Multi-View Refinement of AO-Connectors in Distributed Software Systems

Steven Op de beeck, Marko van Dooren, Bert Lagaisse, Wouter Joosen
IBBT-Distrinet, KU Leuven, 3001 Leuven, Belgium.
{steven.opdebeeck, marko.vandooren, bert.lagaisse, wouter.joosen}@cs.kuleuven.be

Abstract

This paper presents MView, a technique that enables the separation of various developer stakeholder views on an architectural connector in distributed software systems.

While state-of-the-art AO-ADLs focus on describing compositions using aspect-based connectors, there is no support for describing a connector across multiple architectural views. This is, however, essential for distributed systems, where run-time and distribution characteristics are not represented in a single view. The result is connectors that suffer from monolithic descriptions, where the views of different stakeholders are tangled.

MView untangles these stakeholder views by defining them in separate modules and specifying refinement relations between these modules. We have integrated MView in a prototypical ADL, which allows code generation for multiple AO-middleware platforms.

We evaluate MView in terms of stakeholder effort in a content distribution system for e-Media. We have created an Eclipse-plugin that supports the ADL, and performs code generation to the JBoss and Spring middleware platforms.

Categories and Subject Descriptors C.2.4 [*Distributed Systems*]: Distributed Applications; C.2.11 [*Software Engineering*]: Software Architectures

1. Introduction

Distributed software systems consist of *complex compositions* of components and third-party subsystems. This complexity is inherent to the growing need to take into account the run-time and distribution characteristics of compositions, as well as their crosscutting nature. This is a trend that is gaining support from AO-Middleware platforms (AOM) such as ReflexD, DyMAC, and AWED [21, 23, 36], which offer direct support for such complex compositions.

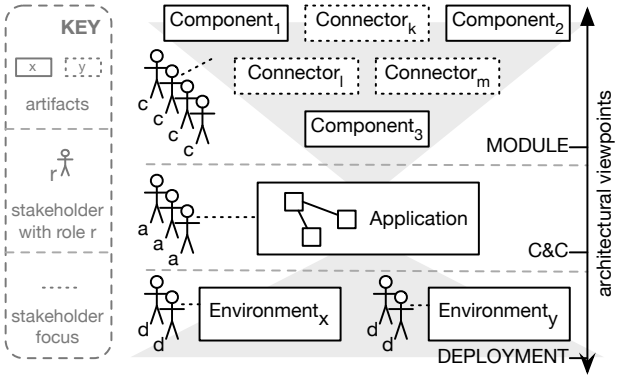

Figure 1. The development process in terms of stakeholder roles and architectural viewpoints.

In software architecture, a *view* captures the concerns of a *stakeholder* on the basis of a *viewpoint*, while a viewpoint defines the language for describing a specific facet of the system [20]. Distributed software systems are described based on the views of multiple developer stakeholders with varying expertise and development roles: e.g. component developers, application assemblers, and deployers. Their views are specified in accordance with at least the architectural viewpoints of module, component-and-connector and deployment, respectively [9]. Figure 1 outlines how stakeholder roles match viewpoints in the development process.

To describe components and their compositions, several Architectural Description Languages [6, 14, 17] (ADL) have been proposed. More recently, various AO-ADLs [15, 16, 24, 26, 28, 31–34] have been defined that capture the crosscutting compositions of software systems in AO-connectors.

The goal of these AO-ADLs differs from ours: they aim to separate the crosscutting concerns of stakeholders with role c (see Figure 1) by means of AO-connectors k, l, and m. However, they do not separate the various views of stakeholders c, a, and d, involved in such an AO-connector throughout the development process. Here, c, a, and d stand for *component developer*, *assembler*, and *deployer*.

AOSD'12, March 25–30, 2012, Potsdam, Germany.

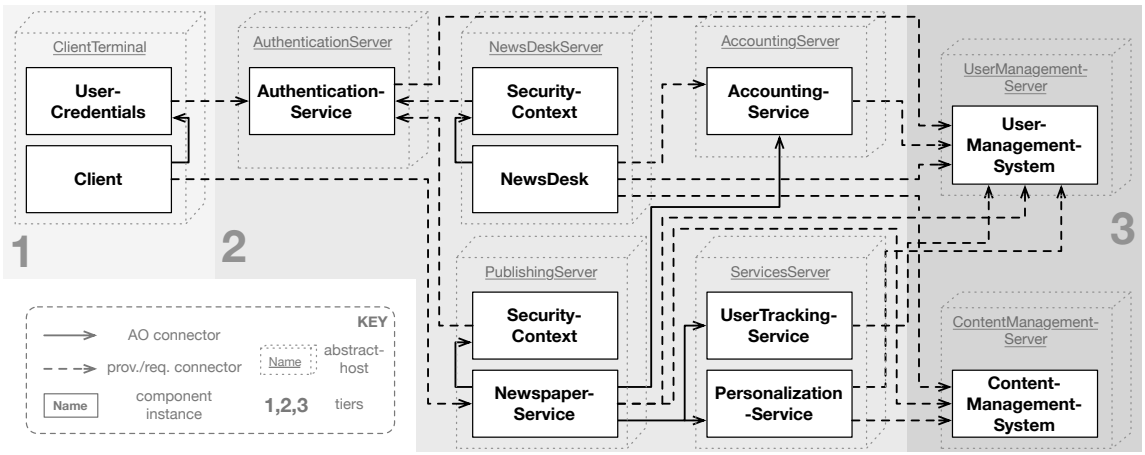

Figure 2. A component-and-connector view on the publishing architecture.

Consider for example the composition of the *accounting* and *news delivery* services in a content distribution system for e-Media. Informally, the composition is described as follows:

> "Call the *chargeForService* method, whenever these conditions hold: the *execution* of *method* fetchArticle of the NewsRemote *interface*, and on the NewspaperService *component instance*, and located on the PublishingServer *host*."

This complex composition suffers from a monolithic description, where the views of different stakeholders are tangled. We discern the following expert stakeholders: a component developer (*execution of a method of an interface*), an application assembler (*of a component instance*), a deployer (*located on a remote host*). Since the views of these stakeholders are not all available or even relevant at one time during architecture, a single artifact should not capture this entire composition. Furthermore, because of low-level details like *host* conditions, adapting the composition for use in another deployment environment is error-prone.

We propose MView to achieve the desired modularity of stakeholder views in the development process. MView is a technique for *multi-view refinement* of connector descriptions, that builds on inheritance and step-wise refinement. In MView, a complex composition is modularized across multiple connectors, where each one deals with the composition in the context of the view of a *single* stakeholder. For example, a module view adds module related constraints: *method* M *on interface* I. MView enables one connector view to refine another connector view. So, as the architecture is further developed, a connector can be refined by an additional view, for example by a deployment view that adds deployment related constraints: *must be running on host* R. As a result, the complex composition is specified through a process of multi-view refinement of connectors. Each connector view is constructed separately, by the right stakeholder for the job, at the appropriate time during development.

We have integrated MView in a prototypical ADL, called MViewADL, that we support with an Eclipse plugin [2]. This plugin supports code generation to multiple distributed middleware platforms, JBoss [1] and Spring [3].

We have applied MView in a case study on e-Media [38] where we compare its stakeholder effort with other techniques. The case study has a detailed architecture that is based on industry requirements and it serves as the running example throughout this paper.

The remainder of the paper is structured as follows. Section 2 introduces the e-Media case study, followed by an illustration of the problem. In Section 3, we analyze the requirements for MView refinement in ADLs and we discuss the support in existing approaches. Section 4 introduces and illustrates the MViewADL, followed by a detailed explanation of MView refinement. Section 5 discusses automatic code generation to JBoss. Section 6 evaluates MView in terms of stakeholder effort, and revisits the requirements for MView. Section 7 presents the related work, and Section 8 concludes.

2. Case Study and Problem Illustration

In this section, we define the case study that we use throughout the paper, followed by a detailed problem illustration.

The subject of our case study is a content distribution system for e-Media. A digital newspaper offers news in different media formats and sizes, to be delivered through different communication channels. It supports various additional services, like flexible accounting, tracking user-interest and content personalization. A consumer can browse recent headlines and read article summaries for free. However, to have access to the full content and additional services, a consumer is required to sign up. The system charges a signed-up consumer for paid services by means of micro-payments. User-tracking, personalization, authorization and accounting are examples of services that are integrated using complex composition.

This case study was developed throughout a number of research projects in the e-Media space, in close collaboration with actual industrial news publishers. The architecture and implementation were performed for an AOSD Industry Demonstrator [38].

Figure 2 shows a component-and-connector view of a relevant subset of the publishing software architecture, consisting of component instances, their compositions and abstract host allocation. The architecture is deployed along three tiers: (1) Client, (2) Business and (3) Storage.

The Business tier consists of a number of subsystems: the Newspaper, NewsDesk and Auxiliary services. The NEWSPAPERSERVICE component is externally accessible by the CLIENT. It supplies the services to browse and read articles.

Authentication and Authorization are supplied by the AUTHENTICATIONSERVICE, USERCREDENTIALS and SECURITYCONTEXT components. The additional services USERTRACKINGSERVICE and PERSONALIZATIONSERVICE can be activated on a per-user basis. User tracking keeps track of the reading behavior of the consumer, while personalization uses this to personalize the view of the consumer on the news.

The ACCOUNTINGSERVICE component is responsible for doing the accounting of *Service Usage* behavior of the consumer. The methods that belong to this *Service Usage* category are fetchArticle, listNewestArticles, listArticlesForTag, and listArticlesForCategory —a part of the main interface of the NEWSPAPERSERVICE. The complex composition between the NEWSPAPERSERVICE and ACCOUNTINGSERVICE components is modeled, in our component-and-connector view, by means of a full line arrow. The accounting composition is tangled by the following stakeholder views: Accounting must only interact with (a) the specific *service usage* methods of (b) the NEWSREMOTE interface, implemented by any component instance that is (c) deployed on the PUBLISHINGSERVER abstract host.

Problem Illustration. We illustrate the problem of tangling further with an example in an existing ADL: Fractal-ADL [32]. Figure 3 shows a complex composition description that realizes *Accounting*.

The example shows two component declarations, ACCOUNTINGSERVICE and NEWSPAPERSERVICE with their provided interfaces (*role="server"*). The part of the composition we are interested in, is the *weave* XML element (line 12–18). The *root* attribute indicates the application scope of the aspect, and the *acName* attribute indicates the component that supplies the additional behavior (advice). The *pointcutExp* attribute contains the *conditions* for composition, and must conform to the following template:

"(client | server | both) component; interface; method:type"

Client and *Server* indicate an expected outgoing or incoming call respectively, followed by a *component* name, *interface* name, and *method* name, and return *type*.

```
1  <definition name="NewspaperApp">
2    <component name="AccountingService">
3      <interface name="AccountingRemote" role="server"
4          signature="org.objectweb.....AspectComponent"/>
5    </component>
6    <component> name="NewspaperService">
7      <interface name="NewsRemote" role="server"
8          signature="Service"/>
9    </component>
10   <binding client="NewspaperService.NewsRemote"
11      server="AccountingService.AccountingRemote"/>
12   <weave root="this" acName="AccountingService"
13     aDomain="ServiceAccounting"
14     pointcutExp="
15 <!-- PublishingServer; (unsupported informal description) -->
16       SERVER *;
17       NewsRemote;
18       fetchArticle(ContentItemId):ContentItem" />
19 </definition>
```

Figure 3. The complex SERVICEACCOUNTING weave combines contribution from three stakeholders into a single description.

This *pointcutExp* representation is tangled with the views of multiple stakeholders, as indicated by the different gray areas. The first one from the bottom (line 17–18) is a module-viewpoint description that focusses on interfaces and their methods. It describes a pointcut that matches calls of the fetchArticle method, of the NEWSREMOTE interface. This is the responsibility of a module developer familiar with the newspaper service and the *Accounting* requirements. The next view (line 16) further specifies the pointcut in terms of the Component-and-connector viewpoint: only incoming calls are allowed for any (indicated with *) component instance. This is the job of an application assembler. Finally, the called component is required to be allocated on the *PublishingServer* abstract host (line 15). This is the job of an application deployer. This last condition, however, is not supported in Fractal-ADL. We had no choice other than to specify it informally in the architectural description.

This example shows how the views of different stakeholders w.r.t. a single composition are tangled in a monolithic connector description. Furthermore, it shows that the rigid structure of this particular description is difficult to refine across multiple connectors.

3. Requirements Analysis

In this section, we present four requirements for ADL features that are necessary in order to support multi-view refinement. We apply these requirements in a study of the related work.

3.1 Requirements

It is our goal to enable the stakeholders, that are involved with a composition, to specify their views on that composition separately and at the appropriate time during development—*the right stakeholder at the right time*. This requires the monolithic connector to be split up into multiple explicitly related connectors, that are no longer tangled with respect to the concerns of the involved stakeholders.

We break this goal down into four requirements for ADL features that we deem necessary. We use these requirements in a study of the related work to determine for each approach what is missing in order to achieve our goal. These requirements are, in part, based on the Classification and Comparison Framework for ADLs by Medvidovic, et al. [22], but applied in the context of distributed middleware systems.

1. ***Open Connector.*** First, the specification of a connector is defined as the high-level model of its composition behavior. In order to consistently refine this specification across the levels of architectural abstraction (the architectural process), the ADL must support connector specification and it must be in a form that is open for customization. To contrast, a black-box connector does not convey its meaning and cannot be further refined. Second, it is required for the connector to support composition that is sufficiently expressive to support complex composition, as detailed in Section 2. This requirement is based on the *Semantics* and *Evolution* features of the Classification Framework [22].

2. ***Multi-Platform Support.*** First, the ADL is highly independent of the details of a specific run-time platform. It facilitates generic specifications in the face of heterogeneous component models. This avoids postponing important decisions until such a platform is chosen, e.g. the specification of complex compositions. Additionally, it avoids the risk of architects building architectures for one specific platform or make the wrong trade-offs because of it. Second, in the face of this heterogeneity, it is desirable for an ADL to come with the tools required to assist in the generation of implementation code, preferably to multiple platforms, that is consistent with the software architecture [22].

3. ***Multiple Views and Viewpoints.*** The description of the architecture occurs in terms of multiple views and viewpoints. Each view describes the architecture from the perspective of a particular stakeholder concern (or a set thereof). Each view is constructed according to an architectural viewpoint that matches the expertise of the stakeholder. Support for this requirement means a clear multi-view description across a set of architectural viewpoints. For distributed systems such a set would have to include at least the viewpoints of module, component-and-connector and deployment [9]. Examples of views can be seen in Figure 2, which shows a graphical representation of a component-and-connector view, but also the composition in Figure 3 is considered a view. We build on the *Multiple Views* feature in [22].

4. ***View and Viewpoint Relations.*** First, the views in an architectural description of a system are always related but often only implicitly, in part because they describe different perspectives of the same system. But some views are related more closely and elicit explication. An example is the *unification* of identical elements in different views. Explicit descriptions of relations between views has many uses [12], some of which are essential to our goal, such as composition and model transformation. Second, this requirement encompasses support for explicit relations between views of different viewpoints.

3.2 Existing Approaches

Table 1 shows the features that are supported by various approaches in the related work. For each requirement we discuss the most important observations.

	1. Open Connector	2. Multi-Platform Support	3. Multiple View(point)s	4. View(point) Relations
[33] DAOP-ADL (a)	○			
[34] AO-ADL (b)	○	●		
[32] Fractal-ADL (c)	○	○		
[24] AspectLEDA (d)	○	○	○	
[15] AspectualACME (e)	○	○		
[27] π-ADL^{ARL} (f)	○	○	○	○
[31] Prisma (g)	○	●		
[11] View Composition (h)		○	●	○
[19] Multi Perspective (i)		●	●	○
[8] Stratified Frameworks (j)		○	○	○
[25] Viewpoints Framework (k)		○	●	○

○ – limited support ● – support

Table 1. Feature matrix for requirements.

1. ***Open Connector.*** ADLs a–g support a connector that is open for customization. However, the connector often has semantics that are constrained to a specific composition model that, for instance, does not support deployment context (see problem illustration in Section 2). The approaches h–k have connectors —which act as black boxes— with pre-defined semantics, that do not allow customization. Approach j makes abstraction of the connector altogether.

2. ***Multi-Platform Support.*** ADLs b–g have not been created for use in a single platform. Fractal, AspectLEDA and π-ADL support generation to a single platform (FAC, and twice Java, respectively). Only AO-ADL and Prisma support code generation to multiple platforms. DAOP-ADL was specifically designed for the DAO-Platform. Approaches h–k support more generic descriptions by design, when compared to a typical ADL. Only the Multi Perspective approach supports generation to multiple platforms.

3. ***Multiple Views and Viewpoints.*** π-ADL^{ARL} and AspectLEDA support a structural and behavioral viewpoint. Stratified frameworks supports only a structural viewpoint. Approaches h, i and k, on the other hand, support an arbitrary number of views and viewpoints.

4. ***View and Viewpoint Relations.*** π-ADL^{ARL} supports a refinement relation between intermediate architectural

views, however, limited to a single viewpoint (component-and-connector). The ADL does not support refinement in terms of hosts, for example. The generic view approaches h–k support relations between views as well. Specific cross-viewpoint relations are considered feasible but are not detailed.

4. MView

MView is a technique for multi-view refinement of architectural connector descriptions. It facilitates the decomposition of a complex monolithic connector into connector descriptions that are no longer tangled with stakeholder views. To apply and validate MView, we have integrated it in the MViewADL prototypical ADL.

First, we explain and illustrate the core concepts of MViewADL, without going into the refinement details. Then we provide a detailed explanation of refinement. We end with an overview of the available tool support.

4.1 Introducing MViewADL

MViewADL is an ADL for component-based distributed systems, that supports aspect-oriented composition. The running example of this section is a refinement scenario of the compositions of both the *Accounting* and *Authorization* services from the e-Media system.

```
1 component NewspaperService {
2   provide { NewsRemote, ManagementRemote }
3   require { UserProfileRemote, ContentBrowseRemote }
4 }
5
6 abstract connector ServiceUsageCn {
7   abstract ao-composition ServiceUsage {
8     pointcut {
9       kind: execution;
10      signature:  ContentItem fetchArticle(ContentItemId),
11                  List listNewestArticles(int),
12                  List listArticlesForTag(Tag),
13                  List listArticlesForCategory(Cat);
14      callee { interface: NewsRemote; }
15    }
16  }
17 }
```

Figure 4. The *component*, *connector* and *AO-composition* declarations.

MViewADL has two kinds of *module* declarations: the *component* and the *connector*. Both modules are shown in an example in Figure 4. A component has a name and it contains a *provide* and optional *require* element as its only members. These describe the dependencies of the component in terms of the *interface* declarations that it provides and requires, respectively. The example shows the NEWSPAPERSERVICE component with its dependencies (line 1).

The connector describes the composition between components. A connector has a name and the following members: the optional *require* and *provide* elements, and zero or more *AO-composition* (AOC) declarations. The AOC consists of a single *pointcut* and a single *advice* member. Either member is optional. Without both present, the AOC is abstract. A connector that contains at least one abstract member, is abstract itself. We show in the next section that refinement is used to introduce missing or abstract members.

A *pointcut* has a *kind* member, which can be `call` or `execution`, and a signature member. The signature is a comma-separated list of method patterns, supporting negations and wildcards to define the set of join points for composition. A pointcut supports further constraining in terms of the `caller` and `callee` join point context of the composition. The kind, signature, caller, and callee members of the pointcut are related through conjunction.

The caller and callee element can constrain the pointcut further in terms of the following join point context properties: the *interface*, the *component*, the *host*, the *instance* and the *application*. Each property accepts a comma-separated list of definitions from its respective type, and it supports negation, like the signature does.

The *advice* has a *type* (before, around, or after) and a *service* member that references a method that acts as the advice. The specific method is resolved through the interfaces in the *require*-member of the connector.

The connector in the example is SERVICEUSAGECN. It has a single AOC member called SERVICEUSAGE. The AOC is abstract because it is incompletely specified, as it contains only a pointcut element (lines 8–15) and no advice. The pointcut is of kind *execution* (line 9). This means that interception needs to takes place at the callee side. The pointcut has a signature comprising the four *Service Usage* methods (lines 10–13). The pointcut is further constrained in terms of the callee context: the called component must provide the NEWSREMOTE interface (line 14).

```
1 abstract application NewspaperApp {
2   host PublishingServer; host ContentManagementServer; host
        ServicesServer;
3   ContentManagementSystem cms on ContentManagementServer;
4   NewspaperService    ns  on PublishingServer;
5   // other host and instance declarations cut
6 }
```

Figure 5. The *application* declaration.

Figure 5 shows a basic example of an application declaration. An application consists of zero or more members of each of the following declarations: the *host*, the *module instantiation*, and the *inline module*. An application is abstract if it contains a member that is abstract, it has no hosts, or it has no instances.

A host declaration has a name, and, optionally, a mapping to a real-world host (`host NS is "news.com";`). Host declarations are used to represent host deployment topologies. A module instantiation has a name, a type that refers to a component or connector declaration, and a host on which the instance must be allocated, prefixed with the `on` keyword. An inline module declaration is a component or connector declaration that is specified inline, as a part of the application description.

The application declaration in the example is called NEWSPAPERAPP. It declares a number of abstract hosts: PUBLISHINGSERVER, SERVICESSERVER, etc. (line 2), and a number of instances: *cms* and *ns*. ns is an instance of the NEWSPAPERSERVICE component that is allocated on the PUBLISHINGSERVER host. This application does not specify any inline module declarations. Subsection 4.2 on refinement contains more elaborate examples of the application type.

4.2 Multi-view Refinement

In this subsection we provide a detailed explanation of multi-view refinement (MView) of architectural descriptions.

A key feature of MView is explicit support for the step-wise refinement [7] and redefinition of pointcuts in AO-connectors. Pointcuts are refined based on context information specific to the role of the stakeholder: e.g. the allocation of components on host, new components that provide additional join points. Another MView feature is the refinement of application deployment topologies. This enables the definition of additional context for composition in terms of instances, hosts, etc.

First, we introduce the general concept of MView refinement and provide more context for the running example. Then we explain the refinement of declarations, followed by an explanation on how inherited elements can be redefined.

4.2.1 The MView Concept

In Section 3, we illustrated the problem of *view tangling in connector description* by means of the Accounting composition in Fractal-ADL. Now, we introduce some more context for the example from the requirements of the case study:

"Any consumer can view the headlines and summaries for the newest articles, free of charge. However, only a signed-up consumer can browse and access full-content articles. Whenever a consumer makes use of paid services, both Authorization and Accounting apply. Authorization regulates consumer access to these services, while Accounting needs to keep track of consumers using these services."

After analyzing this further, we conclude that the compositions of Authorization and Accounting intersect at the same four methods: *fetchArticle*, *listNewestArticles*, *listArticlesForTag*, and *listArticlesForCategory*. We summarize composition at these methods as *Service Usage*.

Figure 6 illustrates the untangled views and the possibility for hierarchical reuse between the Authorization and Accounting compositions. The SERVICEUSAGECN connector (at the top) captures the composition with the *Service Usage* methods. Both the Authorization and Accounting compositions share this view. The connectors for both compositions reuse the SERVICEUSAGECN connector description by refining it. The figure shows that the untangling of stakeholder views in compositions is achieved by means of step-wise refinement of connector descriptions. Each connector —possibly a refinement of the previous— captures the concerns of a

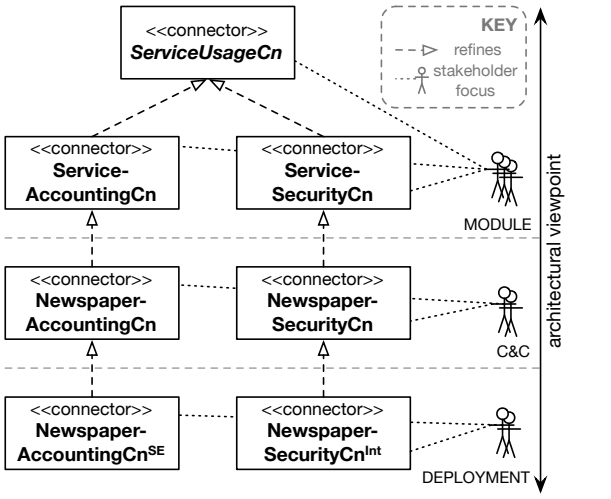

Figure 6. Eliminating tangling using refinement.

particular stakeholder as a *view* on the composition. A stakeholder expresses his view in terms of the knowledge associated with the *architectural viewpoint* that matches his expertise.

Only the availability of context knowledge imposes constraints on the order in which the views can be defined. In this paper we apply the scenario that is typical for the development of distributed middleware applications. There, development adheres to a structure that consists of three stages and developer roles [5]: *(a)* development of components and connectors by a module developer, *(b)* composition of components into applications by an application assembler, and *(c)* deployment of applications by a deployer. We start at the module development stage. In this stage, the description of compositions, in connectors, is limited to the knowledge of the module viewpoint (components, interfaces, methods, etc.). Then, in the application assembly stage, connectors can be created, and existing connectors can be refined, in terms of previous, and additional context (instances, host allocation, etc.). Finally, the deployment stage makes hosts explicit and allows connector refinement in this context. MView supports alternative scenarios where one starts at the application assembly stage with an abstract application that is further refined as development progresses with components, connectors, hosts and instances.

4.2.2 Refinement of Declarations

We start the running example at the top of Figure 6 and move through it one refinement at a time. We limit the explanation to the composition of Accounting, as Authorization employs refinement in a similar manner.

The first connector, SERVICEUSAGECN, was explained in our discussion on MViewADL in Section 4.1 (see Figure 4).

Figure 7 shows two examples of refinement: the SERVICEACCOUNTINGCN connector that refines SERVICEUSAGECN,

```
1 connector ServiceAccountingCn refines ServiceUsageCn {
2   require { AccountingRemote }
3
4   ao-composition ServiceUsage refines
        ServiceUsageCn.ServiceUsage {
5     advice {
6       service: chargeForService();
7       type: after;
8     }
9   }
10 }
```

Figure 7. Refinement of a Connector and an AO-composition with an *advice* member.

and the ServiceUsage AO-composition that refines its parent by the same name.

Multi-view refinement of declarations is similar to the familiar technique of inheritance in object-orientation, but it differs in the kinds of members that can be *inherited*, and in the way these members can be *redefined*. For instance, by default, pointcut members (elements without a name) will be merged under refinement, instead of overridden.

If declaration A ***refines declaration*** B**,** A has a *refines*-part that starts with the `refines` keyword (line 1), followed by the name of declaration B. The refinements that A can perform are: *(1)* redefine members inherited from B (*merge* or *override*), *(2)* introduce members that are missing from B, if B is abstract, or *(3)* add new members. In addition, A inherits all members of B that are not redefined in the body of A. In this relationship B is the parent and A the child. Refinement is only supported between declarations of the same declaration type, and it is not allowed to create loops in the refinement graph.

In MViewADL, the declarations that are refinable are: the AO-composition, the connector, and the application.

The AOC refinement in Figure 7 (line 4) results in ServiceUsage inheriting all members of its parent by the same name (line 7 in Figure 4), without redefinition. Additionally, refinement adds an *advice* element to the AOC. The new advice element declares the *chargeForService()* method as advice of type *after*. *chargeForService* is a method from the AccountingRemote interface that plays the roll of advice for this composition. The AOC does not further refine the inherited pointcut element.

The ServiceAccountingCn connector on line 1 refines the ServiceUsageCn connector of Figure 4. The connector inherits the ServiceUsage AOC from its parent. However, because both the newly defined AOC and the inherited AOC share the same name, the inherited one is overridden. It is also necessary to supply the fully qualified name of the AOC that is being refined, because the ServiceUsage name now points to the local one in this context (line 4). Additionally, the connector defines the *require* element (line 2). This dependency on the AccountingRemote interface is required by the *advice* element in the ServiceUsage AOC, as AccountingRemote provides the *chargeForService()* method.

The goal of this ServiceAccountingCn connector is to compose the *Accounting* service. It does this by reusing a pointcut that was defined in a parent connector, ServiceUsageCn, and by then refining it with the accounting advice specified here. Similarly, the ServiceSecurityCn connector (Figure 6) composes the *Authorization* service by refining the same parent connector with advice that handles authorization.

The connector and AOC descriptions in figures 4 and 7 are all done in the context of the Module viewpoint.

4.2.3 Redefinition of Members

The child in a refinement relation inherits all members from its parent, except for the ones that it redefines. The semantics of the redefinition depend on the kind of the member that is redefined.

When a child inherits a member that is a declaration and that already exists locally, redefinition always implies *override*.

Overriding an inherited declaration happens when the child in a refinement relation has a local declaration that has the same name as the declaration that is inherited. As a result, only the local declaration is accessible.

Because override is the only possible behavior between declarations, emphasizing it by adding the `override` modifier to a declaration means that override is the only desirable outcome. In MViewADL, these declarations are: the *component*, the *connector*, the *AOC*, the *host*, and the *instantiation*.

When a child inherits a member that is not a declaration (member without a name) and that already exists locally, redefinition defaults to the *merge* technique.

Merging elements K ***and*** M is a structurally recursive operation. First, it is verified whether the locally declared element has the `override` modifier, or if it is a declaration. This turns the *merge* behavior into *override*, which result in choosing the locally declared element over the inherited element. Second, if it *is* a merge, all members of K will be merged, type by type, with members of element M, according to the merge semantics of that type. This is repeated until an element is reached that does not have a body, that is a declaration, or that demands override semantics.

In MViewADL, the elements without a name, that support merge are the *require*, the *provide*, the *pointcut*, its *signature*, the *caller* and *callee*, and their context properties: *interface*, *component*, *host*, *instance* and *application*, and finally, the *advice*. There are three elements that do not support merge, because they can hold only a single value: the pointcut *kind* (Figure 4), and the advice *type* and *service* (Figure 7).

We have previously discussed the refinement of the connector and the AO-composition. Now we consider the refinement of an *application* declaration (Figure 8). In the example, the AccountedNewspaperApp application refines NewspaperApp (Figure 5). The AccountedNewspaperApp applica-

```
1 abstract application AccountedNewspaperApp refines NewspaperApp{
2   host AccountingServer;
3
4   AccountingService accServs  on AccountingServer;
5   NewspaperAccountingCn accConn on PublishingServer;
6
7   connector NewspaperAccountingCn refines ServiceAccountingCn {
8     ao-composition ServiceUsage refines
          ServiceAccountingCn.ServiceUsage {
9       merge pointcut {
10        merge callee {
11          override component: NewspaperService;
12        }
13      }
14    }
15  }
```

Figure 8. The NEWSACCOUNTING connector refining the SERVICEACCOUNTING connector with a component condition; in the context of the ACCOUNTEDNEWSPAPERAPP.

```
1 application NewspaperAppDeployment refines
      AccountedNewspaperApp, SecuredNewspaperApp, ... {
2   host PublishingServer is "pub0.news.com";
3   host StagingServer    is "stage.internal.news.com";
4   // other host declarations cut
5   NewspaperService  stagingNS on StagingServer;
6   NewspaperAccountingCn accConn on PublishingServer;
7
8   connector NewspaperAccountingCn refines
        AccountedNewspaperApp.NewspaperAccountingCn {
9     ao-composition ServiceUsage refines AccountedNewspaperApp.
          ...NewspaperAccountingCn.ServiceUsage {
10      pointcut { callee { host: ! StagingServer; } }
11    }
12  } // security connector refinement cut
13 }
```

Figure 9. The NEWSACCOUNTINGCN connector refining its parent with a host condition; in the context of the NEWSPAPERAPPDEPLOYMENT.

tion inherits all of its members, and adds a host declaration (line 2), a component and a connector instantiation (line 4–5), and a connector declaration (line 7).

The NEWSPAPERACCOUNTINGCN connector, in this application, refines the SERVICEACCOUNTINGCN connector from the previous example (Figure 7). The connector adds one member: the SERVICEUSAGE AOC. This AOC is a refinement as well. It refines the pointcut that it inherits from its parent. In this scenario, the composition is further constrained by limiting interception to the those callee components that are of type NEWSPAPERSERVICE. In other words, only those join points remain where the callee is of component type NEWSPAPERSERVICE. Because the refining AOC shares the same name as its parent, the parent is overridden instead of inherited.

We have explicitly added the superfluous `merge` modifier to the pointcut and callee elements to illustrate the contrast with the `override` component element. In this example, a stakeholder has specified that all prior definitions of *component* are to be ignored for the callee.

Next, the signature is merged with the signature of the parent by concatenating their lists of members. This conforms to an implicit disjunction with *super*, if both elements define a signature. Merging the caller (or callee) element means merging each of the context properties of the same type (interface, instance, etc.). Merging these is similar to merging the signature element.

Finally, when merging a pointcut (line 9 in Figure 8), the *kind* of the child overrides the *kind* of the parent, if both elements define a kind. This is because the kind can only hold a single definition (either execution or call). In addition, at any point in this explanation, the `override` keyword may force an override. Similarly, when merging the advice, the *type* (and *service*) member of the child overrides that of the parent, if both child and parent define it.

The application descriptions and connector refinements are all done in the context of the Component-and-Connector viewpoint.

Finally, the non-abstract application specification in Figure 9 refines the previous application specifications (line 1) in the context of the Deployment viewpoint. Every inherited abstract host declaration is overridden with a host that defines a physical system (lines 2–4). Connectors are refined in terms of deployment concerns as well. In this case, the deployer set up a *staging environment* where employees can perfect layout and editing before publishing to the production server. The STAGINGSERVER hosts a NEWSPAPERSERVICE instance (line 5) that serves up the online newspaper for internal review. As the *accounting* service is not required for the instance on this host, the NEWSPAPERACCOUNTINGCN connector is refined to exclude the STAGINGSERVER host (line 10).

4.3 Tool-support and Implementation

We have implemented and validated MView in an Eclipse plugin [2]. The reusable core of this plugin is a parser for the MViewADL, which supports multi-view refinement. The parser has been developed using ANTLR [29]. The language meta-model is built on top of Chameleon [37] —an in-house meta-framework for programming language construction. We support automatic code generation for particular middleware systems, which is further detailed in Section 5.

5. Code Generation for Middleware Systems

In this section we discuss automatic code generation for distributed middleware systems. We currently support generation to two of the more industry-ready application middleware platforms, JBoss [1] and Spring [3]. Because JBoss and Spring are related to some extend, we limit our discussion to JBoss.

We continue the running example and discuss the result of the automatic generation of the Accounting connector into an implementation artifact for JBoss. Figure 10 shows the result of this generation: a JBoss aspect class (line 3). The aspect is called NEWSPAPERACCOUNTING. This name is taken from the connector type that is instantiated in our refinement scenario in the previous section (Figure 8, line 5).

The MViewADL language model supports resolving the refinement relations between the different ao-composition

```
1 package accounting;
2 import accounting.AccountingRemote; // other imports cut
3 @Aspect  public class NewspaperAccounting {
4   public static String[] VALID_HOSTS = {};
5   public static String[] INVALID_HOSTS = {"unit..."};
6   /* the pointcut definition */
7   @PointcutDef(
8   "execution(ContentItem *->fetchArticle(Co..Id))" +
9   "OR execution(List *->listNewestArticles(int))" +
10  "OR execution(List *->listArticlesForTag(Tag))" +
11  "OR execution(List *->listArticlesForCat..(Cat))" +
12  "AND class($instanceof(NewsRemote))" )
13  public static Pointcut newspaperAccounting;
14  /* required for advice */
15  @EJB private AccountingRemote accountingRemote;
16  /* advice method /*
17  @Bind(
18    pointcut="newspaperAccounting",
19    type=AdviceType.AFTER,
20    cflow="NpAccountingHostConditions")
21  public void chargeForService() {
22    accountingRemote.chargeForService();
23  }
24 }
```

Figure 10. A JBoss aspect class generated from the NEWSPAPERACCOUNTING connector.

descriptions of the NEWSPAPERACCOUNTING connector. This results in a complete specification for that particular composition. The generator uses this specification to output the JBoss pointcut (line 13) and advice (line 21) in Figure 10.

The pointcut is configured by means of the `PointcutDef` annotation (line 7). It is described in the JBoss pointcut language, but we can clearly recognize the *execution* of the four *service usage* methods on a component implementing the NEWSREMOTE interface.

The advice body (line 21) denotes a call to a business method on an instance of a component implementing the ACCOUNTINGREMOTE interface. To retrieve such an instance, JBoss uses dependency injection (line 15). The advice is linked to its pointcut using the `@Bind` (line 17) annotation. It references the name of the pointcut and includes the type of advice: `after`.

The host-conditions on the composition with a callee component are shown on lines 4 and 5. A callee component conforms if it is deployed on a host in the valid list, but not in the invalid list.

Additionally, Figure 10 shows the various concerns of multiple stakeholders in this composition by means of the gray areas. The three shades of gray retain the same meaning as in the motivating example in Section 3 (the roles of module developer, assembler, and deployer).

Verifying Host Conditions via Dynamic CFlow The generation to JBoss is seldom a one-to-one mapping from source to target model. JBoss has some limitations that require specific techniques to solve. One of these is the verification of host conditions.

JBoss does not support reasoning about host allocation in the pointcut. Alternatively, we use a dynamic control flow, or cflow, statement that determines at runtime whether the advice should execute or not.

Whenever a pointcut specifies host-conditions, we generate a cflow class that verifies whether the callee is not allocated on the testing host "UNIT.NEWS.COM", every time after the pointcut matches, but before the advice is executed.

The cflow class in this example is called NPACCOUNTINGHOSTCONDITIONS, and it is coupled to the aspect by means of the cflow-attribute of the `Bind` annotation (Figure 10, line 17) of the advice.

6. Evaluation

In this section we evaluate how MView refinement affects the development effort for the different stakeholders in comparison with other architectural techniques. We conclude by revisiting the four requirements from Section 3.

6.1 Stakeholder Effort

We used the MViewADL to specify the architectural structure of the e-Media case study. The case has four architectural concerns that are composed through AO techniques, namely *accounting*, *user tracking*, *personalization*, and *authorization*. This amounts to 11 connectors and a total of 24 *refinements*, distributed evenly across the four architectural concerns.

In this evaluation we compare the effort, that is required of stakeholders to define such views, between MView refinement and two other, less systematic, techniques for architectural description.

To estimate the effort, we use the Lines of Code metric (LOC) in an absolute as well as a relative comparison. The two other techniques we compare MView to are (a) the *import of unchanged descriptions* and (b) *manual specification*. We use the MViewADL syntax in each of these cases to avoid representational differences from skewing the measurements.

Technique (a), called *Import*, is that of the import of unchanged specifications between two artifacts. Just like with refinement, all elements within the parent are imported, unless they are overridden. However, *Import* does not support connector refinement. Each connector that requires refinement must be manually copied and locally adjusted. An approach that applies this technique is Fractal-ADL [32].

Technique (b), called *Flatten*, is that of manual copy and local adjustment. Stakeholders put all their descriptions, belonging to a single stage in the process (e.g. Module), into a single artifact. Instead of refinement, stakeholders in a later stage manually copy the specification before adding the necessary changes. Approaches that apply this technique are DAOP-ADL, AO-ADL, and AspectualACME [15, 33, 34].

For both techniques, we only count the lines of code after adjustment is completed. Replaced or removed lines are disregarded. The results of our evaluation are presented in Table 2. The numbers in the row named *Total* represent the to-

	MView	*%*	*Import*	*%*	*Flatten*	*%*
Components	150		150		150	
Interfaces	74		74		74	
sum (x)	224		224		224	
Module	90	*21*	123	*21*	105	*17*
Assembly	80	*19*	162	*28*	149	*24*
Deployment	38	*9*	75	*13*	156	*25*
sum (y)	208	*49*	360	*62*	410	*66*
Total $(x + y)$	432		584		634	

Table 2. Stakeholder effort in terms of codebase size (LOC)

tal size of the architectural description codebase for the three techniques. From these LOCs numbers we conclude that the total effort to construct the application is considerably less with MView (432 vs. 584 and 634 LOC, respectively 26% and 32% less LOC). This total consists of the sums of the sizes of components and interfaces (sum x); and the sizes of connector descriptions in Module, Assembly and Deployment (sum y).

The effort for the components and the interfaces is the same in each approach (224 LOC) —their syntax is identical. Therefore, this reduction in effort can be completely attributed to the connector descriptions in Module, Assembly and Deployment (208 vs 360 and 410 LOC, respectively 42% and 49% less LOC to compose and deploy the application). The relative effort of the connector descriptions, in comparison with the total effort of a certain technique, also decreases: MView requires 208 out of 432 LOC for this process (or 49% of the total effort), while the other approaches require 62% (Import) and 66% (Flatten).

We now focus on the effort for the assembler and deployer. These are the stakeholders that are responsible for the descriptions in *Assembly* and *Deployment*.

It is the job of a deployer to refine the descriptions with physical host allocations and deployment-level connector refinements. Their effort when applying each of the three techniques, is considerably less with MView: 38 vs 75 and 156 LOC. The relative effort for MView is 9%, vs 13% and 25% for *Import* and *Flatten* respectively. Flatten requires the most effort, as the deployers need to copy the entire application assembly description, before carefully adding changes. Import does a lot better, as the adjustments of the deployers are limited to the host allocations and the changes to the full copies of a small number of connector descriptions. MView performs best, as refinement allows the changes to connectors to be done in terms of the smaller delta.

A similar conclusion applies to the *application assembler* stakeholder in the assembly stage: 80 LOC for MView vs 149 LOC and 162 LOC for the other techniques. The relative effort is respectively 19%, vs 28% and 24%, for *Import* and *Flatten* respectively. A lot more connector refinements are performed at this stage which explains the bad performance of Import.

The composition effort in *Module* is 90 LOC for MView vs 123 and 105 for the other techniques. The relative effort for these tasks is 21% for MView, vs 21% and 17% for *Import* and *Flatten* respectively. For MView, a relatively big size of the descriptions is pushed to the connectors in *Module* as the reusable bits are defined in these connectors (see Figure 4), while the other stages can be described as a delta. However, MView can still express this in lesser LOC in comparison to the other techniques. This is because of internal reuse in *Module* connectors.

In summary, MView does not only reduce the development effort in absolute numbers, it also reduces the effort for the assembler and the deployer relatively to the total specification size. This is because more of the composition workload is pushed towards the stakeholders in the earlier development stages, and can be easily reused.

6.2 Revisiting the Requirements

MViewADL builds strongly on each of the requirements from Section 3 to reach our goal of allowing different stakeholders to separately specify their views on the architectural descriptions. MViewADL allows views of different viewpoints to be related together in a way that is generic enough to be supported by multiple relevant technologies.

Open Connector. MViewADL supports connector specifications that are customizable through *refinement*. Currently, refinement has complete access to a parent's description. This may not always be desirable. The study of accessibility modifiers like *private* and *final* is left to future work.

Multi-Platform Support. MViewADL focusses on distributed AO-Middleware. Our toolchain already supports generation to JBoss and Spring.

Multiple Views and Viewpoints. MViewADL currently has support for the three important viewpoints in distributed systems design: *module*, *C&C*, and *deployment*. While *refinement* is broadly applicable, the syntax and semantics of the ADL need to be extended to support additional viewpoints.

View and Viewpoint Relations. View relations in multi-viewpoint representations are challenging [25]. While MView supports refinement between views of different viewpoints, this is only possible if the viewpoints share a clear goal —the description of distributed systems. Adding arbitrary viewpoints might prove challenging as well.

Discussion. The validation of MView and the ADL is limited to an application case study in the e-media domain. However, our ADL is applicable beyond this specific application and this specific domain, to typical distributed component-based architectures. To further validate this, it is part of future work to apply MView in additional case studies. As stated in the introduction, architectures for distributed systems are the focus of MView. This strong focus, however, limits its applicability in software systems with different architectural styles, such as embedded systems, or the internal structures of compilers [9].

7. Related Work

The typical AO-ADLs, that have also been considered in the analysis in Section 3: *DAOP-ADL* [33], *AO-ADL* [34], *Fractal-ADL* [32], *AspectLEDA* [24], *AspectualACME* [15], *π-ADL* [27], *Prisma* [31], all use some form of AO-Connector. While these connectors capture complex composition to a certain degree, they do not support the distribution context in distributed software systems, nor stepwise refinement. While inheritance could be used, instead of refinement, to achieve a similar separation (without the *merge* operation) of stakeholder concerns in a complex composition, it is not supported by default in these AO-ADLs. As it would require some restructuring of the composition specifications in some, if not all, of these languages. Finally, with the exception of *AO-ADL* and *Prisma*, tool-support for the generation to multiple platforms is missing.

AO-middleware technologies such as JBoss, Spring, GlueQoS [40], CAM/DAOP [33], DADO [39], FAC [32] and Prose [35] do not support the evaluation of distributed context properties. Supporting those platforms in the code generation thus needs a similar approach as presented in the generation to JBoss. On the other hand, platforms such as JAC [30], AWED [23], ReflexD [36] and DyMAC/MStage [21] do support the evaluation of distributed context properties, greatly simplifying the generation to these platforms.

MStage is an extension to the DyMAC platform to develop DyMAC applications over different stages using refinement. The refinement is limited to a fixed set of context properties of DyMAC components such as interfaces, component names, hosts and applications. MView ADL, however, offers a middleware-independent multi-platform ADL that supports refinement at the level of architectural views and over an open set of properties.

ArchJava [4] brings user-defined connectors to an OO-programming language. The connectors are customizable and the strong presence of inheritance should allow stakeholder separation. *ArchJava*, however, does not support AO-Connectors. While MView does not focus on implementation specifications, connectors at this level would simplify code-generation.

Batory et al. present a software composition model and associated tool set, called AHEAD [10], that supports large-scale refinement of aspect-like modules in a product family. There are important differences between AHEAD and MView. First, MView has a more focused goal, it supports stepwise refinement of interaction, not behavior. Second, MView supports stepwise refinement across multiple views and viewpoints at the architecture level, while AHEAD supports multiple levels of abstraction in the design of a software system. Finally, the AHEAD tool set does not target AO middleware.

Model-driven development of distributed software systems partially targets a similar goal as MView: a higher-level system description based on abstractions above platform-specific artifacts and implementation details. In model-driven middleware (e.g. [13, 18, 41]), multiple design models of aspects and applications can be specified, composed and possibly verified. Once composed, these models can be automatically synthesized to deployment descriptors for a specific (non-AO) middleware platform of choice [13] or to middleware implementations itself [41].

8. Conclusion and Future Work

In software architecture, AO-Connectors capture complex compositions between components. But state-of-the art AO-ADLs do not allow the separation of the various stakeholder views involved in such connectors. This results in monolithic descriptions in which these stakeholder views are tangled.

MView is a technique that enables multi-view refinement of architectural connector descriptions. Complex composition is specified through a process of multi-view refinement of connectors, each in the context of a specific stakeholder view.

We have integrated MView in a prototypical ADL and constructed an Eclipse-based tool to support MView and the ADL. The tool supports automatic code generation to specific middleware frameworks (currently JBoss and Spring).

Our evaluation of MView in the e-Media case study showed that MView reduces the overall stakeholder effort (in LOC), and the relative effort of assemblers and deployers with respect to the total specification size.

It is part of our ongoing work to further enhance the transformation framework and to validate the benefits of interaction untangling and reuse on additional case studies.

Acknowledgments

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, the Fund for Scientific Research (FWO) in Flanders and by the Research Fund KU Leuven.

References

[1] Redhat inc., http://labs.jboss.com/jbossaop.

[2] Mview tool, http://distrinet.cs.kuleuven.be/software/mview.

[3] The spring enterprise platform http://www.springsource.com/products/enterprise.

[4] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language support for connector abstractions. In *Object-Oriented Programming*, 2003.

[5] P. Allen and S. Frost. *Planning team roles for CBD*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[6] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997.

[7] S. Apel, C. Kästner, T. Leich, and G. Saake. Aspect refinement-unifying aop and stepwise refinement. *Journal of Object Technology*, 6(9):13–33, 2007.

[8] C. Atkinson and T. Kühne. Aspect-oriented development with stratified frameworks. *IEEE Software*, 20(1):81–89, 2003.

[9] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, second edition, 2003.

[10] D. Batory, J. N. Sarvela, A. Rauschmayer, S. Member, and S. Member. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30, 2003.

[11] N. Boucké. *Composition and Relations of Architectural Models Supported by an Architectural Description Language*. PhD thesis, October 2009.

[12] N. Boucké, D. Weyns, R. Hilliard, T. Holvoet, and A. Helleboogh. Characterizing relations between architectural views. In *LNCS*, volume 5292. Springer, September 2008.

[13] G. Deng. Resolving component deployment & configuration challenges for enterprise dre systems via frameworks & generative techniques. In *International Conference on Software Engineering*. ACM, 2006.

[14] P. Feiler, B. Lewis, S. Vestal, and E. Colbert. An overview of the sae architecture analysis & design language (aadl) standard. In *Architecture Description Languages*, volume 176. Springer Boston, 2005.

[15] A. Garcia, C. Chavez, T. Batista, C. Sant'anna, U. Kulesza, A. Rashid, and C. Lucena. On the modular representation of architectural aspects. In *Software Architecture*, volume 4344 of *LNCS*. Springer Berlin / Heidelberg, 2006.

[16] A. F. Garcia, E. M. L. Figueiredo, C. N. Sant'Anna, M. Pinto, and L. Fuentes. Representing architectural aspects with a symmetric approach. In *Early Aspects '09*. ACM, 2009.

[17] D. Garlan, R. T. Monroe, and D. Wile. Acme: An architecture description interchange language. In *CASCON'97*, 1997.

[18] J. Gray, T. Bapty, S. Neema, D. C. Schmidt, A. Gokhale, and B. Natarajan. An approach for supporting aspect-oriented domain modeling. In *International Conference on Generative Programming and Component Engineering*. Springer-Verlag New York, Inc., 2003.

[19] J. Grundy. Multi-perspective specification, design and implementation of components using aspects. *International Journal of Software Engineering and Knowledge Engineering*, 10(6), December 2000.

[20] ISO/IEC. Systems and software engineering - architecture description. *ISO/IEC standard, draft D8*, August 2010.

[21] B. Lagaisse. *A Comprehensive Integration of AOSD and CBSD Concepts in Middleware*. PhD thesis, K.U.Leuven, Dec. 2009.

[22] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1), Jan 2000.

[23] L. D. B. Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée. Explicitly distributed aop using awed. In *AOSD'06*. ACM, 2006.

[24] A. Navasa, M. A. Pérez-Toledano, and J. M. Murillo. An adl dealing with aspects at software architecture stage. *Inf. Softw. Technol.*, 51(2), 2009.

[25] B. Nuseibeh, J. Kramer, and A. Finkelstein. Viewpoints: meaningful relationships are difficult! In *International Conference on Software Engineering*. IEEE Computer Society, 2003.

[26] F. Oquendo. π-arl: an architecture refinement language for formally modelling the stepwise refinement of software architectures. *SIGSOFT Softw. Eng. Notes*, 29, September 2004.

[27] F. Oquendo. pi-adl: an architecture description language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3), 2004.

[28] K. Palma, Y. Eterovic, and J. M. Murillo. Extending the rapide adl to specify aspect oriented software architectures. In *15th International Conference on Software Engineering and Data Engineering*, page 170. ISCA, 2006.

[29] T. Parr and R. Quong. Antlr: A predicated (k) parser generator, 1995.

[30] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Jac: A flexible solution for aspect-oriented programming in java. In *Reflection*, 2001.

[31] J. Pérez, I. Ramos, J. J. Martínez, P. Letelier, and E. Navarro. Prisma: Towards quality, aspect oriented and dynamic software architectures. In *International Conference on Quality Software*, 2003.

[32] N. Pessemier, L. Seinturier, L. Duchien, and T. Coupaye. A component-based and aspect-oriented model for software evolution. *Int. J. Comput. Appl. Technol.*, 31(1/2), 2008.

[33] M. Pinto, L. Fuentes, and J. M. Troya. A dynamic component and aspect-oriented platform. *Computer Journal*, 48(4), 2005.

[34] M. Pinto, L. Fuentes, and J. M. Troya. Specifying aspect-oriented architectures in ao-adl. In *Information and Software Technology*. Elsevier, 2011.

[35] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD'02*. ACM, 2002.

[36] E. Tanter and J. Noyé. A versatile kernel for multi-language aop. In *Generative Programming and Component Engineering*, LNCS. Springer Berlin / Heidelberg, 2005.

[37] M. van Dooren. *Abstractions for improving, creating, and reusing object-oriented programming languages*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, June 2007.

[38] D. Van Landuyt, S. Op de beeck, E. Truyen, and P. Verbaeten. Building a digital publishing platform using aosd. In *LNCS Transactions on Aspect-Oriented Software Development*, volume 8, December 2010.

[39] E. Wohlstadter, S. Jackson, and P. T. Devanbu. Dado: Enhancing middleware to support crosscutting features in distributed, heterogeneous systems. In *ICSE*, 2003.

[40] E. Wohlstadter, S. Tai, T. Mikalsen, I. Rouvellou, and P. Devanbu. Glueqos: Middleware to sweeten quality-of-service policy interactions. In *26th International Conference on Software Engineering*. IEEE Computer Society, 2004.

[41] C. Zhang, D. Gao, and H.-A. Jacobsen. Generic middleware substrate through modelware. In *Middleware*, 2005.