Composing Architectural Aspects Based on Style Semantics

Christina Chavez[1]
flach@dcc.ufba.br

Alessandro Garcia[2]
afgarcia@inf.puc-rio.br

Thais Batista[3]
thais@ufrnet.br

Marcel Oliveira[3]
marcel@dimap.ufrn.br

Claudio Sant'Anna[1]
santanna@dcc.ufba.br

Awais Rashid[4]
marash@lancaster.ac.uk

[1]Computer Science Department, Federal University of Bahia, Brazil
[2]Computing Department, Pontifical Catholic University of Rio de Janeiro, Brazil
[3]Computer Science Department, Federal University of Rio Grande do Norte, Brazil
[4]Computing Department, Lancaster University, United Kingdom

ABSTRACT

The lack of architecturally-significant mechanisms for aspectual composition might artificially hinder the specification of stable and reusable design aspects. Current aspect-oriented approaches at the architecture-level tend to mimic programming language join point models while overlooking mainstream architectural concepts such as styles and their semantics. Syntax-based pointcuts are typically used to select join points based on the names of architectural elements, exposing architecture descriptions to pointcut fragility and reusability problems. This paper presents style-based composition, a new flavor of aspect composition at the architectural level based on architectural styles. We propose style-based join point models and provide a pointcut language that supports the selection of join points based on style-constrained architectural models. Stability and reusability assessments of the proposed style-based composition model were carried out through three case studies involving different styles. The interplay of style-based pointcuts and some style composition techniques is also discussed.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Languages (e.g., description, interconnection, definition)*

General Terms

Design, Languages, Verification

Keywords

Architectural aspects, architectural styles, pointcut languages, style-based composition

AOSD'09, March 2–6, 2009, Charlottesville, Virginia, USA.

1. INTRODUCTION

Architectural aspects are components that modularize widely-scoped properties that naturally cut across one or more of the elements in a given architectural decomposition [1, 7, 9]. The importance of taming architectural aspects is attested by the recent interest of the software architecture community on understanding how aspects influence architectural descriptions (ADs) [11, 9] and by the growing number of aspect-oriented (AO) architecture description languages (ADLs) [7, 2]. In this context, *aspect composition* requires the identification of architecturally-relevant join points in which aspects and other architectural elements are naturally combined together.

However, current AO ADLs overlook mainstream architectural concepts such *architectural styles*, and aspect composition resorts to join point models that mimic low-level programming language join point models. This may lead to a conceptual misalignment between the techniques, models and ADLs used by architects to document architectures and the expressiveness of AO approaches. For instance, architects commonly organize the components of large software systems on the basis of one or more architectural styles. An architectural style encapsulates important decisions about architectural elements and emphasizes important constraints on these elements and their relationships. Architectural styles allow architects to reuse and apply successful design knowledge to a particular class of systems, with the support of style-specific tools, analysis, and implementations [10]. Moreover, *syntax-based pointcuts* are typically used by current AO ADLs to select join points based on the names of architectural elements, exposing AD to pointcut fragility [21, 19] and reusability problems. The characteristics and limitations of low-level join point models and syntax-based pointcuts that affect aspect composition at the architectural level are also discussed (Section 2).

In this context, we present *style-based composition*, a new flavor of aspect composition at the architectural level based on the semantics of architectural styles. Style-based aspect composition is defined in terms of *style-based join point models* (Section 3), high-level join point models that stand for abstract models of system behaviour at the architectural level. Such abstract models entail significantly different sets of join points from those supported by existing AO ADLs

and AspectJ. For each style, a join point model needs to be defined and documented. The documentation makes the style semantics explicit so that the style-based joint point models can be used for architectural composition and analysis. In style-based aspect composition, pointcut definitions are inevitably based on querying over multiple style-based join point models. They also require an open-ended *pointcut language* that takes into account these join point models (Section 4).

Style-based join point models are expected to facilitate aspect specification and composition in the context of real large-scale and complex systems which are seldom homogeneous. Style-based pointcuts are expected to reduce pointcut fragility and promote reuse. Stability and reusability assessments of the style-based composition model are presented, based on the outcomes of three case studies involving different styles and application domains: the *Health Watcher (HW)* system [33], the *Conference Management (CM)* system [29], and the *MobileMedia (MM)* [12] product line for data manipulation in mobile devices. Our analysis also discusses the interplay of style-based composition and some existing techniques for composing architectural styles in the context of the HW system (Section 5). Finally, our proposal is compared with related work (Section 6) before disclosing final remarks and future work directions (Section 7).

2. MOTIVATION

The *Health Watcher* (HW) design (Section 2.1) is a real-life case study that involves several architectural styles and different architectural aspects, including exception handling (Section 2.2). Architectural descriptions for the HW system are provided in AspectualACME [14], an AO ADL that extends ACME [17] for expressing aspectual interactions and configurations (Section 2.3). The HW design is used to discuss characteristics and limitations of low-level join point models and syntax-based pointcuts for composing aspects at the architectural level (Section 2.4).

2.1 The Health Watcher: a case study

Health Watcher (HW) is a Web-based information system that supports the registration of complaints to the Brazilian Public Health System [33]. HW is implemented in AspectJ [22] and has been used as a benchmark for AO software development [33, 13, 25, 14, 18] due to the heterogeneity of the crosscutting concerns found in its implementation. Figure 1 illustrates a primary view of the HW runtime architecture. It combines four classical architectural styles: *(i)* client-server, *(ii)* layers, *(iii)* shared-data, and *(iv)* pipe-and-filter. The first layer consists of browsers, located in client hosts. Browsers communicate with servlets (GUI) in a Web server, via HTTP connection. The Web server (the second layer) accepts connections, processes requests, and returns resulting data to clients. The Web server also works as a client and accesses the remote interface exposed by the `Application` server (the third layer), which consists of business logic services.

The `Application` server is also organized in layers. The upper layer implements the core business rules and is connected to an external server, the `SUS` component. SUS is a centralized system with updated information about the Brazilian Public Health System. The middle layer, `Data-Management`, manages data services and accesses a relational

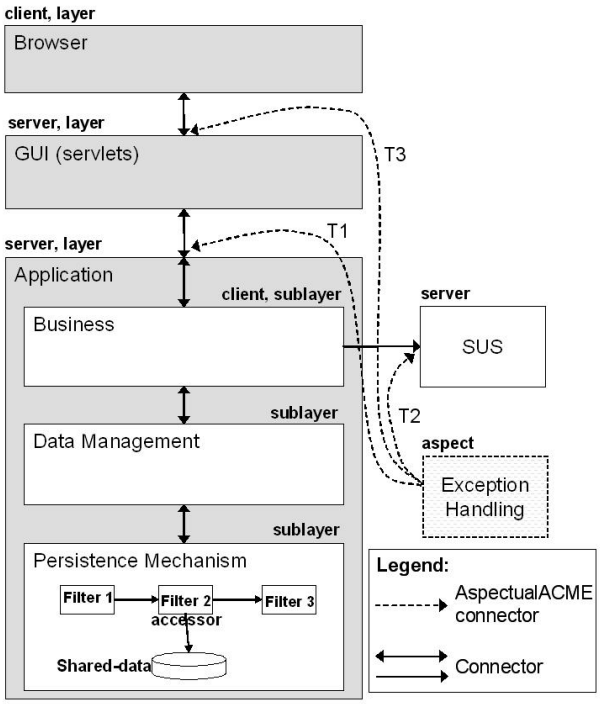

Figure 1: The topology of the HW system.

database used as the persistence mechanism. The lower layer, `Persistence Mechanism`, combines two styles to represent the inner components: shared-data and pipe-and-filter. For simplicity, interfaces and connector details are omitted in Figure 1.

The AO design of Health Watcher has exhibited superior architectural stability even in the presence of heterogeneous evolutionary changes when compared with non-AO architecture alternatives [19]. Aspects were used in the HW system to improve the modularization of three widely-scoped crosscutting concerns: exception handling [13], transaction management [33], and distribution [33, 25]. The next subsection focuses on discussing the crosscutting nature of the `ExceptionHandling` component in the HW architecture.

2.2 Architectural exception handling

Error handling (EH) [26] has been extensively referred in the literature as a classical crosscutting concern in systems following different kinds of design decompositions, such as layered architectures, MVC architectures, and data shared architectures [22, 26, 13, 33]. Error handling is widely recognized as a global design issue [15, 31] and tends to affect almost all the system's core modules and their interfaces [26]. It has also been documented as an anti-modularity factor in well-known catalogues of architectural patterns [4].

Exceptions are of architectural nature whenever they are associated with coarse-grained 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 [13]. 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.

Listing 1: HW Architecture Elements for EH

```
System HW = {
 Component ExceptionHandling = {
   Port communicationFaultHandler = {...}
   Port errorLoggingHandler = {...}
   Port informationAppendingHandler = {...}
   Port staleConnectionHandler = {...}
 }
 Connector Type Termination = {
   Role faultyService;
   Role terminationBasedRecovery = {
    Invariant returnValue.type !> Exception; }
   Glue terminationBasedRecovery after
        faultyService;
 }
 Connector Type Retrial = {
   Role faultyService = {
    Property retrials: int;
    Invariant retrials faultyService <= 3; }
   Role localStateRecovery;
   Glue localStateRecovery after faultyService;
 }
 Connector Type Propagation = { ... }
 Attachments { ... }
}
```

In the HW system, a number of exceptions are of architectural nature and are associated with several components and their interfaces. EH has been used as an error-recovery strategy that complements other techniques for improving system reliability, such as the atomic transaction mechanism in `DataManagement` layer. EH promotes the implementation of specialized forward recovery measures, and it is mostly realized by a `ExceptionHandling` component in the HW architectural specification (Figure 1). This component consists of systemic exception handlers in charge of either putting the system back in a coherent state or recording some relevant information for later manual recovery [33, 14]. Figure 1 shows the `ExceptionHandling` component, and how it affects multiple layers and components of the HW design.

2.3 Exception handling in AspectualACME

Listing 1 presents a textual description in AspectualACME for the `ExceptionHandling`'s key elements used in the HW system. AspectualACME extends ACME and, therefore, supports the description of systems, components, connectors, attachments, ports, roles, and representations. The HW system is composed of components, connectors and attachments that describe its topology. Components such as `Exception Handling`, *Application*, *SUS* and *GUI* are the computational units and support multiple interfaces known as *ports*. The ports of the `ExceptionHandling` component provide handlers for the exceptions raised by the other HW components. Examples of handlers are `communicationFaultHandler`, `informationAppendingHandler` and `staleConnectionHandler` (Listing 1).

Connectors implement the interaction protocols between components. Connectors have *roles*, which are associated to component ports. *Attachments* define a set of port/role associations. Three connectors are used to capture the abnormal interaction protocols employed in the HW architecture [33, 14]: (i) `Termination`, (ii) `Retrial`, and (iii) `Propagation` (Listing 1). These connectors are special-purpose units of modularity that specify links through which only exceptions flow, i.e., they are "ducts" of exceptions. They denote different forms of interaction between the point where the exception is raised (exception throwing port) and the handler execution at the `ExceptionHandling` component. In other words, the exception connectors determine the abnormal interactions that can take place between the `ExceptionHandling` component and the rest of the architecture. 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 Java and AspectJ, the programming languages used to implement the HW system. Retrial-based recovery is implemented by explicitly re-invoking the target service within the exception handlers.

AspectualACME defines a few additional abstractions and mechanisms to support the description of aspectual compositions. Aspects are represented as conventional components. Extensions affect only connectors and attachments. The *glue clause* describes the details of aspectual interactions. The aspect composition strategy (after, before or around) is localized inside connectors [14]. Different composition protocols for EH are captured by distinct specifications in the glue clause of each connector. For instance, the glue clause from the `Termination` connector (Listing 1) determines that, once the exception (received through the `faultyService` role) is processed by the component handler attached to the `terminationBasedRecovery` role, the faulty service is discontinued and a normal value is returned to the caller of the faulty service. This normal return is enforced by the invariant associated with the `terminationBasedRecovery` role. Similar reasoning applies to the `Retrial` and `Propagation` connectors. However, the `faultyService` is re-invoked (maximum of three times) in the first case, and an exception is always propagated to the caller in the second case.

2.4 Limitations of AO ADLs

In most AO ADLs, aspect compositions rely on *low-level join point models* and *syntax-based pointcut definitions* that depend on specific naming conventions or resort to enumeration of syntactic ADL elements, such as identifiers for components, connectors, ports and roles.

Listing 2 presents an example of AD in AspectualACME. In this example, the join point model adopted is style-agnostic and based on require service/provide service semantics. It does not take into account possible points of interest prescribed by different architectural styles and their semantics. Moreover, each attachment declaration relates one port to one role, matched by their full signatures. The AD is verbose and difficult to reuse across multiple instantiations of the `ExceptionHandling` component.

AspectualACME and other ADLs provide AspectJ-like wildcards in order to select a set of components and their ports, promoting the adoption of naming conventions and possibly better descriptions of intended composition semantics. For instance, servers can be renamed to `Application-Server`, `GUIServer`, and `SUSServer`, and the lexical-based pattern *Server can be used to match those elements. Ports can also be renamed to `*Service`. Listing 3 presents an AD in AspectualACME that uses wildcards.

Previous analysis of existing AO ADLs [2, 7, 14] revealed that their support for pointcut definitions is mostly based on the syntax of the ADs. Some AO ADLs [2] support the selection of multiple elements based on other structural prop-

Listing 2: Conventional aspect composition for EH

```
System HW = {
 Component Application = { ... }
// Other components (SUS, GUI)

 Component ExceptionHandling = { ... }
 Connector T1,T2,T3: Termination;

 Attachments {
   T1.terminationBasedRecovery to
     ExceptionHandling.communicationFaultHandler;
   T2.terminationBasedRecovery to
     ExceptionHandling.communicationFaultHandler;
   T3.terminationBasedRecovery to
     ExceptionHandling.communicationFaultHandler;
// Binding the connectors to ports that raise
// exceptions to be propagated to upper layers

   T1.faultyService to Application.Facade;
   T2.faultyService to SUS.InfirmaryList;
   T2.faultyService to SUS.HospitalList;
   T3.faultyService to GUI.ComplaintRegistration;
   ...
 }
}
```

Listing 3: Using Wildcards in AspectualACME Pointcuts

```
System HW = {
 ...
 Attachments {
   T1.terminationBasedRecovery to
     ExceptionHandling.communicationFaultHandler;
   T1.faultyService to *Server.*Service;
 }
}
```

erties, such as all provided interfaces, all required interfaces, or all ports attached to a particular role. However, pointcuts are style-agnostic.

The use of syntax-based pointcuts leads to some liabilities. First, the use of name conventions in pointcut expressions becomes awkward when somebody is trying to understand the rationale motivating the composition. Second, the architecture configuration specification also becomes brittle in the presence of changes [19] and often leads to the well-known pointcut fragility problem [21]. Third, support for describing architectural connections in ADLs should allow us to specify recurring cases of architectural compositions of components and connectors so that they could be reused in different contexts. This means that the descriptions of recurring pointcuts in attachments relative to a crosscutting concern, such as EH, should not be coupled to the actual names or structural properties of a particular AD: it should express the composition purpose.

In the next sections, we present our approach to aspect composition based on style semantics, and style-based join point models and style-based pointcuts are motivated, defined and evaluated.

3. STYLE-BASED JOIN POINT MODELS

This Section introduces join point types and join point models for architectural styles (Section 3.1) and documents a style-based join point model for the client-server style using two complementary approaches (Section 3.2). The documentation makes the style semantics explicit so that our style-based join point models can be used for architectural composition and analysis. Informal examples illustrate the use of style-based join point types (Section 3.3).

3.1 Style-based join point types and models

A *style-based join point type* categorizes join points of interest under the perspective of some architectural style. For example, some join points of interest for client-server systems are "points at which clients request a service", "points at which servers execute a service", "points at which servers return a result" and "points at which clients receive a result". These join points can be categorized by join point types such as *request-service*, *invoke-service*, *return-result* and *receive-result*, respectively. A set of join point types related to some style characterizes a *style-based join point model*. Table 1 presents join point models for the client-server [10], the pipe-and-filter [32] and the layered [4] styles.

Table 1: Style-specific join point types.

Style	Join point types	Points at which
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
Pipe-filter	read-data	data is read
	write-data	data is written
	end-of-data	end-of-data is signalled
	close	input port is closed
Layered	receive-task	task request is received
	delegate-task	task is delegated to lower layer
	return-result	result is returned by lower layer
	receive-result	result is received by upper layer

Style-based join point models are the basis for the new flavor of architectural composition we have called *style-based composition* or "composition based on style semantics". Such a composition strategy clearly requires documenting join point models and their behaviour, that is, the explicit description of semantics details associated to architectural elements and their style-based interactions.

3.2 Documenting styles and join point models

In this Section we present an extension to a style guide template used to describe component-and-connector (C&C) styles [10] that includes the informal description of join point types (Section 3.2.1), and the textual notation for structural and behavioural descriptions of architectural styles used in style-based compositions (Section 3.2.2). The client-server style is used to illustrate both approaches to documentation. Style guides and descriptions for the pipe-and-filter and layered styles can be found at [6].

3.2.1 A Style Guide Template

Table 2 presents an extended style guide for the client-server style and its join point model (some detailed information that is not relevant in the context of this paper has been omitted). The client-server style is a special kind of call-return style [10] in which the components interact by

Table 2: Extended style guide for client-server style.

Elements	**Component types**: Client (requires services of some other component); Server (provides services to other components) **Connector types**: request-reply synchronous invocation of services from client to server
Relations	Attachments determine which services can be requested by which clients.
Comput. model	Clients request a service from a server, and wait to receive a result for that request. When the server receives a request from a client, it executes a service and may return a result to the client.
Join point types	*request-service* (points at which clients request a service); *invoke-service* (points at which servers execute a service); *return-result* (points at which servers return a result); *receive-result* (points at which clients receive a result).

requesting services of other components. The client-server joint point model allows architects to express naturally auditing requirements on server operations by stating that "after a server `Server` "executes a service" `S` do some auditing on `Server`"; or to express that some data conversion is needed in the client side by stating that "before the client `C` "receives result" `R` do some conversion C1 on `R`". With the proposed extension, information about the join point types, such as the one presented in Table 1, can be added to the style vocabulary and used in specifications of components, connectors, ports, roles and style-based composition.

3.2.2 Structural and Behavioural Specifications

In our approach, AspectualACME [14] is used for structural descriptions and CSP [20] is used for behavioural specifications associated to C&C styles.

Architectural styles are called *families* in AspectualACME. A family is a system type that also defines a design space [30]. A family specification consists of sets of types of components, connectors, properties, and sets of rules that specify how elements of those types may be legally composed. *Properties* can be attached to architectural elements as annotations. In our approach, AspectualACME properties hold formal behavioural descriptions in CSP.

CSP specifications add more semantic details to architectural elements and reveal ordering of interactions among them. By using this formal notation, we make the style semantics explicit so that our style-based joint point models can be communicated and used for composition and analysis. Furthermore, the use of a formal notation allows us not only to think about the correctness of style-based composition with respect to the expected behaviour of the system, but to formally prove such correctness.

Listing 4 presents a specification for the client-server style in AspectualACME. Join point types are naturally formalized by CSP events: *request-service*, *receive-result*, *invoke-service* and *return-result*. Other events such as *request-error* and *return-error* were defined to indicate abnormal situations. The behaviour of port types (`ClientPortT`, `ServerPortT`) and role types (`ClientSideRoleT`, `ServerSideRoleT`) are specified by means of properties – `PortBehaviour` and `RoleBehaviour`, respectively. The behaviour of component types and connector types are described by the *Computation* property and the *Glue* property, respectively. Properties localize CSP specifications where these events are used, and expose the elements' semantics. For instance, the property `PortBehavior` of the `ClientPortT` port type localizes a CSP specification that states that after the port engages in a *request-service* event, it may either wait for a *receive-result* event or for an *receive-error* event (□ is external choice). In both situations, the SKIP process indicates successful termination (no deadlock).

Listing 4: Client-server style in AspectualACME

```
Family ClientAndServerFam = {
 Port Type ClientPortT = {
   Property PortBehaviour =
     ''CLIENT(id) = request_service.id →
           ( receive_result.id?m → SKIP
            □ receive_error.id → SKIP );
            CLIENT(id) '';
 }
 Component Type ClientT = {
   Port sendRequest: ClientPortT;
   Property Computation =
     ''C_COMP(m, id) = CSP_expression'';
 }
 Port Type ServerPortT = {
   Property PortBehaviour =
     ''SERVER(id) = invoke_service.id →
           ( return_result.id!S_COMP(m, id) → SKIP
            ⊓ return_error.id → SKIP );
            SERVER(id) '';
 }
 Component Type ServerT = {
   Port receiveRequest: ServerPortT;
   Property Computation =
     ''S_COMP(m, id) = CSP_expression'';
 }
 Role Type ClientSideRoleT = {
   Property RoleBehavior = '' ... '';
 }
 Role Type ServerSideRoleT = {
   Property RoleBehavior = '' ... '';
 }
 Connector Type CSConnT = {
   Role clientSide: ClientSideRoleT;
   Role serverSide: ServerSideRoleT;
   Property Glue =
     ''CS_GLUE(client, server) = CSP_expression'';
 }
}
```

A process algebra like CSP can be used to describe systems composed by interacting components, which are independent self-contained processes with interfaces used to interact with the environment. The use of such formalisms provide a way to explicitly specify and reason about interaction between different components. Furthermore, phenomena that are exclusive to the concurrent world, that arise from the combination of components and not from the components alone, like deadlock and livelock, can be more easily understood and controlled using such formalisms.

Tool support is another reason for the success of CSP in industrial applications, and consequently, for our choice to use it as the formal notation. For instance, FDR [27] provides an automatic analysis of correctness and of properties like deadlock and divergence. In fact, although out of the scope of this paper, we have used FDR to formalise the architectural styles' behaviour and to perform an automatic analysis of correctness and of properties like deadlock and divergence. We have also checked the valid specification of the architectural composition based on the style semantics. The formal specification of the HW case study presented in this paper using CSP is available at [6].

3.3 Using style-based join point types

Style-based join point types are expected to be used by style-based pointcuts. Tables 3 and 4 present informal examples of style-based pointcuts for the Distribution and the Persistence concerns. These pointcuts select join points of interest based on the semantics of the three styles presented in Table 1 and illustrate the use of style-based join point types. In Section 4 we present language constructs that support the specification of such pointcuts.

Table 3: Style-based pointcuts for Distribution.

Style	Pointcuts	Join Point Types
Client-server	trigger remote communication	request-service, return-result
Pipe-and-filter		write-data
Layered		delegate-task, return-result

The `trigger remote communication` pointcut is in charge of supporting the Distribution concern by selecting join points where a distributed communication takes place. It is associated with the semantics of each style. For the client-server style, this pointcut matches points in which a client requests a service from a (remote) server and the server returns the result to the (remote) client. For the pipe-and-filter style, the remote communication takes place when the filter writes data in a (remote) pipe. Similarly, in the layered style, the remote communication occurs when a layer delegates a task to another (remote) layer or when a layer returns a result to a (remote) layer.

Table 4: Style-based pointcuts for Persistence.

Style	Pointcuts	Join Point Types
Client-server	store data	return-result
Pipe-and-filter		write-data
Layered		return-result

The `store data` pointcut represents the Persistence concern by selecting join points where data must be stored. For the client-server style, the persistence is needed when the results of a service are returned by the server. For the pipe-and-filter style, the `store data` pointcut matches the point where the data are written in a pipe. In the layered style, the pointcut matches points where a result is returned by a layer.

4. STYLE-BASED COMPOSITION

Style-based aspect composition requires pointcuts for selecting style-sensitive join points and mechanisms for composing aspects at the selected join points, according to some strategy. In this Section, we present a pointcut language that quantifies over architectural elements constrained by style-semantics (Section 4.1), and special language constructs to support attachments to the selected elements (Section 4.2).

4.1 A style-based pointcut language

A style-based pointcut language supports the formal specification of style-based pointcuts. An *architectural pointcut designator* is a new kind of element that selects join points that are architecturally relevant for aspect composition, possibly exposing context data.

Primitive pointcuts that are sensible to style-based joint point types (Table 1) are provided for architectural composition. Composite pointcuts can be defined using the operators (`&&`, `||`, `!`). The proposed pointcut language is generic and is able to handle new styles that may appear bringing their own join point types.

The set of pointcut designators supported by the pointcut language are:

- *performs*(*elem*, *eventKind*): selects the join points where some architectural element *elem* performs event *eventKind*.
- *inside*(*elem*, *elementType*) selects the joint points that execute below the execution of an architectural element *elem* that is an instance of *elementType*.
- *SatisfiesType*(*elem*, *elementType*) selects the join points where *elem* is an instance of *elementType*.
- *Pointcut*0 `&&` *Pointcut*1: captures each join point that is picked out by both pointcuts (*Pointcut*0 and *Pointcut*1).
- *Pointcut*0 `||` *Pointcut*1: captures each join point that is picked out by either pointcuts (*Pointcut*0 or *Pointcut*1).
- `!`*Pointcut*: captures each join point that is not picked out by *Pointcut*.

User-defined pointcuts are declared in the `Pointcuts` block, which is defined in architectural configurations. Each pointcut declaration defines and names architectural pointcuts that can be used for composition in attachments and reused in other pointcut declarations.

As an example, consider once again the description given in Listing 2 and the pattern $*Server.*Service$ used to match server elements (properly renamed to follow naming conventions) to be attached to the `Termination` connector. Listing 5 provides a new AD in which servers are selected by means of style-based pointcuts. The pointcut designator `faultyServers` captures $return - error$ join points that are observed in components of type `ServerType`. In this case, pointcut and attachment definitions are not dependent on the names of components or ports but rather on the expected semantics of architectural elements. These pointcut and attachment definitions are robust in face of structural and lexical changes and can be reused in different ADs together with the EH handler.

The selected style-based join points are related to instances of the types prescribed by the style – e.g., components, ports, connectors, roles and systems (or configurations). A pointcut that can be used in the attachments block (e.g. `faultyServers`) must comprise (or be filtered to comprise) a set of component ports since it will be attached to an aspectual connector role.

4.2 Aspect composition

A conventional attachment in AspectualACME defines a relationship between a connector and a component by associating a port interface on a component with a role interface on a connector, using the operator `TO`. AspectualACME also

Listing 5: Aspect composition based on style semantics

```
System HW: ClientServerFam = {
 Component Application = { ... }
 ...
 Component ExceptionHandling = { ... }
 Connector T1: Termination;

 Pointcuts {
    faultyServers(s) =
        SatisfiesType(s, ServerType) &&
        performs(s, return-error);
 }
 Attachments {
   T1.terminationBasedRecovery to
     ExceptionHandling.communicationFaultHandler;
   T1.faultyService to* faultyServers;
   ...
 }
}
```

supports **-attachments* that associate one aspectual connector' role interface with several ports using the new operator `TO*`. The `Attachments` block localizes the definition of conventional attachments as well as *-attachments.

The `faultyServers` pointcut defined in the `Pointcuts` block (Listing 5) is used by an *-attachment, defined in the `Attachments` block. The `faultyServers` pointcut evaluates to { all sets of ports on components of type `ServerType` that perform the join point type (or event) return-error }. The meaning of an attachment such as `T1.faultyService to* faultyServers;` is: "bind each port in the set `faultyServers` to the connector role `T1.faultyService`".

5. EVALUATION

This section presents a systematic evaluation on the usefulness of style-based join point models and the proposed pointcut language. Section 5.1 presents assessment procedures used in the evaluation. To facilitate the discussion of the results, we discuss first one example of EH connector in the HW system (Section 5.2) and a representative set of pointcuts (Section 5.3) reused in all the three analyzed architectures. We present the evaluation outcomes according to three different perspectives: (i) pointcut expressiveness and intra-project reuse (Section 5.4), (ii) pointcut stability in the presence of heterogeneous types of changes (Section 5.5), and (iii) stability of pointcut definitions in composite architectures (Section 5.6).

5.1 Assessment procedures

Three case studies have been used: the HW system (Section 2), a Conference Management (CM) system [29], and a product line for data manipulation in mobile devices, called MobileMedia (MM) [12]. HW and MM are medium-sized systems (around 10K lines of code each). The complete architecture is composed of approximately 40 components each, from which 4 are EH aspects. Whereas these applications are from different domains and entail significantly distinct architectures, they all realise the client-server and layers styles and implement their own EH strategies. These similarities enabled us to assess the reuse degree of style-based pointcuts in realistic hybrid architectural designs.

The evaluation was carried out in a stepwise fashion and focused on the stability and reusability analysis. We have compared the existing syntax-based pointcuts with their style-based equivalents defined for all the three target applications. While translating syntax-based pointcuts to style-based pointcuts, there were several cases were one-to-one mapping did not hold. For instance, some syntax-based pointcuts need not to be defined due to the increased reuse (superior quantification) obtained with style-based pointcuts. This will be discussed in Section 5.4.

The first assessment step primarily analyzed *the extent to which the stable description of these two pointcut categories was sustained through the ten releases of the HW architecture*. The investigation included a multitude of pointcuts for transaction management, EH, persistence and distribution. Pointcut additions and modifications were quantified in order to enable us to evaluate the stability and reusability of the composition descriptions. Considering all the case studies, we have analyzed 23 changes to both systems, ranging from localized component refactorings to widely-scoped architecture increments or modifications.

The second step assessed the *inter-project reusability of style-based EH pointcuts*, which were common to all the three analyzed systems. We have also quantified the stability of such pointcuts through the releases of the CM and MM systems.

5.2 Exception connectors

The HW system encompasses a number of connectors to modularize three architectural crosscutting concerns: exception handling, transaction management, and distribution (Section 2.1). In the following, we specifically concentrate on the discussion of connectors and alternative style-based pointcuts employed to EH policies in the HW architecture; some of them were also applied to the CM and MM architectures. Our choice for EH is driven by the fact that its crosscutting nature is style-agnostic: global exception handlers tend to affect several system components independently of the choice of styles and their compositions [4]. This allows us to evaluate the applicability of style-based pointcuts based on different architecture decompositions and hybrid software architectures. The other reasons are that: (i) EH has been extensively referred in the literature as a classical crosscutting concern [22, 33], and (ii) the benefits and drawbacks of aspectizing EH using AO programming techniques have been well explored nowadays [13, 33].

In Listing 1 we presented the definition of the main architectural elements to address concerns specific to EH in the HW system. Three connectors were used to capture the crosscutting nature in which the abnormal interaction protocols are employed in the HW architecture decomposition [33, 14]: (i) `Termination`, (ii) `Retrial`, and (iii) `Propagation`.

Listing 6 presents the CSP specification of an aspectual connector that deals with the `Retrial` semantics. It specifies a protocol in which a service is retried after service invocations. The connector is used to re-invoke the faulty service after the handler execution and before returning the result to the caller. Style-based pointcuts are defined in the HW architecture description to attach architectural elements to aspectual connectors that modularize the abnormal collaboration protocols. The retrial protocol is also used in the MM architecture, while both termination and propagation policies are applied to all the three case studies.

Listing 6: Retrial connector in CSP.

```
Connector Type Retrial = {
  Role faultyService =
         invoke-service →
         return-result!x → faultyService
  Role localStateRecovery =
         invoke-service →
         return-result!x → localStateRecovery
  Glue = faultyService.invoke-service →
         localStateRecovery.invoke-service →
         localStateRecovery.return-result →
         faultyService.return-result → Glue
}
```

Listing 7: Exception handling pointcuts

```
System HW: ClientServerFam = {
  Component ...
  Connector Type Termination = { ... }
  Connector Type Retrial = { ... }
  Pointcuts {
   faultyServers(s) = ...;
   faultyLayers(l) =
        SatisfiesType(l, LayerType) &&
        performs(l, delegate-task-error);
   faultyInnerLayers(l) =
        faultyLayers(l) &&
        inside(l, ServerType);
   faultyStrictInnerLayers(l) =
        faultyInnerLayers(l) &&
        ! SatisfiesType(l, ClientType)
   faultyAcessorFilters(c) =
        SatisfiesType(c, AccessorType) &&
        SatisfiesType(c, FilterType) &&
        performs(c, data-access-error);
  }
  Attachments { ... }
}
```

5.3 Pointcuts for exception handling

Listing 7 provides a list of style-based pointcuts for exception handling. These pointcuts are used to attach the affected architectural elements to the aspectual connectors involved in EH. For example, pointcut `faultyServers` (Listing 5) exploits the join point model defined by the client-server style in order to select 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. The `Retrial` connector (Listing 6) 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)* retry the service execution again before they are propagated as exceptional results to the clients. This systemic exception handling strategy crosscuts all the `return-result` events from the three server instances defined in the HW architecture. This is also hold the CMS and MM architectures.

On the other hand, we cannot specify a composition in HW architecture 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-result` events immediately 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 the relevant information associated with server faults before they are propagated as exceptional results to the clients. Not all the information details are propagated to the clients. This generic EH strategy crosscuts all the `return-result` events from the three server instances defined in the HW and CM architectures.

It is important to highlight that certain EH strategies required the refinement of style-based pointcuts with some lexical matchings. The lexical matchings were mostly required in cases where exception handlers were non-generic and/or local (i.e. exceptions masked in the specific component interfaces without the need of propagating them to the callers).

5.4 Expressiveness and intra-project reuse

The first style-based pointcut in Listing 7 is an alternative solution for the syntax-based pointcuts presented in Listings 2 and 3. Those original pointcuts respectively used explicit references to syntactical elements and wildcards with naming convention in the HW description, rather than the semantics of the affected elements by EH. As a result, they suffer from a core problem: they do not capture the rationale behind the composition. As discussed above, the crosscutting nature of the attachment is considerably dependent on the client-server style semantics: we want to attach the retrial connectors to all the faulty servers before they are propagated as results to the clients. In order to understand the wildcard-based pointcut (Listing 3), the architect needs to examine the name of all the components and ports in the HW description.

Independently from the specific nature of the involved components, the client-server style instances in the three case studies exhibited several points where failures could recurrently occur. These recurring failure points included client requests at the client side, client requests at the server side, server replies at the server side and server replies at the client side. It was common the case where architects attached similar EH strategies to those points in a modular fashion. The style-based pointcuts for client-server EH (Section 5.3), have been proved to be quite generic and were reused across the multiple client-server instances in the HW architecture description. This was one of the main reasons on why fewer additions and modifications (Figure 2(a)) were required to style-based EH pointcuts than to their syntax-based counterparts for the HW architecture releases. An analysis of the data presented in Figure 2(b) revealed that a similar phenomenon happened with the pointcuts for other HW aspects (Figure 2(b)).

We have observed that the higher number of syntax-based pointcut modifications was directly correlated with the difficulty of reusing them within the same AD. They have explicit references and dependencies to component and port names of the HW design (Listings 2 and 3). When trying to reuse wildcard-based pointcuts (Listing 3) through HW releases, we needed to rename components and ports of the target architecture. Even so, it was not always the case that we could change the names of existing components and ports because (i) they could be off-the-shelf components, and (ii) the renaming would cause ripple effects in ADs since the existing attachments need to be modified accordingly as well.

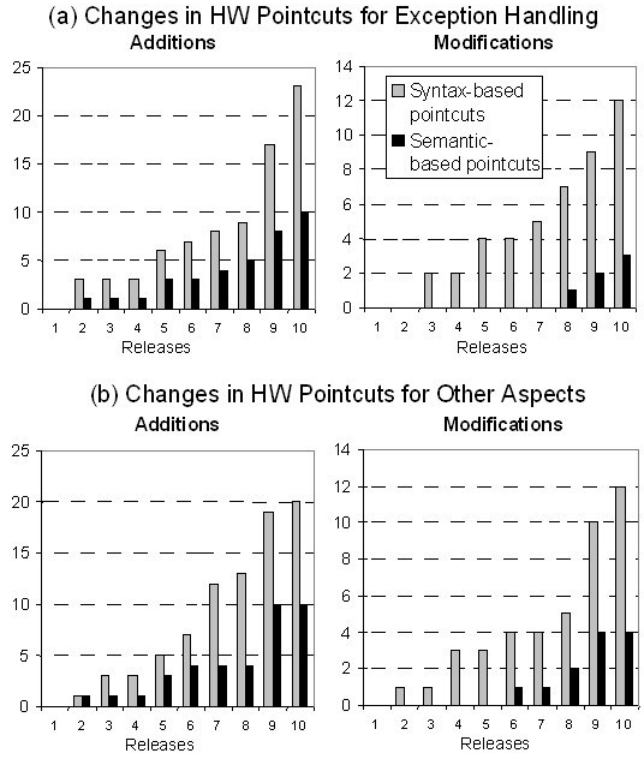

Figure 2: Pointcut stability in HW architecture.

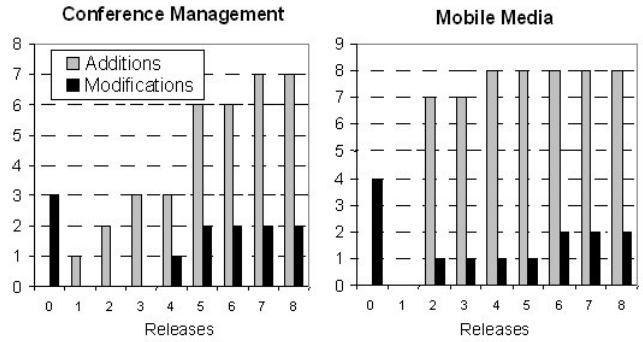

Figure 3: Reuse effort for style-based EH pointcuts.

5.5 Pointcut stability and inter-project reuse

Figures 2(a) and 2(b) show that style-based pointcuts were consistently more stable through the HW architecture releases. We have noticed that the stability of style-based pointcuts was also superior in releases where major architectural changes were implemented, such as versions 5, 9, and 10. For instance, in release 5, the HW system was supporting two possible architecture configurations [33]: in one of them, the HW system is used via an HTML and Javascript user interface, which interacts with Java servlets running in the Web server. In the other configuration, a Java user interface interacts directly with an `Application` server using Java RMI. In other words, the Web server is optional.

However, the style-based pointcuts for EH discussed before did not need to be changed to accommodate both HW configurations. They are still valid and stable even when the number of servers is a variation point. On the other hand, some syntax-based pointcuts, such as the one in Listing 2, were modified according to the (de-)activation of the Web server. Explicit references to the Web server needed to be removed. A similar problem occurred with the syntax-based persistence pointcuts when the optional use of an object-oriented database was introduced in the 6th release (Figure 2(a)). This variation had an impact on the port names of the `PersistenceMechanism` layer.

Thus, the use of syntactical aspect compositions often exacerbated the problem of pointcut fragility in ADs. Because the compositions were specified with an explicit reference to the name of components or ports, it created a tight coupling between aspects and the affected HW, MM, and CM elements. These strong syntactic dependencies harmed their architecture maintainability. An analysis of MobileMedia and Conference Management architectures shows that the lexical matchings in EH pointcuts were also fragile and not resilient to changes. In fact, several observed architectural-level ripple effects were related to the need of either changing names of existing architecture elements or including new components or interfaces realizing the style-specific types. Mainly in the former case, there was a considerable incidence of syntax-based pointcut declarations being undesirably broken. All the poincuts needed to be carefully revisited and changed also in the MM and CM architecture descriptions.

We have also investigated how easy was to reuse style-based pointcuts for EH (Section 5.3) across multiple software ADs. Some style-based pointcuts naturally did not make sense to reuse in the target applications. No syntax-based pointcut could be reused from a project to another due to their dependence on architecture element names. Figure 3 presents the number of additions and modifications that were required to reuse the style-based EH pointcuts in the MM and CM applications (release 0). From the 10 EH pointcuts being reused, three modifications were required for the CM description and four for the MM description. Certain pointcut modifications needed to be implemented in order to adapt them to specific style combinations (Section 5.6) in the MobileMedia and Conference Management designs. For instance, the fourth pointcut in Listing 7 is specialized for layered architectures with shared data, which was not instantiated in the MM and CM architectures; the later is a layered architecture centered on the blackboard style for data management and communication. The next section discusses other lessons learned in terms of defining style-sensitive pointcuts in the presence of style compositions.

5.6 Style composition and pointcuts

As discussed before, styles can be composed with each other in several ways. For instance, the HW architecture involves the combination of several kinds of styles which directly determines the set of architectural join points available. Overall, our three case studies exhibited three types of style composition: *(i)* hierarchical composition (Section 5.6.1), *(ii)* overlapping (Section 5.6.2), and *(iii)* conjunction (Section 5.6.3). In the following subsections, we briefly discuss the interplay of such style composition categories and style-based pointcuts in the context of the HW case study.

5.6.1 Hierarchical composition

In the investigated architectural designs, distinct styles were used at different levels of the 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. In AspectualACME, it is supported by *representations* that elaborate a parent element (component, connector, etc.) and support the description of encapsulation boundaries, as well as multiple refinement levels. 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 provided by the representation of the `Application` component insulates the join points of that particular instance of the Layered style from the rest of the architecture, which is realizing other architectural styles. Hence, the set of style-based join points in the internals of the `Application` component is determined by the semantics of the Layered style, in addition to the particular names of its inner interfaces, connectors, and components.

One of the EH pointcuts in the HW architecture selects all the exceptions in *delegated tasks* to layers inside the `Application` server, described by the `faultyInnerLayers` pointcut (Listing 7). This architectural pointcut selects only join points that are exceptional events in components 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` of the pointcut language to capture specific join points involved in hierarchical architecture composition (Listing 7).

5.6.2 Style overlapping

The most recurring use of multiple styles in HW, MM, and CM architectures was the creation of architectural elements that satisfy multiple types, each type taken from a different style. Such architectural elements form an overlapping zone of styles, and they embody vocabulary and satisfy the constraints of all the styles used. 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 stylistic compositions in the HW architecture was exploited to express certain useful style-based pointcuts for error-handling configurations. 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, described by the `faultyStrictInnerLayers` pointcut (Listing 7). In other words, it includes any exception raised by tasks executed by the `PersistenceMechanism` and `DataManagement` layers, but excludes the `Business` layer given the fact that 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 (Section 2.2).

5.6.3 Style conjunction

Another style composition category found in HW, CM and MM architectural descriptions is conjunction, which results in hybrid styles [28]. A system can declare that it satisfies multiples styles, by means of the union of their design vocabularies, and conjoining their constraints. In AspectualACME, when more than one style is used, the new system must be an instance 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.

One example of stylistic conjunction is the symbiosis involving the pipe-and-filter and repository styles in the HW design, which encompasses the addition of the components defined by the shared data style – i.e., accessors and databases – to a pipe-filter system. Some internal components 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. The `faultyAcessorFilters` pointcut (Listing 7) captures exceptions returned as result to *data accesses* to components in which their types are hybrid, i.e., conjunctions of *accessors* and *filters*. 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 `staleConnectionHandler` service (Section 2.2).

6. RELATED WORK

In previous work [29], we presented an exploratory study of the influence exerted by style-based composition on the stability of architectural modules addressing error handling and security, in the context of alternative designs for the CM system. Each design adopted a different style (blackboard, reflective blackboard, reactive coordination and stigmergic coordination). In this work, stability and reusability assessments of the style-based composition model were carried out by means of three case studies involving different architectural styles and different application domains.

Style-based join points and existing ADLs. Existing AO approaches at the architecture level do not support style-based pointcut specifications. The exploitation of style-based join point models seems to address at the architectural level the recurring problem of pointcut fragility, a common concern in conventional programming-level join point models. As discussed in Section 5, in previous empirical studies on architecture stability [25, 19] using the HW system, we observed that style-based pointcuts tended to be more stable in the presence of widely-scoped design changes because the syntactical changes in ADs occur more often than modifications on the stylistic architectural choices [25]. Certainly, AO ADLs may resort to conventional pointcut designators in conjunction with style-based designators.

Semantics-based composition in requirements and design. The drawbacks of syntax-based AO compositions have already been discussed [8] from the perspective of AO requirements engineering. The similarities with our work are the discussions about the disadvantages of syntax-based compositions in contrast with the expressiveness of semantics-based composition in early aspects. However, the main dif-

ferences are that of focus (ours is on ADLs) and the use of a formal language (CSP) versus natural language to express join point models and compositions.

Klein *et al.* [23] argue in favor of a semantics-based approach for pointcut languages at a more abstract level. They propose a semantics-based weaving algorithm for hierarchical message sequence charts, a scenario formalism, and assess its benefits and limitations. In our work, we propose a semantics-based approach for pointcut languages at the architectural level. We resort to style-based join point models and a pointcut language to support style-based composition in ADs and provide initial experimental assessment on the stability and reuse of such ADs.

Stein *et al.* [34] point out that most pointcut languages at the modeling level are based on existing AOP languages and that there are no means to express join point selection criteria with respect to the behavior of a system. They provide graphical visualization of join point selections via Join Point Designation Diagrams (JPDDs), where each visualization can be examined with respect to the selection's underlying conceptual model (control flow-oriented, data flow-oriented, or state-oriented conceptual model). In our work, we have also defined an early aspects approach that is independent of existing AOP languages. However, instead of focusing on visualization diagrams and conventional join point models, we propose style-based join point models and provide a textual pointcut language to express the selection of join points based on the semantics of architectural styles.

Application-specific models and pointcuts. Kellens *et al.* [21] propose model-based pointcuts, defined in terms of a conceptual model of the base program, and show how such pointcuts can be less fragile with respect to changes in the base program. Brichau *et al.* [3] extend the model-based pointcuts technique and use a logic-based pointcut language to express application-defined pointcut predicates. In this point their work is similar to ours as it exploits the expressive power of a logic language to declare pointcuts and to address the pointcut fragility problem. However, instead of addressing application-specific join point models, our proposal focuses on and argues in favor of more high-level, style-specific join point models and enforces style semantics on every application that adheres to those architectural styles.

7. CONCLUSIONS

The definition of pointcut languages plays a central role in aspect-oriented approaches [23, 21]. Although it has been known that the abstraction level of pointcuts must be lifted [21, 34], syntax-based pointcuts are typically used to select join points based on the names of architectural elements, exposing architectural description to pointcut fragility [21, 19] and reusability problems.

We propose style-based join point models and provide a pointcut language that supports the selection of join points based on the semantics of architectural styles. Style-based join point models and the composition approach proposed here are meant to be language-independent and adaptable to different ADLs and formalisms to define style-specific behaviors. In this paper, architectural descriptions are expressed in AspectualACME and formal behavioural descriptions in CSP. We plan to exploit the proposed approach with other languages and formalisms in future work.

The definition and the rationale behind style-based join point models emerged from our analysis of existing AO ADLs [2, 14] and our experience on designing and assessing AO architectures for a number of distinct application domains. AO applications include a reflective middleware system [5], multi-agent systems [16], a CVS system [13], product lines for quality control measurements [24] and J2ME games [24].

In this paper, our main claim is that style-sensitive composition promotes the development of architectural aspects that are more reusable and stable in face of syntactic changes to the architecture. Our systematic comparison of style-based composition against syntax-based composition in three case studies has provided initial evidences that support our claim. We also discussed briefly the interplay of some techniques for style composition and style-based pointcuts in the context of our exploratory studies; we certainly need to gather more empirical evidence that may allow us to characterize other benefits and liabilities related to these issues. Future experimental work is also needed to assess the adequacy of style-based pointcuts to express architect's design intent and composition rationale, and the challenges of their adoption by the Software Architecture community.

8. ACKNOWLEDGMENTS

This work is partially supported by grant 486125/2007-6: Brazilian Council for Scientific and Technological Development(CNPq), and grant 219/2008: Fundação de Amparo à Pesquisa do Estado da Bahia (Fapesb). Thais Batista thanks the Brazilian Petroleum Agency (ANP)/PRH22 for the partial financial support for her research.

9. REFERENCES

[1] E. Baniassad et al. Discovering early aspects. *IEEE Software*, 23(1):61–70, 2006.

[2] T. Batista et al. Reflections on Architectural Connection: Seven Issues on Aspects and ADLs. In *Early Aspects'06 at ICSE*, pages 3–9, Shangai, China, May 2006.

[3] J. Brichau, A. Kellens, K. Gybels, K. Mens, R. Hirschfeld, and T. D'Hondt. Application-specific models and pointcuts using a logic metalanguage. *Comput. Lang. Syst. Struct.*, 34(2-3):66–82, 2008.

[4] F. Buschmann et al. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[5] N. Cacho et al. Composing design patterns: a scalability study of aspect-oriented programming. In *AOSD '06: Proc. 5th Intl Conf. on Aspect-oriented software development*, pages 109–121, New York, NY, USA, 2006. ACM Press.

[6] C. Chavez et al. Composing architectural aspects based on style semantics. http://www.dcc.ufba.br/∼flach/JPM/, 2008.

[7] R. Chitchyan et al. Survey of aspect-oriented analysis and design approaches. Technical Report AOSD-Europe-ULANC-9, AOSD-Europe, May 2005.

[8] R. Chitchyan et al. Semantics-based composition for aspect-oriented requirements engineering. In *AOSD'07: Proc. 6th Intl Conf. on Aspect-oriented software development*, pages 36–48, New York, NY, USA, 2007. ACM Press.

[9] P. Clements et al. Aspects in architectural description: Report on a first workshop at aosd 2007. *SIGSOFT Softw. Eng. Notes*, 32(4):33–35, 2007.

[10] P. Clements et al. *Documenting Software Architectures - Views and Beyond*. Addison-Wesley, 2007.

[11] C. Cuesta et al. Architectural aspects of architectural aspects. In *Proc. of European Workshop on Software Architecture (EWSA 2005)*, pages 247–262, 2005.

[12] E. Figueiredo et al. Evolving software product lines with aspects: An empirical study on design stability. In *30th Intl Conf. on Software Engineering (ICSE 2008)*, pages 261–270, Leipzig, Germany, 2008.

[13] F. Filho et al. Exceptions and Aspects: The Devil is in the Details. In *Proc. of FSE-14, 14th Intl Conf. on Foundations on Software Engineering*, pages 152–162, 2006.

[14] 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 *Proc. of European Workshop on Software Architecture (EWSA 2006)*, Nantes, France, Sept 2006.

[15] A. Garcia et al. 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 et al. Separation of Concerns in Multi-agent Systems: An Empirical Study. In *Software Engineering for Multi-Agent Systems II, Research Issues and Practical Applications (SELMAS 2003)*, pages 49–72, 2003.

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

[18] P. Greenwood et al. On the contributions of an end-to-end aosd testbed. In *Early Aspects'07 at ICSE*, Washington, DC, USA, 2007. IEEE Computer Society.

[19] P. Greenwood et al. On the impact of aspectual decompositions on design stability: An empirical study. In *LNCS 4609, Proc. 21st European Conf. on Object-Oriented Programming (ECOOP)*, pages 176–200. Springer, 2007.

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

[21] A. Kellens et al. Managing the evolution of aspect-oriented software with model-based pointcuts. In *LNCS 4067, Proc. 20th European Conf. on Object-Oriented Programming (ECOOP)*, pages 501–525. Springer, 2006.

[22] G. Kiczales et al. An Overview of AspectJ. In *Proc. 15th European Conf. on Object-Oriented Programming (ECOOP)*, pages 327–355. Springer, 2001.

[23] J. Klein et al. Semantic-based weaving of scenarios. In *AOSD'06: Proc. 5th Intl Conf. on Aspect-oriented software development*, pages 27–38, New York, NY, USA, 2006.

[24] U. Kulesza et al. Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming. In *Proc. 9th Intl Conf. on Software Reuse (ICSR'06)*, Turin, Italy, June 2006.

[25] U. Kulesza et al. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In *Proc. of 9th Intl Conf. on Software Maintenance - ICSM'06*, 2006.

[26] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proc. 22nd Intl Conf. on Software Engineering (ICSE 2000)*, pages 418–427, New York, NY, USA, 2000. ACM Press.

[27] F. S. Ltd. Fdr: User manual and tutorials, version 2.82, 2005.

[28] N. R. Mehta and N. Medvidovic. Composing architectural styles from architectural primitives. In *Proc. 9th ESEC / 11th ACM SIGSOFT FSE (ESEC/FSE-11)*, pages 347–350, New York, NY, USA, 2003. ACM Press.

[29] A. Molesini, A. Garcia, C. Chavez and T. Batista. On the quantitative analysis of architecture stability in aspectual decompositions. In *Proc. 7th Working IEEE/IFIP Conf. on Software Architecture (WICSA 2008)*, pages 29–38, Vancouver, BC, Canada, 2008.

[30] R. Monroe. Capturing software architecture design expertise with armani. Technical Report CMU-CS-98-163, Carnegie Mellon Univ. School of Computer Science, January 2001. Version 2.3.

[31] 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.

[32] M. Shaw. Beyond objects: a software design paradigm based on process control. *SIGSOFT Softw. Eng. Notes*, 20(1):27–38, 1995.

[33] S. Soares et al. Implementing distribution and persistence aspects with aspectj. In *Proc. of OOPSLA 2002*, pages 174–190, 2002.

[34] D. Stein et al. Expressing different conceptual models of join point selections in aspect-oriented design. In *AOSD'06: Proc. 5th Intl Conf. on Aspect-oriented software development*, pages 15–26, New York, NY, USA, 2006. ACM.