HW architecture involves the combination of several kinds of styles which directly determines the set of architectural join points available. The HW system exhibits four types of style composition: *(i)* hierarchy (Section 5.2.1), *(ii)* overlapping (Section 5.2.2), *(iii)* conjunction (Section 5.2.3), and *(iv)* specialization (Section 5.2.4). The following subsections discuss the interplay of these composition types and the style-based join points in the context of the HW architecture.

5.2.1 Hierarchical composition

In the HW design, distinct styles were used at different levels of architectural hierarchy, so that the internal architecture of certain components was defined in a different style than its surroundings. This kind of inter-style combination is named hierarchical composition. Hierarchical composition provides a scoping mechanism for defining architectural pointcuts that applies exclusively to enclosing and/or enclosed elements. For instance, the Application server is internally structured as a set of layers. In this case, the encapsulation boundary of the Application component insulates the join points of that particular instance of the Layers style from the rest of the architecture, which is realizing other architectural

```
1 System ...
2   Component ExceptionHandling = {
3     Port communicationFaultHandler = {..}
4     Port errorLoggingHandler = {..}
      Port informationAppendingHandler = {..}
      Port staleConnetionHandler = {..}
5   }
6   Connector Termination = {
7     Role faultyService ;
8     Role terminationBasedRecovery = {
9       Invariant: returnValue.type ≠ Exception;
10    };
11    Glue terminationBasedRecovery
12          after-terminate faultyService;
13  }
14  Connector Retrial-before-return-result = {
15    Role faultyService = {
16        Invariant: #retrialsoffaultyService ⩽ 3;
17    }
18    Role localStateRecovery;
19    Glue localStateRecovery
20          after-retry faultyService;
21  }
22  Connector Propagation = {
23    Role faultyService;
24     Role exceptionTransformation =
       Invariant: returnType.value = Exception;
25    Glue exceptionTransformation
26          after-propagate faultyService;
27  }
28   Pointcuts
29    PortSet1: Ports, Failures(P), ∀c ∈ Components: Type(c)= Server,
30    where PortSet1 = return-result
31    PortSet2: Ports, Failures(P), ∀c ∈ Components: Type(c)= Layer,
32     where PortSet2 = delegate-task ∧ c inside Server 33     PortSet3:  Ports,
Failures(P), ∀c ∈ Components: Type(c)= Layer ∧ Type(c) = ¬ Client
34     where PortSet3 = delegate-task ∧ c inside Server 35     PortSet4:  Ports,
Failures(P), ∀c ∈ Components: Type(c)= hybrid(accessor, filter),
36    where PortSet4 = data-access 37  }
38  Attachments { .. }
39  End System
```

Figure 12: HW Architecture Elements for Exception Handling.

styles. Hence, the set of style-based join points in the internals of the Application component is determined by the semantics of the Layers style, in addition to the particular names of its inner interfaces, connectors, and components.

One of the pointcut descriptions for exception handling in HW has selected all the exceptions in *delegated tasks* to *layers* inside the Application *server*. It is illustrated as pointcut 2 in Figure 11, and formally described in Figure 12 (lines 31-32). This architectural pointcut picks up only join points that are exceptional events in components that are defined as layers inside the Application component. The purpose is to associate this pointcut with the handler used to append relevant layer-specific information after the occurrence of such exceptional events and before they are propagated through different layers in the internal Application server architecture, so that the topmost handler has enough information to implement the error recovery behavior. Note that we relied on a specific operator `inside` to capture specific join points involved in a hierarchical architecture composition (line 32 in Figure 12).

5.2.2 Overlapping

The most recurring use of multiple styles in the HW architecture was the creation of architectural elements that had multiple types, each type taken from a different style. Such architectural elements form an overlapping zone of styles, and they embody vocabulary and satisfy constraints from multiple styles. We can say that style-specific types are superimposed in those components since they are assuming multiple responsibilities defined by different styles. The presence of overlapping-based compositions in the HW architectures was exploited to express certain useful style-based pointcuts for architectural error handling strategies.

For example, it was used to quantify over exceptional events returned as results of *task delegations* in the internal *layers* of the Application *server* with no external communication to other *servers*. This pointcut is illustrated as pointcut 3 in Figure 11, and described in Figure 12 (lines 33-34) using our pointcut language. In other words, it includes any exception raised by tasks executed by the layers PersistenceMechanism and DataManagement. On the other hand, it excludes the Business layer since it is a client of the SUS server; the Business layer is the overlapping zone here since it plays both the roles of client and layer. This particular pointcut was used to determine that all the internal server exceptions should follow the termination policy.

5.2.3 Conjunction

Another style composition category found in the HW architecture design was conjunction, which results in hybrid styles [28]. It involves the combination of two styles through the union of their design vocabularies, and conjoining their constraints. When more than one style is used as a supertype, the new style must be a substyle of the conjunction of the parent styles. In such cases it may be necessary to also define new types of components or connectors that pertain to more than one style. Hence, the hybrid join point model is defined by the resulting types defined by the conjunction.

A stylistic conjunction in HW design was the symbiosis involving the pipe-and-filter and repository styles, which encompassed the addition of the components defined by the repository style (accessors and databases) to a pipe-filter system. Some internal com-

ponents of the persistence mechanism had filter-like behaviors, while also accessing the system database. That is, those components are subtypes of filter and accessor types. Pointcuts were defined in the HW architecture to capture exceptions returned as result to all the *data accesses* to components in which their types are *hybrid*, i.e., conjunctions of *accessors* and *filters*. Figure 12 (lines 35-36) defines this pointcut using a special-purpose composition operator `hybrid`. In addition, pointcuts were defined in the HW architecture to capture database connection exceptions associated with *data writing* that were raised through interfaces that are *subtypes* of *streams*. The overall employed strategy was the retrial of the target database services by using the stale connection handler.

5.2.4 Specialization

Specialization is a kind of relationship in which a style is a substyle of another by strengthening the constraints, or by providing more specialized versions of some of the element types. For instance, a pipeline style might specialize the pipe-filter style (Section 2.2) by prohibiting non-linear structures and by specializing a filter element type to a pipeline "stage" that has a single output port. Style specializations lead us to think what happens with the original join point model of the new style. In general, the family of join points is a superset of the join points defined by the parent style since types may be subtypes of other types, with the interpretation that: *(i)* a subtype satisfies all of the architectural properties of its supertype(s), and *(ii)* that subtype respects all of the contraints of those supertypes.

6 Discussion and related work

The characterization of style-based compositions for aspect-oriented software architectures provides more expressive means for software architects document the crosscutting impact of certain components over the architectural decomposition. The definition and the rationale behind style-based join point models is the result of the analysis of existing AO ADLs [3, 13] and our extensive experience over the last six years designing and assessing aspect-oriented architectures for a number of distinct application domains. Such AO applications include a reflective middleware system [5], multi-agent systems [16], a design measurement tool [5, 8], a context-sensitive tourist guide system [12], a CVS system [10], and product lines for quality control measurements [24] and J2ME games [24].

Style-Based Join Points and Existing ADLs. Even though the nature of several crosscutting concerns is dependent on the chosen architectural styles (Section 2) and the rules governing each family of software architectures, the existing architecture specification approaches do not fully offer support for style-based pointcut specifications. The exploitation of style-based join point models seems to address better at the architectural level the recurring problem of pointcut fragility that is common concern in conventional programming-level join point models. For example, if the name of a certain interface or component changes, syntax-based pointcut declarations are likely to be undesirably broken in the presence of architecture maintenance and evolution. In our previous empirical study on architecture stability [25] based on the HW sytem, we observed that style-based pointcuts tended to be more stable in the presence of widely-scoped design changes because the frequency of syntactical changes in architecture descriptions occur much often than

the stylistic architectural choices [25]. It was evidenced by the nature of changes found in the deployed version of the HW system. Of course, it does not impede architectures of using conventional pointcut designators in conjunction with style-based designators. For example, a `within` pointcut could be defined to match points in the execution that belong to any kind of configuration, component or connector.

Architectural Aspect Interactions. In HW case study, the architecture exhibited some points of interaction involving two or more crosscutting concerns. The interactions manifested in direct and indirect forms. The direct forms included situations where explicit connectors were used to bind the components modularizing the crosscutting concerns. For instance, some required interfaces in the `ExceptionHandling` component access the `AbortOperation` port provided by the `TransactionManagement` component in the `DataManagement` Layer. The indirect interactions involved two or more aspectual connectors which have join points in common. Such interactions were resolved at the configuration-level architectural description using extra operators in the attachments part [13].

As a result, we have extended the ACME language to support two basic kinds of resolution operators at the attachment level for solving interaction conflicts: `precedence` and `XOR`. There were situations where a precedence was specified as valid for the whole architecture, and scenarios where they were defined for specific join points. For example, the `Retrial` connector has precedence over the `Termination` connector at all the join points they have in common, while at the port `savingService`, the `Termination` connector is tried first and, secondly, the backward recovery with abort in case the exception was not successfully handled. When there is a precedence relation between two connectors X and Y, where the execution of Y depends on the satisfaction of a condition associated with X, the architect can explicitly document it using a condition statement together with the `around` glue in X. The `XOR` operator was used to indicate, for instance, that for every shared join point, only one of the either termination or retrial should be non-deterministically chosen.

7 Conclusions

This paper discussed the influence of architectural styles in the definition of architectural aspects. The selection of an architecture pattern determines the design space for the particular problem at the architect hands. Like any abstraction, the chosen style emphasizes some concerns of the problem and suppresses others [33]. Some concerns are expected to be well localized within specific kinds of modular units defined by the style, while others are expected to crosscut their boundaries. We have concentrated our discussion in three well-known architectural styles: pipe-and-filter, client-server and layered patterns. Based on the formal specification of these styles, we defined a style-based architectural pointcut language and a composition mechanism. We have evaluated the applicability of our proposal using the Health Watcher system where we could exploit the combined use of architectural styles in pointcut definitions. By exploring exception handling scenarios that affects this example, we have illustrated the expressiveness of the style-based composition model in the presence of different kinds of style combinations.

We have concluded that: *(i)* architectural aspects are widely-scoped properties that naturally crosscut the boundaries of system components with respect to the main kind of decomposition prescribed by an architectural style; *(ii)* for each architectural style, there

are particular join points for the style-specific component and connectors ; *(iii)* aspectual composition is expressed in the attachments section and in the glue clause inside the connector. While the attachments section models structural composition, the glue part models style-based behavioral composition.

Although we have used the Wright ADL notation in this work, the presentation of the style-based join point models and composition in this paper are enough generic. As a result, they can be easily applied using other notations. For instance, there is a straightforward conversion from Wright to the ACME ADL via a direct tool interchange support defined in [17]. Similarly, other formal definition languages can be used instead of CSP to define the style-specific behavior. As a future work we intend to formally specify the aspect-oriented architectural style in order to document the family of aspect-oriented software architectures.

References

[1] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[2] E. Baniassad, P. C. Clements, J. Araujo, A. Moreira, A. Rashid, and B. Tekinerdogan. Discovering early aspects. *IEEE Software*, 23(1):61–70, 2006.

[3] T. Batista, C. Chavez, A. Garcia, C. Sant'Anna, U. Kulesza, A. Rashid, and F. Filho. Reflections on Architectural Connection: Seven Issues on Aspects and ADLs. In *Workshop on Early Aspects - Aspect-Oriented Requirements Engineering and Architecture Design*, Shangai, China, May 2006.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[5] N. Cacho, C. Sant'Anna, E. Figueiredo, A. Garcia, T. Batista, and C. Lucena. Composing design patterns: a scalability study of aspect-oriented programming. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 109–121, New York, NY, USA, 2006. ACM Press.

[6] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. P. Alarcon, J. Bakker, B. Tekinerdogan, and A. J. Siobhán Clarke and. Survey of aspect-oriented analysis and design approaches. Technical Report AOSD-Europe-ULANC-9, AOSD-Europe, May 2005.

[7] F. Cristian. A recovery mechanism for modular software. In *ICSE '79: Proceedings of the 4th international conference on Software engineering*, pages 42–50.A, Piscataway, NJ, USA, 1979. IEEE Press.

[8] E. Figueiredo, A. Garcia, and C. J. P. de Lucena. AJATO: an AspectJ Assessment Tool. In *Proc. of the ECOOP.06, Demo Session*, Nantes, France, July 2006.

[9] F. Filho, P. H. S. Brito, and C. M. F. Rubira. Modeling and Analysis of Architectural Exceptions. In *FM'2005 Workshop on Rigorous Engineering of Fault-Tolerant Systems*, pages 112–121, Newcastle, UK, July 2005.

[10] F. Filho, N. Cacho, R. Ferreira, E. Figueiredo, A. Garcia, and C. Rubira. Exceptions and Aspects: The Devil is in the Details. In *Proc. of FSE-14, 14th International Conference on Foundations on Software Engineering*, 2006.

[11] F. Filho, P. H. da S. Brito, and C. M. F. Rubira. Specification of Exception Flow in Software Architectures. *Journal of Systems and Software - Special Issue on Architecting Dependable Systems (Accepted for publication)*, 2006.

[12] A. Garcia, T. Batista, A. Rashid, and C. Sant'Anna. Driving and Managing Architectural Decisions with Aspects. In *SHARK.06 workshop, ICSR.06 Conference*, Turin, Italy, June 2006.

[13] A. Garcia, C. Chavez, T. Batista, C. Sant'Anna, U. Kulesza, A. Rashid, and C. Lucena. AspectualACME: An Architecture Description Language for Aspect-Oriented Software Architectures. In *European Workshop on Software Architecture (EWSA 2006)*, Nantes, France, September 2006.

[14] A. Garcia and C. Lucena. Taming heterogeneous agent architectures with aspects. *Communications of the ACM*, October 2006.

[15] A. Garcia, C. M. F. Rubira, A. Romanovsky, and J. Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, 2001.

[16] A. Garcia, C. Sant'Anna, C. Chavez, V. T. da Silva, C. J. P. de Lucena, and A. von Staa. Separation of Concerns in Multi-agent Systems: An Empirical Study. In *SELMAS*, pages 49–72, 2003.

[17] D. Garlan, R. T. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[18] D. Garlan and M. Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

[19] J. B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.

[20] E. Hilsdale and J. Hugunin. Advice weaving in aspectj. In *AOSD'04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM Press.

[21] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice–Hall, 1985.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–355. Springer, 2001.

[23] J. Kienzle and R. Guerraoui. AOP: Does it Make Sense? - The Case of Concurrency and Failures. In *Proceedings of ECOOP 2002*.

[24] U. Kulesza, V. Alves, A. Garcia, C. J. P. de Lucena, and P. Borba. Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming. In *Proceedings of the 9th International Conference on Software Reuse (ICSR'06)*, Turin, Italy, June 2006.

[25] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. von Staa, and C. Lucena. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In *Proceedings of 9th International Conference on Software Maintenance - ICSM06*, 2006.

[26] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning, 2003.

[27] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 418–427, New York, NY, USA, 2000. ACM Press.

[28] N. R. Mehta and N. Medvidovic. Composing architectural styles from architectural primitives. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 347–350, New York, NY, USA, 2003. ACM Press.

[29] D. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[30] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[31] R. Rashid, A. Chitchyan. Persistence as an aspect. In *AOSD Conference*, pages 120–129, 2003.

[32] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.*, 12(2):191–221, 2003.

[33] M. Shaw. Comparing architectural design styles. *IEEE Softw.*, 12(6):27–41, 1995.

[34] M. Shaw and P. Clements. Toward boxology: preliminary classification of architectural styles. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, pages 50–54, New York, NY, USA, 1996. ACM Press.

[35] M. Shaw and D. Garlan. *Software Architecture: Perspective on an Emerging Discipline.* Prentice Hall, 1996.

[36] S. Soares. *An Aspect-Oriented Implementation Method.* PhD thesis, Federal Universisty of Pernambuco, Brazil, October 2004.

[37] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with aspectj. In *OOPSLA*, pages 174–190, 2002.