Modularity and Variability of Distributed Software Architectures through Multi-view Refinement of AO-Connectors

Steven Op de beeck, Marko van Dooren, Bert Lagaisse, and Wouter Joosen

iMinds-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 architect views on an architectural connector in distributed software systems.

A distributed system faces a lot of variability in the functions that it needs to support, and the deployment environments in which it needs to execute. Modularity of functionality and composition is essential in achieving this variability.

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 architecture views. This is, however, essential for distributed systems, where run-time and distribution characteristics are represented in multiple views. This results in connectors that suffer from monolithic descriptions, in which the views of different architects are tangled.

MView untangles these architect views by defining them in separate modules and specifying refinement relations between these modules. We have integrated MView in an ADL, called MViewADL, which is supported by an Eclipse-plugin that performs code generation to the JBoss and Spring middleware platforms.

We evaluate MView in a content distribution system for e-Media. First is an assessment of the architect effort required for creating variations of that system, followed by an evaluation of the extent to which changes ripple through the architecture.

1 Introduction

Architecting distributed applications involves *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 [41, 26, 24], which offer direct support for such complex compositions.

Moreover, architects are faced with variations of the application features as well as variations in the deployment environments. Supporting modularity and

Fig. 1. The development process in terms of architect roles and views.

variability of complex compositions are key challenges to achieve appropriate separation of concerns in distributed software architectures.

In software architecture, a *view* captures the concerns of *stakeholders* on specific facets of the system [21]. The architecture description of a system uses various views that each consist of one or more *architecture models*. In the case of distributed software systems, each model is described in accordance with one of, at least, the architecture views of *Module*, *Component-and-Connector* and *Deployment* [9]. These views match the expertise of developer stakeholders with the following developer roles: component developers, application assemblers, and deployers, respectively. Figure 1 illustrates architect roles and matching views throughout the development process. Developer stakeholders will be called architects for the remainder of the paper.

To describe the models in these views, several Architecture Description Languages [6, 15, 18] (ADL) have been proposed. More recently, various AO-ADLs [35, 32, 16, 27, 38, 30, 37, 36, 17] have been defined that capture the crosscutting compositions of software systems in AO-connectors.

However, the goal of these ADLs differs from ours. While they aim to separate the crosscutting concerns of architects with role *c* (see Figure 1) by means of AO-connectors *k*, *l*, and *m*, they do not separate the various views of architects *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*.

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 *News*-*paperService* *component instance*, and located on the *PublishingServer* *host*.”

This complex composition suffers from a monolithic description, where the views of different architects are tangled. We discern the following expert architects: 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 architects 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.

In summary, such monolithic descriptions clearly lack the modularity to achieve appropriate separation of concerns across the different architecture views and roles. As a consequence, this lack of modularity also limits variability; managing multiple feature variations and deployment variations of such applications with optimal reuse of composition logic becomes nearly impossible.

We propose MView to improve the modularity and variability of architect views in the development process. MView is a technique for *multi-view refinement* of architecture descriptions that builds on inheritance and step-wise refinement [7]. In MView, a complex composition is modularized across multiple connector models, where each one deals with the composition in the context of the view of a *single* architect. For example, a model in the Module view adds module related constraints: *method M on interface I*. MView enables one connector model to refine another connector model. So, as the architecture is further developed, a connector can be refined by a model of an additional view, a deployment view that adds deployment constraints: *must be running on host R*. As a result, the complex composition is specified through a process of multi-view refinement of connector models. Each connector model is constructed separately, by the right architect for the job, at the appropriate time during development.

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

We have applied MView in a case study on e-Media [42] where we compare it with other techniques in terms of architect effort, across a broad set of variations in terms of feature composition as well as deployment environments. In addition, we evaluate the techniques by the extent to which they contain the ripple effects of changes to the architecture. 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 MViewADL, followed by a detailed explanation of MView refinement. Section 5 discusses automatic code generation to JBoss. Section 6 evaluates MView in an extended case study, in terms of architect effort, and the containment of ripple effect upon changes, and it revisits the requirements for MView. Section 7 presents the related work, and Section 8 concludes.

Fig. 2. A model of the component-and-connector view on the e-Media architecture.

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 [42].

Figure 2 shows a model of the 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 model, by means of a full line arrow. The accounting composition is tangled by the following architect views: Accounting must only interact with (a) the specific *service usage* methods of the `NewsRemote` interface, (b) 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 [36]. Listing 1 shows a complex composition description that realizes *Accounting*.

```

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>

```

Listing 1. The `ServiceAccounting` complex composition in FractalADL, combining contributions from three architects into a single description.

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*.

This *pointcutExp* representation is tangled with the views of multiple architects, as indicated by the different gray areas. The first one from the bottom (line 17–18) is a Module-view description that focuses 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 description (line 16) further specifies the pointcut in terms of the Component-and-connector view: 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.

In summary, this example illustrates how the views of different architects with respect to 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. As a consequence, managing modularity and variability across an application and its variations is a complex and error-prone task. In the next section we introduce requirements for techniques to help solve this problem.

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 architects that are involved with a composition, to specify their views on that composition separately and at the appropriate time during development—*the right architect 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 architects.

We break this goal down into four requirements for ADL features that we deem necessary. In a study of the related work, we determine for each approach whether it has none, limited, or full support for each of these requirements. The requirements are inspired by the work on the Classification and Comparison Framework for ADLs by Medvidovic, et al. [25].

Open Connector. 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 the specification of connectors, preferably in a form that

is open for adaptation. In contrast, a black-box connector does not convey its meaning and cannot be further refined.

In addition, the connector should also support a composition mechanism that is sufficiently expressive to capture complex compositions, with runtime and deployment characteristics, that are required by our case study (see Section 2).

A black-box connector is an example that has *no support* for this requirement. A connector that is open for adaptation has *limited support*. If a connector is open for adaptation and is sufficiently expressive, it has *full support*. This requirement is based on the *Semantics and Evolution* features of the Classification Framework [25].

Multi-platform Support. The ADL must be 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. In addition, it avoids the risk of architects building architectures for one specific platform or make the wrong trade-offs because of it.

Furthermore, in the face of this heterogeneity, the ADL comes with the tools required to assist in the generation of implementation code, preferably to multiple platforms, that is consistent with the software architecture. This part of the requirement is based on the *Implementation Generation* feature [25].

An ADL that facilitates generic specifications offers *limited support* for this requirement—it should not be targeting a single run-time platform. A generic ADL with the proper tools has *full support*.

Multiple Models and Views. Architecture description is performed by means of multiple models in terms of varying views. Each model, belonging to a view, describes the architecture from the *perspective* of the particular architect concern (or a set thereof) that this view addresses.

An ADL that supports multiple models that are, however, limited to a single architectural view, has *limited support* for this requirement. An ADL that supports multiple models across multiple architectural views has *full support*. For distributed systems the set of views that needs to be included, contains at least the Module, Component-and-connector and Deployment views [9].

Examples of a view can be seen in Figure 2, which shows a graphical representation of a component-and-connector view, and in Listing 1, which shows a text-based description of a Component-and-connector view. This requirement is related to the *Multiple Views* feature [25].

Model and View Relations. The models in an architectural description of a system are always related but such relations often remain implicit. These relations exist mostly because these models describe different parts of the same system. However, some models are related more closely and elicit an explicit description of the relation. For instance, the *unification* of identical elements in different models.

Table 1. The feature matrix for our requirements

	[38] DAOP-ADL (a)	[36] Fractal-ADL (b)	[27] AspectLEDAL (c)	[16] AspectualACME (d)	[11] View Composition (e)	[29] π -ADL (f)	[8] Multi Perspective (g)	[20] Stratified Frameworks (h)	[28] Viewpoints Framework (i)	[8] Multi Perspective (j)	[28] Viewpoints Framework (k)
Open Connector (1)	○	○	○	○	○	○	○	·	·	·	·
Multi-platform Support (2)	·	●	·	○	·	●	○	○	●	○	○
Multiple Models & Views (3)	·	○	·	○	·	·	●	●	●	○	●
Model & View Relations (4)	·	·	·	·	·	·	○	○	○	○	○

· – no support ○ – limited support ● – full support

Explicit descriptions of relations between models has many uses [12], some of which are essential to our goal, such as composition and model transformation. While relations between models of the same view can be expressed in a straightforward manner, once they are understood—a component in one model is the same kind of element as a component in another model—relations between models belonging to different views are challenging because these views are often inconsistent [28]. However, as it is our goal to support architects with different views in contributing to the same architecture, this is exactly what we are looking for with this requirement.

An ADL that supports relations between models in the same view has *limited support* for this requirement. In addition, an ADL that supports relations between models in different views, in particular the Module, Component-and-connector and Deployment views, has *full support* for this requirement.

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.

Open Connector. ADLs a–g support a connector that is open for adaptation. 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). Approaches h–k have black-box connectors with pre-defined semantics that do not support adaptation. The Stratified Frameworks approach makes abstraction of the connector by representing it as a line in its models.

Multi-platform Support. DAOP-ADL and Fractal were created with a specific platform in mind: the DAO-platform and the FAC-plaform, respectively. AspectLEDAL and π -ADL support generation to Java code. Only AO-ADL and Prisma

support code generation to multiple platforms. AspectualACME does not have particular generic descriptions and has no tool support. Approaches h–k support more generic descriptions by design, when compared to a typical ADL. Of these approaches, only the Multi-Perspective approach supports generation to multiple platforms.

Multiple Models and Views. Only ADLs b, d, and g clearly support descriptions that can consist of multiple models and only $\pi\text{-ADL}^{ARL}$ supports a structural and behavioral view. Stratified frameworks supports multiple models, but only in terms of a structural view. Approaches h, i and k, on the other hand, support an arbitrary number of models and views.

Model and View Relations. $\pi\text{-ADL}^{ARL}$ supports a refinement relation between models, however, limited to a single view (component-and-connector). The ADL does not support refinement in terms of hosts, for example. The generic view approaches h–k support relations between models as well. Specific cross-view relations are considered feasible but challenging and are not detailed.

In general. The approaches that we studied seemingly fit into one of two possible categories. Approaches a–g are part of the first category. These approaches focus on describing components and how they are composed together using connectors. They offer support for open connectors, however, without providing the possibility to capture complex compositions. These approaches rarely support multiple independent models, let alone models belonging to different views. As a consequence, they are not concerned with capturing relations between models. The only exception is $\pi\text{-ADL}^{ARL}$, which supports descriptions that have an integrated structural and behavioral view.

Approaches h–k are part of the second category. These approaches focus on capturing an architecture in terms of multiple models, often across multiple views, and the relationships between these models. Documenting cross-view relationships between models, however, remains challenging because of the general-purpose nature of these approaches and the views they support [28]. These approaches are not concerned with the details of the mechanisms used to compose components; a connector is represented as an abstract relation between two components.

In the following section we introduce MViewADL, our proposal for a description language that satisfies all of these requirements. We revisit this claim in our evaluation, in Section 6.3.

4 MView

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

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 }

```

Listing 2. 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 Listing 2. 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* member (before, around, or after). It has a *service* member that references a method that acts as the advice. Finally, it has an optional *instance* member that references a specific component instance to call the advice method on. The specific method that acts as advice, 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 take 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;
3   host ContentManagementServer;
4   host ServicesServer;
5   ContentManagementSystem cms on ContentManagementServer;
6   NewspaperService ns on PublishingServer;
7   // other host and instance declarations cut
8 }

```

Listing 3. The *application* declaration

Listing 3 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 *in-line 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 in-line module declaration is a component or connector declaration that is specified in-line, 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 4), 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 in-line 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 of architectural descriptions (MView).

A key feature of MView is explicit support for the stepwise refinement [7] and redefinition of pointcuts in AO-connectors. Pointcuts are refined based on context information specific to the role of the architect: e.g. the allocation of components on host, or 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.

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 3 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 model. The connectors for both compositions reuse the *ServiceUsageCn* connector description by refining it. The figure shows that the untangling of architect views in compositions is achieved by means of step-wise refinement of connector models. Each connector —possibly a refinement of the previous— captures the concerns of a particular architect as a *model* on the composition. An architect expresses this model in terms of the knowledge associated with the *architectural view* 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)

Fig. 3. Eliminating tangling using refinement.

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 view (components, interfaces, methods, etc.). Then, in the application assembly stage, connectors can be created, and existing connectors can be refined, in terms of previous context and Component-and-connector view context (instances, host allocation, etc.). Finally, the deployment stage makes hosts explicit and allows connector refinement in terms of previous context and Deployment 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.

Refinement of Declarations. We start the running example at the top of Figure 3 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 MView-ADL in Section 4.1 (see Listing 2).

Listing 4 shows two examples of refinement: the `ServiceAccountingCn` connector that refines `ServiceUsageCn`, and the `ServiceUsage` AO-composition that refines its parent by the same name.

```

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 }

```

Listing 4. Refinement of a Connector and an AO-composition with an *advice* member

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 Listing 4 (line 4) results in *ServiceUsage* inheriting all members of its parent by the same name (line 7 in Listing 2), 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 Listing 2. 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 3) composes the *Authorization* service by refining the same parent connector with advice that handles authorization.

The connector and AOC descriptions in figures 2 and 4 are all done in the context of the Module view.

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 this declaration *should* have an inherited parent to override. 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 four elements that do not support merge, because they can hold only a single value: the pointcut member *kind* (Listing 2), and the advice members *type*, *service* (Listing 4), and *instance*.

We have previously discussed the refinement of the connector and the AO-composition. Now we consider the refinement of an *application* declaration (Listing 5). In the example, the `AccountedNewspaperApp` application refines `NewspaperApp` (Listing 3). The `AccountedNewspaperApp` application 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 (Listing 4). The connector adds

```

1 abstract application AccountedNewspaperApp refines NewspaperApp {
2   host AccountingServer;
3
4   AccountingService    accSrvs 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  }

```

Listing 5. The NewsAccounting connector refining the ServiceAccounting connector with a component condition; in the context of the AccountedNewspaperApp.

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, an architect 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 Listing 5), 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*, *service*, and *instance* members of the child override 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 view.

Finally, the non-abstract application specification in Listing 6 refines the previous application specifications (line 1) in the context of the Deployment view. 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 re-

```

1  application NewspaperAppDeployment refines AccountedNewspaperApp, SecuredNewspaperApp,
    PersonalizedNewspaperApp {
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 }

```

Listing 6. The NewsAccountingCn connector refining its parent with a host condition; in the context of the NewspaperAppDeployment.

quired for the instance on this host, the NewspaperAccountingCn connector is refined to exclude the StagingServer host (line 10).

Multiple Refinement of Declarations. MView supports multiple refinement of declarations. Like multiple inheritance in object-orientation, this feature enables a declaration to refine more than one parent declaration. In MViewADL, all the declarations that are refinable (AO-composition, connector, and application) also support multiple refinement.

NewspaperAppDeployment, in Listing 6, is an example of an application declaration that refines multiple parent applications. MView will iterate through the set of parent declarations from last to first, resolving each refinement relation individually and aggregating the results. Conflicts are detected using the rule of dominance [40]. If a declaration inherits two different definitions for a member that cannot be merged, then it must provide a new definition to resolve the conflict.

StagedNpAccnCn, Listing 14, is a connector declaration from the extended e-Media case study used in our evaluation in Section 6. It shows how multiple refinement can be applied to connector and AO-composition declarations to modularize the *staging* solution from Listing 6 and enable its reuse. In this case, a conflict arises because both AO-compositions contain a pointcut. However, because the pointcut supports merge, this conflict is automatically handled, in the way that was intended.

4.3 Tool-support and Implementation

We have implemented and validated MView in an Eclipse plugin [1]. The reusable core of this plugin is a parser for the MViewADL, which supports multi-view refinement. The parser has been developed using ANTLR [33]. The language meta-model is built on top of Chameleon [14] —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 [2] and Spring [3]. Because JBoss and Spring are related to some extent, we limit our discussion to JBoss.

```
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 }
```

Listing 7. A JBoss aspect class generated from the NewspaperAccounting connector

5.1 JBoss AOP in a nutshell

The core concept is the aspect class, which supports an arbitrary number of pointcut and advice definitions. Definition of these constructs is done by means of annotation bindings or by providing bindings in separate XML artifacts.

We follow the growing trend in application development to include more middleware configuration as simple annotations in the code. The annotation approach leads to JBoss artifacts with the following characteristics:

1. An aspect is implemented as a plain Java class that is annotated with the `@Aspect` annotation.
2. A pointcut is declared by means of the `@PointcutDef` annotation, attached to a public, static `Pointcut` field. The pointcut description is defined as a `String`.
3. An advice method delegates business behavior to a business component. Advice is coupled with a pointcut by annotating the advice method with the

@Bind annotation. The `type` attribute of the annotation determines the type (before, around, etc.) of the advice.

4. JBoss supports a *dynamic cflow*¹ by means of a class that implements the `DynamicCFlow` interface, which imposes the `boolean shouldExecute()` method, and is annotated with `@DynamicCFlowDef`. A `cflow`-class is bound to an advice method by means of the `cflow` attribute of the `@Bind` annotation. It allows to execute additional code to decide whether to trigger the advice or not.
5. The `@EJB` annotation refers to *Dependency Injection* where type-information of the field is used to automatically find and inject a suitable component from a directory service.

5.2 Generating the NewspaperAccounting aspect

We continue the running example and discuss the result of the automatic generation of the Accounting connector into an implementation artifact for JBoss. Listing 7 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 (Listing 5, line 5).

The MViewADL language model supports resolving the refinement relations between the different ao-composition 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 Listing 7.

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, Listing 7 shows the various concerns of multiple architects 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).

¹ Dynamic CFlow in JBoss is completely unrelated to and inconsistent with AspectJ's `cflow` concept.

```

1  abstract class HostConditions implements DynamicCFlow {
2  private List<String> validHosts, invalidHosts;
3  // ... constructor, getters/setters ...
4  /** @return true if should execute; otherwise false */
5  public boolean shouldExecute(Invocation invocation) {
6      String hostname (new ServerInfo()).getHostName();
7      return validHosts().contains(hostname)
8          && ! invalidHosts().contains(hostname);
9  }
10 }
11 @DynamicCFlowDef
12 public class NpAccountingHostConditions
13     extends HostConditions implements DynamicCFlow {
14     public NpAccountingHostConditions() {
15         super(GoodbyeWorldCallee.VALID_HOSTS,
16             GoodbyeWorldCallee.INVALID_HOSTS);
17     }
18 }

```

Fig. 4. The abstract HostConditions dynamic cflow class and its extension for NewspaperAccounting perform runtime evaluation of host conditions

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 NpAccountingHostConditions class, an extension of the HostConditions class, is generated as well. This class initializes the validHosts and invalidHosts lists based on the String arrays in the NewspaperAccounting aspect (Listing 7, lines 4–5). This extending class is coupled to the aspect by means of the cflow-attribute of the Bind annotation (Listing 7, line 17) of the advice.

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 (Listing 7, line 17) of the advice.

Supporting Distributed AO in JBoss. MViewADL supports the description of a composition in terms of caller and callee distribution constraints simultaneously, e.g. the caller component instance must not be on host *A*, while the callee should be on host *B*. Because of the lacking distribution support in JBoss, an aspect that is deployed on host *B* has no way of knowing what host the caller is on, and vice-versa. A single JBoss aspect, deployed either on host *A* or on host *B*, cannot fully capture the MViewADL connector in this example.

To solve this we need two aspects, one on each host, that cooperate to collect the necessary context information. JBoss enables aspects to pass along data to each other via the invocation context, but only if these aspects are part of the

Fig. 5. Communication between remote aspects by means of a proxy component

local interception chain for the same join point. JBoss does not preserve this data when moving from the interception chain at the *call*-side to that of the *execution*-side, in case of a remote invocation.

We can enable the communication between remote aspects by means of the pattern that is shown in Figure 5. The original invocation 1' of `doService(a,b)` between `caller` and `callee` is intercepted by the `callerCtx` aspect 1.. The aspect collects the required context data, but instead of proceeding with the invocation, it calls the `CalleeProxy` component `doService(a ,b,metadata)` 2.. With this call, the aspect pushes the original target object (`calleeBean`), the original arguments and the additional data (`metadata`). The *execution* of this proxy call is intercepted by the `calleeCtx` aspect 3.. This aspect pops the metadata, aggregates it with its own data and decides whether to proceed with the proxy call. If it proceeds, `CalleeProxy` invokes the original call on the target object, with the original arguments 4..

6 Evaluation

In this section we evaluate MView refinement in terms of its support for variability and modularity of architectural descriptions in a case study of the e-Media system. First, in Section 6.1, we explain the application and deployment variations in the architecture of the extended e-Media system. Second, in Section 6.2–*Variability*, we use the extended e-Media system in an evaluation of MView where we compare it, in terms of architect description effort, to two techniques without refinement (*Import* and *Flatten*). Third, in Section 6.2–*Modularity*, we compare how MView and the two other techniques deal with the ripple effects of introducing changes into the architectural descriptions. We conclude by revisiting the four requirements from Section 3.

6.1 Architectural Variations in the Extended e-Media Case Study

We used the MViewADL to specify the architectural structure of the e-Media system. The e-Media system consists of a basic Newspaper service that is accompanied by four additional services, namely *Accounting*, *Personalization*, *User tracking*, and *Authentication*, by means of aspect-oriented techniques.

A well designed complex distributed system supports variations to the desired feature set, as well as variations based on the deployment into different enterprise environments. Supporting variation in the feature set implies design-level alternatives where architects are able to easily combine desired configurations. This enables architects to adapt the architecture to changes in the requirements over time. Supporting variation in the environment implies making sure that architects can easily adapt the architecture to changes in the deployment environment or to deploy it in a entirely different environment.

The e-Media system is extended with the following four variations in the two dimensions of feature set and deployment environment. We give a brief overview of these four variations, before describing them in detail in the remainder of this subsection.

1. Variations in the feature set
 - (a) **Load balancing.** Architects extend the e-Media system with an application-level load balancing solution. Selecting this variation implies creating an e-Media system with an additional Newspaper service during the design of the architecture.
 - (b) **Non-personalized.** Architects remove the Personalization and User tracking services from the e-Media system. Selecting this variation implies creating an e-Media system that consists of the basic Newspaper service, only extended with Accounting and Authentication.
2. Variations in the deployment environment
 - (a) **Multiple environments.** Architects design the e-Media system to support deployment in two environments which we will call CNN and NYTimes.
 - (b) **Staging environment.** Architects design a deployment-level extension of the e-Media system to support a staging environment.

Figure 6 illustrates graphically how these variations are structured in the architectural descriptions using MView refinement. The combination of the proposed variations (load balanced and non-personalized, CNN and NYTimes, as well as a staged and non-staged version) results in eight distinct variants of the application.

(1a) Load Balancing. The Load Balancing variation deals with the quality requirement of availability of the Newspaper service. The application-level load balancer in this case study is motivated by the work of Othman, et al. [31] on a Load Balancing service in Adaptive Middleware. Application-level load balancing provides the flexibility of making content-aware decisions based on application-defined metrics, at run-time.

Fig. 6. Overview of the e-Media system variations with MView refinement.

Figure 6 shows how the e-Media architecture descriptions are extended to support the load balancing requirement. The `LoadBalancedNewspaperApp` application description (detailed in Listing 8) instantiates a `LoadBalancerNS` component and a `LoadBalancingNSCn` AO-connector on the Proxy host. It declares an alternative `PublishingServer` host and puts an additional `NewspaperService` instance on it.

The load balancing connector uses the aspect-oriented interception mechanism to avoid making invasive changes to existing components [24]. The connector ensures the load balancer intercepts all calls of clients to the `NewspaperService` component. Based on their load and availability, it delegates these calls to one of two `NewspaperService` instances.

The Accounting and Personalization services now serve both `NewspaperService` instances. To avoid that these services become the bottleneck, Accounting-Service and PersonalizationService need to be load balanced too. This is done

```

1 abstract application LoadBalancedNewspaperApp refines BasicNewspaperApp {
2   host Proxy;
3   LoadBalancedNS    lbNS      on Proxy;
4   LoadBalancedNSCn  lbNSConn  on Proxy;
5
6   host PublishingServerAlt2;
7   NewspaperService  nsAlt2     on PublishingServerAlt2;
8 }

```

Listing 8. The LoadBalancedNewspaperApp application description, refining the Basic-NewspaperApp application with hosts and instances.

```

1 abstract application LB_AccountedNewspaperApp refines AccountedNewspaperApp,
2   LoadBalancedNewspaperApp {
3   host AccountingServerAlt2;
4   AccountingService  accsAlt2    on AccountingServerAlt2;
5   NewspaperAccountingCnAlt1  accnConn  on PublishingServer;
6   NewspaperAccountingCnAlt2  accnConnAlt2 on PublishingServerAlt2;
7
8   connector NewspaperAccountingCnAlt1 refines NewspaperAccountingCn {
9     ao-composition ServiceUsage refines NewspaperAccountingCn.ServiceUsage {
10      pointcut { callee { override instance: ns; } }
11      advice { instance: accs; }
12    }
13  }
14  connector NewspaperAccountingCnAlt2 refines NewspaperAccountingCn {
15    ao-composition ServiceUsage refines NewspaperAccountingCn.ServiceUsage {
16      pointcut { callee { override instance: nsAlt2; } }
17      advice { instance: accsAlt2; }
18    }
19  }
20 }

```

Listing 9. The LB_AccountedNewspaperApp application description, refining Accounted-NewspaperApp and LoadBalancedNewspaperApp with hosts, instances and connector refinements for load balancing.

separately for each service in the LB_AccountedNewspaperApp and LB_Personalized-NewspaperApp application descriptions. As both descriptions are similar, we explain only the one of Accounting.

Listing 9 shows LB_AccountedNewspaperApp refining the AccountedNewspaperApp and LoadBalancedNewspaperApp applications. It defines an additional host and puts a new AccountingService instance on it. Because the Accounting service is triggered by means of an AO-composition, that composition needs to be refined to have each Accounting instance (lines 11 and 17) intercept calls to the corresponding NewspaperService instance (lines 10 and 16). All of these changes can be modularized within a single description by means of MView refinement.

```

abstract application LB_NewspaperApp refines LB_AccountedNewspaperApp,
  LB_PersonalizedNewspaperApp, SecuredNewspaperApp {}

abstract application NP_NewspaperApp refines AccountedNewspaperApp, SecuredNewspaperApp {}

```

Listing 10. The LB_NewspaperApp and NP_NewspaperApp declarations, combining three and two other apps through refinement, respectively.

```

1 application NYTimes refines BasicNewspaperApp {
2   host PublishingServer      is "published.nytimes.com";
3   host UserManagementServer is "customers.nytimes.com";
4   host ContentManagementServer is "data.nytimes.com";
5   host NewsDeskServer       is "workers.nytimes.com";
6   host ClientTerminal       is "*";
7   host Intranet             is "/*.private.nytimes.com";
8 }
9
10 application CNN refines BasicNewspaperApp {
11   host PublishingServer      is "news.cnn.com";
12   // etc.
13 }

```

Listing 11. Application deployment descriptions for the `BasicNewspaperApp` in the CNN and NYTimes environments

Finally, Listing 10 shows how `LB_NewspaperApp` combines the application descriptions `LB_AccountedNewspaperApp`, `LB_PersonalizedNewspaperApp` and `SecuredNewspaperApp` using refinement.

(1b) Excluding Personalization. While the load balancing variation is about including an additional feature, the personalization variation deals with excluding the Personalization service from the Newspaper application. Personalization is an expensive service in terms of system load. Deploying a variation of the application without this feature may be desirable in particular cases. Furthermore, the Personalization service is also unsuited if a publisher wants to roll out a statically designed site layout.

Creating an application description that does not include Personalization is straightforward. Instead of refining all three parent applications, like in Listing 6 (line 1), the `PersonalizedNewspaperApp` parent is not included in the declaration. The resulting application description, called `NP_NewspaperApp`, is shown in Listing 10.

(2a) Changes in the Environment. Deploying distributed systems of a certain size and complexity (like the e-Media system) is a non-trivial task. The choice for a certain deployment configuration has an impact on the quality and execution of the system. Consequently, this configuration must be easy to adapt as the deployment environment is often subject to change. These changes range from replacing a single host to deploying in an entirely different environment at another organization. The e-Media case study defines two different deployment environments, CNN and NYTimes. Listing 11 shows the deployment configuration of the `BasicNewspaperApp` in both environments.

As shown in Figure 6, the CNN and NYTimes descriptions are reused, by means of refinement, in a total of eight application variants. The `NYT_NP_NewspaperApp` variant is shown in Listing 12. It represents the non-personalized Newspaper application that is deployable at the NYTimes environment. It refines the `NP_NewspaperApp` description from Listing 10 with additional host definitions and a refinement of the `NewspaperSecurityCn` connector. The hosts definitions are not part of the NYTimes description because they are declared as a part of the Accounting and Authen-

```

1 application NYT_NP_NewspaperApp refines NP_NewspaperApp, NYTimes {
2   host AccountingServer is "accounting.nytimes.com";
3   host AuthServer is "auth.nytimes.com";
4
5   NewspaperSecurityCn secnCn on PublishingServer;
6
7   connector NewspaperSecurityCn refines SecuredNewspaperApp.NewspaperSecurityCn {
8     ao-composition ServiceUsage refines
7       SecuredNewspaperApp.NewspaperSecurityCn.ServiceUsage {
9       pointcut { caller { merge host: ! Intranet; } }
10    }
11  }
12 }

```

Listing 12. The NYT_NP_NewspaperApp description, defining additional hosts and refining the Authentication connector.

```

1 abstract application StagedNewspaperApp {
2   host StagingServer;
3   NewspaperService stagingNS on StagingServer;
4
5   abstract connector StagedNewspaperStemCn {
6     abstract ao-composition StagedStemComposition {
7       pointcut { callee { merge host: ! StagingServer; } }
8     }
9   }
10 }

```

Listing 13. The StagedNewspaperApp description, defining an additional host, a NewspaperService instance and the stem staging connector.

tication services and do not belong to the basic Newspaper application. The connector refinement is required to limit Authentication with the Newspaper service to hosts outside of the intranet.

(2b) Staging Environment. The staging environment variation involves extending the deployment configuration with an additional `NewspaperService` that is used as an internal test setup for verifying content and website layout. This is similar to the staging solution we explained in Listing 6, but it is applied to the CNN and NYTimes environments. Because staging is now applied to four application descriptions instead of one, the common `NewspaperService` instantiation and connector definition (called `StagedNewspaperStemCn`) are now separated out into `StagedNewspaperApp`.

To apply staging to an application, we employ *multiple refinement* of the connector and the AO-composition as shown in Listing 14. This comes down to merging both pointcuts, where the “*exclude StagingServer*” host-condition (Listing 13, line 7) is added to the pointcut that is defined in `NewspaperAccountingCn`.

6.2 Modularity and Variability Evaluation

The first part of this evaluation uses the extended e-Media system (with all eight variants) in a comparison of MView, in terms of architect description effort, to two techniques without refinement (*Variability*). To estimate the effort, we use

```

1 application NYT_NP_StagedNewspaperApp refines StagedNewspaperApp, NYT_NP_NewspaperApp {
2   host StagingServer is "staging.intranet.nytimes.com";
3   StagedNpAccnCn accnConn on PublishingServer;
4
5   connector StagedNpAccnCn refines
6     NewspaperAccountingCn, StagedNewspaperStemCn {
7     ao-composition ServiceUsage refines
8       NewspaperAccountingCn.ServiceUsage, StagedStemComposition {}
9   }
10 }

```

Listing 14. The NYT_NP_StagedNewspaperApp application, applying staging to the non-personalized NYTimes Newspaper application.

the Lines of Code metric (LOC) in an absolute as well as a relative comparison. The second part of this evaluation compares how MView and the two other techniques deal with the ripple effects of introducing changes into the architecture descriptions (*Modularity*).

The two other techniques to which we compare MView are:

- (a) **Import.** A technique that supports the import of unchanged specifications between two description 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 to some degree is Fractal-ADL [36].
- (b) **Flatten.** A technique of manual copy and local adjustment. Architects put all their descriptions, belonging to a single view in the process (e.g. Module), into a single description artifact. Instead of refinement, architects in later views manually copy the specification before adding the necessary changes. Approaches that apply this technique are DAOP-ADL, AO-ADL, and AspectualACME [38, 16, 37].

We use the MViewADL syntax when applying *Import* and *Flatten*, to avoid representational differences from skewing the measurements. We only count the lines of code of the resulting description, after adjustment. A *description artifact* is the physical counterpart (e.g. a file) of an architectural description (e.g. a component, a connector, an application). We will often shorten *description artifact* to descriptor, or artifact in this evaluation.

The Impact of Variability on the Description Effort of Architects.

The results of this evaluation² are presented in Figure 7. To realize all eight variants, the total lines of description per technique, in the order as presented on the left graph, are 711, 1173, and 2257 LOC. This amounts to MView having, respectively, 39% and 69% fewer LOC than *Import* and *Flatten*. The substantial difference in size can be attributed to the potential for reuse in MView vs. *Import* and *Flatten*, with this last technique demonstrating no reuse at all.

² More detailed results are presented in Appendix A.1, A.2, and A.3.

Fig. 7. Architect effort in terms of absolute (*left*) and relative (*right*) description size (LOC).

The Module view descriptions consist of a large common part of 264 LOC—shared among techniques—that defines the interfaces and components. The remaining 107 (15%), 140 (11%), and 125 (5%) LOC, for MView, *Import*, and *Flatten* respectively, are concerned with the description of the basic Module-level connectors. This is a relatively big part for MView, because it is in these connectors that the reusable bits are defined (e.g. Listing 2, 4). This allows later descriptions to be defined as a delta. However, because of internal reuse, MView can still express these in fewer LOC in comparison to the other techniques.

It is the job of a deployer to refine the descriptions with host allocations and deployment-level connector refinements. In the case of the e-Media system, deploying the two application variants (load balanced and non-personalized) in two environments (CNN and NYTimes) in a staged and non-staged setup. The involved effort is considerably less with MView: 174 vs. 476 and 1574 LOC. The relative effort for MView is 25% vs. 41% and 70% for *Import* and *Flatten* respectively. *Flatten* requires the most effort, as the deployers need to copy the entire component-and-connector view application description eight times, 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 entire copies of only a limited number of connector descriptions. MView performs best, as refinement enables the changes to connectors to be done in terms of the smaller delta. Both MView and *Import* require a total of 13 connectors, while *Flatten* needs 44 similar, but separate, connectors.

A similar conclusion applies to the application assembly architects in the Component-and-connector view: 166 LOC for MView vs 293 LOC and 294 LOC for the other techniques. The relative effort is respectively 23%, vs 25% and

Fig. 8. Change ripple effects: the number of affected descriptors (*left*) and the amount of description code in LOC across these affected descriptors (*right*).

13%, for *Import* and *Flatten* respectively. Because the assemblers that use either *MVView* or *Import* build modular descriptions for each of the additional Newspaper services, many application- and connector descriptions are produced. This explains the bad result of *Import*, as it cannot reuse a previous connector during refinement. Also, because *Flatten* cannot benefit from the variability of such a structure, assemblers applying this technique need to manually specify the two application variants—a smaller effort that yields a less flexible design.

The performance of the *Flatten* approach can be improved by applying `#ifdef`-style C preprocessor macros [22, 23]. This enables reuse of shared parts of two or more descriptors by unifying them into a single descriptor and annotating the variations. While we observed reuse of descriptions when the number of variations grew, we also noticed an increase in the complexity of the unified descriptor.

Modularity in the Change Scenarios. In this part of the evaluation we apply a number of change scenarios to the existing extended e-Media system architecture. A change scenario describes a modification to the architecture—the result of a change in the requirements—and describes what the consequences are of that modification across the architecture (ripple effects). We have selected three distinct scenarios that belong to different views. These scenarios follow from a focus shift in the publishing industry towards a broader kind of content delivery that can even include third party services.

1. **A Module-view change** where the `NewsRemote` interface is generalized to reflect a change from fetching articles to fetching arbitrary content items.

2. **A Component-and-connector-view change** where the `NewspaperService` component is replaced with a more general `ContentService` component.
3. **A Deployment-view change**, for the CNN deployment, where the `AccountingServer` host is allocated to the same physical host as the `PublishingServer`, instead of a dedicated one.

Figure 8 shows the resulting ripple effects caused by the three scenarios in terms of the number of affected descriptors and the amount of description code (in LOC) in these descriptors. The modularity of MView is able to contain the impact of these modifications better, which is clearly visible in both graphs³.

In scenario 1, MView is able to contain the interface modification within 2 descriptors (5%) and 39 lines of affected description (5%). Because this change affects an essential pointcut description in the design, *Import* and *Flatten* fare much worse (16 and 13 descriptors, amounting to 753 and 1951 LOC), as they are unable to reuse previous connector definitions. The description code that is located in the affected descriptors amounts to 64% and 86% of the total code, for *Import* and *Flatten*, respectively.

In scenario 2, the replacement of an important component affects 5 descriptors (12%) and 65 lines of description (9%). This is in contrast with *Import* and *Flatten*, where 10 (25%) and 11 (39%) descriptors, and, 437 (37%) and 1886 LOC (83%) are affected, respectively. The impact of this change is large for the *Import* case, because this component is used to constrain a pointcut in parts of the design. In addition, it is the duplicated instantiations and allocations of the component in several application variants and deployments that further worsen *Flatten*'s results.

The changes in scenarios one and two can be considered quite invasive. They involve, respectively, modifying an interface and substituting a component throughout the entire architecture. The third scenario, modifying the deployment, is more in line with how the description of an architecture is generally expected to change in the short term. This non-invasiveness is noticeable from the number of descriptors that are affected for scenario 3, at least for *Import* and *Flatten*. MView and *Import* contain the modification in 4 descriptors (10%) and 80 (11%) and 160 LOC (13%), respectively. *Flatten*, again, scores worst of the three with 8 descriptors (29%) and 1574 LOC (70%). This result would be better for MView and *Import*, if the host allocations such as that of `AccountingServer` in Listing 12, would have been defined in separate descriptors per deployment variation.

In conclusion. The results of both parts of this evaluation are from an architecture that describes just eight variations. Consider what this means when this number reaches twenty or thirty. Expressive ADLs and complementary tool support are necessary to manage this kind of variability and evolution of software architectures.

³ More detailed results are presented in Appendix A.4.

6.3 Revisiting the Requirements

Both the MView approach and MViewADL are built on techniques that satisfy all requirements from Section 3.

MView uses a multi-view approach to allow different architects to each specify their contributions on the architecture in a separate model of the view that matches their expertise. MViewADL currently supports three important views in distributed systems design: Module, Component-and-connector, and Deployment.

The multiple models (of varying views) are combined into a single consistent description using the MView refinement relation. In addition, all declarations (such as connectors) in MViewADL are open such that their contents can be defined and adapted in different models that are related through MView refinement.

Finally, as Medvidovic and Taylor state: “an elegant architectural model is of limited value unless it can be converted into a running application” [25]. MViewADL is designed to be generic such that it can support multiple relevant middleware technologies, as we illustrated by implementing code generation for both JBoss and Spring (see Section 5).

Discussion. In order to achieve the vision put forward in Figure 1, MViewADL reuses concepts and techniques that are also present in the related work (see Section 3): multi-model and multi-view descriptions, modularization and adaptation of complex compositions [24], and model relations, in particular, the refinement relation.

The challenge with MViewADL was integrating these into a single consistent language model that would achieve our requirements. Take for example *refinement*, which we use throughout the model because it fits naturally with the process of architectural refinement. Integrating it meant figuring out the meaning of refinement for each concept to which it was applied. We quickly learned that *override* alone was not enough when refining, leading to the introduction of the *merge* operation.

Integrating refinement into the model came down to reasoning about modularity and separation of concerns. This ended up having a pronounced effect on the structure of MViewADL, i.e. the form of the *ao-composition*, overriding of *host* declarations as an expression of physical deployment, component and connector instantiation, etc.

On the other hand, there is still room for the language to further evolve. Currently, refinement has complete access to a parent’s description. This may not always be desirable. One way to address this is through accessibility modifiers. Also, while refinement is broadly applicable, the syntax and semantics of the ADL will need to be extended to support additional views. Finally, we are able to support and express refinement between models of different views in MViewADL, because these views have been designed with a common goal: the description of distributed systems. Keeping the language and its models consistent when adding arbitrary, independent views might prove challenging as well.

7 Related Work

The typical AO-ADLs, that have also been considered in the analysis in Section 3: *DAOP-ADL* [38], *AO-ADL* [37], *Fractal-ADL* [36], *AspectLEDA* [27], *AspectualACME* [16], π -*ADL* [29], *Prisma* [35], 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 [44], CAM/-DAOP [38], DADO [43], FAC [36] and Prose [39] 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 [34], AWED [26], ReflexD [41] and DyMAC/MStage [24] 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 support architect 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, 45, 19]), 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 [45].

8 Conclusion

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

This lack of modularity decreases the variability of the architecture in terms of the functionality it can support and deployment environments in which it can execute.

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. MView restores the modularity of complex compositions and supports architecture variability.

Our evaluation of MView in multiple variations of the e-Media case study shows that MView reduces the overall stakeholder effort (in LOC), and the relative effort of assemblers and deployers with respect to the total specification size, in comparison to the less systematic description techniques of *Import* and *Flatten*. In addition, our evaluation shows that MView is able to better contain the ripple effects of changes introduced into the architecture, with respect to the other techniques.

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].

Acknowledgements. This research is partially funded by the Research Fund KU Leuven and the Fund for Scientific Research (FWO) in Flanders.

References

- [1] Mview tool, <http://distrinet.cs.kuleuven.be/software/mview>
- [2] Redhat inc., <http://labs.jboss.com/jbossaop>
- [3] The spring enterprise platform <http://www.springsource.com/products/enterprise>
- [4] Aldrich, J., Sazawal, V., Chambers, C., Notkin, D.: Language support for connector abstractions. In: Object-Oriented Programming (2003)
- [5] Allen, P., Frost, S.: Planning team roles for CBD. Addison-Wesley Longman Publishing Co., Inc. (2001)

- [6] Allen, R.: A Formal Approach to Software Architecture. Ph.D. thesis, Carnegie Mellon, School of Computer Science (January 1997)
- [7] Apel, S., Kästner, C., Leich, T., Saake, G.: Aspect refinement-unifying aop and stepwise refinement. *Journal of Object Technology* 6(9), 13–33 (2007)
- [8] Atkinson, C., Kühne, T.: Aspect-oriented development with stratified frameworks. *IEEE Software* 20(1), 81–89 (2003)
- [9] Bass, L., Clements, P., Kazman, R.: *Software Architecture in Practice*. Addison-Wesley, second edn. (2003)
- [10] Batory, D., Sarvela, J.N., Rauschmayer, A., Member, S., Member, S.: Scaling step-wise refinement. *IEEE Transactions on Software Engineering* 30 (2003)
- [11] Boucké, N.: Composition and Relations of Architectural Models Supported by an Architectural Description Language. Ph.D. thesis (October 2009)
- [12] Boucké, N., Weyns, D., Hilliard, R., Holvoet, T., Helleboogh, A.: Characterizing relations between architectural views. In: LNCS. vol. 5292. Springer (2008)
- [13] Deng, G.: Resolving component deployment & configuration challenges for enterprise dre systems via frameworks & generative techniques. In: *International Conference on Software Engineering*. ACM (2006)
- [14] van Dooren, M., Steegmans, E., Joosen, W.: An object-oriented framework for aspect-oriented languages. In: *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. ACM (March 2012)
- [15] Feiler, P., Lewis, B., Vestal, S., Colbert, E.: An overview of the sae architecture analysis & design language (aadl) standard. In: *Architecture Description Languages*, vol. 176. Springer Boston (2005)
- [16] Garcia, A., Chavez, C., Batista, T., Sant’anna, C., Kulesza, U., Rashid, A., Lucena, C.: On the modular representation of architectural aspects. In: *Software Architecture*, LNCS, vol. 4344. Springer Berlin / Heidelberg (2006)
- [17] Garcia, A.F., Figueiredo, E.M.L., Sant’Anna, C.N., Pinto, M., Fuentes, L.: Representing architectural aspects with a symmetric approach. In: *Early Aspects ’09*. ACM (2009)
- [18] Garlan, D., Monroe, R.T., Wile, D.: Acme: An architecture description interchange language. In: *CASCON’97* (1997)
- [19] Gray, J., Bapty, T., Neema, S., Schmidt, D.C., Gokhale, A., Natarajan, B.: An approach for supporting aspect-oriented domain modeling. In: *International Conference on Generative Programming and Component Engineering*. Springer-Verlag New York, Inc. (2003)
- [20] Grundy, J.: Multi-perspective specification, design and implementation of components using aspects. *International Journal of Software Engineering and Knowledge Engineering* 10(6) (December 2000)
- [21] ISO/IEC: Systems and software engineering - architecture description. ISO/IEC standard, draft D8 (August 2010)
- [22] ISO/IEC: Iso/iec 9899: The c programming language. ISO/IEC approved standards (2011)
- [23] Kästner, C., Apel, S.: Virtual separation of concerns – a second chance for pre-processors (9 2009)
- [24] Lagaisse, B.: A Comprehensive Integration of AOSD and CBSD Concepts in Middleware. Ph.D. thesis, K.U.Leuven (Dec 2009)
- [25] Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering* 26(1) (Jan 2000)
- [26] Navarro, L.D.B., Südholt, M., Vanderperren, W., De Fraine, B., Suvée, D.: Explicitly distributed aop using awed. In: *AOSD’06*. ACM (2006)

- [27] Navasa, A., Pérez-Toledano, M.A., Murillo, J.M.: An adl dealing with aspects at software architecture stage. *Inf. Softw. Technol.* 51(2) (2009)
- [28] Nuseibeh, B., Kramer, J., Finkelstein, A.: Viewpoints: meaningful relationships are difficult! In: *International Conference on Software Engineering*. IEEE Computer Society (2003)
- [29] Oquendo, F.: 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)
- [30] Oquendo, F.: π -arl: an architecture refinement language for formally modelling the stepwise refinement of software architectures. *SIGSOFT Softw. Eng. Notes* 29 (September 2004)
- [31] Othman, O., Balasubramanian, J., Schmidt, D.C.: The design of an adaptive middleware load balancing and monitoring service. In: *Proceedings of the 3rd International Workshop on Self-Adaptive Software*. pp. 205–213. ACM (2003)
- [32] Palma, K., Eterovic, Y., Murillo, J.M.: Extending the rapide adl to specify aspect oriented software architectures. In: *15th International Conference on Software Engineering and Data Engineering*. p. 170. ISCA (2006)
- [33] Parr, T., Quong, R.: Antlr: A predicated (k) parser generator (1995)
- [34] Pawlak, R., Seinturier, L., Duchien, L., Florin, G.: Jac: A flexible solution for aspect-oriented programming in java. In: *Reflection* (2001)
- [35] Pérez, J., Ramos, I., Martínez, J.J., Letelier, P., Navarro, E.: Prisma: Towards quality, aspect oriented and dynamic software architectures. In: *International Conference on Quality Software* (2003)
- [36] Pessemier, N., Seinturier, L., Duchien, L., Coupaye, T.: A component-based and aspect-oriented model for software evolution. *Int. J. Comput. Appl. Technol.* 31(1/2) (2008)
- [37] Pinto, M., Fuentes, L., Troya, J.M.: Specifying aspect-oriented architectures in ao-adl. In: *Information and Software Technology*. Elsevier (2011)
- [38] Pinto, M., Fuentes, L., Troya, J.M.: A dynamic component and aspect-oriented platform. *Computer Journal* 48(4) (2005)
- [39] Popovici, A., Gross, T., Alonso, G.: Dynamic weaving for aspect-oriented programming. In: *AOSD'02*. ACM (2002)
- [40] Stroustrup, B.: *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edn. (2000)
- [41] Tanter, E., Noyé, J.: A versatile kernel for multi-language aop. In: *Generative Programming and Component Engineering*. LNCS, Springer Berlin / Heidelberg (2005)
- [42] Van Landuyt, D., Op de beeck, S., Truyen, E., Verbaeten, P.: Building a digital publishing platform using aosd. In: *LNCS Transactions on Aspect-Oriented Software Development*. vol. 8 (December 2010)
- [43] Wohlstadter, E., Jackson, S., Devanbu, P.T.: Dado: Enhancing middleware to support crosscutting features in distributed, heterogeneous systems. In: *ICSE* (2003)
- [44] Wohlstadter, E., Tai, S., Mikalsen, T., Rouvellou, I., Devanbu, P.: Glueqos: Middleware to sweeten quality-of-service policy interactions. In: *26th International Conference on Software Engineering*. IEEE Computer Society (2004)
- [45] Zhang, C., Gao, D., Jacobsen, H.A.: Generic middleware substrate through middleware. In: *Middleware* (2005)

A Evaluation

A.1 Table with evaluation data for the MView technique

stage	variation	artifact	#loc		#conns	#refines	
Module		Interfaces.mview	150		0	0	
		AccountingComponent.mview	10		0	0	
		AuthenticationComponent.mview	10		0	0	
		ClientComponent.mview	10		0	0	
		ContentManagementComponent.mview	6		0	0	
		LoadBalancingComponent.mview	10		0	0	
		NewsDeskComponent.mview	13		0	0	
		NewspaperComponent.mview	12		0	0	
		PersonalizationComponent.mview	11		0	0	
		SecurityContextComponent.mview	10		0	0	
		UserCredentialsComponent.mview	6		0	0	
		UserManagementComponent.mview	6		0	0	
		UserTrackingComponent.mview	10		0	0	
		LoadBalancingConnector.mview	18		1	0	
		PersonalizedContentConnector.mview	24		1	0	
		ServiceAccountingConnector.mview	13		1	2	
		ServiceSecurityConnector.mview	13		1	2	
		ServiceTrackingConnector.mview	24		1	2	
		ServiceUsageConnector.mview	15		1	0	
		total (connector)	107	15.05%	6	22.22%	6 9.23%
		total (module)	371	52.18%	6	22.22%	6 9.23%
Assembly		LoadBalancedNewspaper.mview	8	1.13%	0	0.00%	1 1.54%
		NewspaperApplication.mview	14	1.97%	0	0.00%	0 0.00%
		AccountedNewspaperApplication.mview	17	2.39%	1	3.70%	3 4.62%
		PersonalizedNewspaperApplication.mview	36	5.06%	2	7.41%	6 9.23%
		SecuredNewspaperApplication.mview	21	2.95%	1	3.70%	3 4.62%
	(variation 1a)	LBAccountedNewspaperApp.mview	34	4.78%	2	7.41%	5 7.69%
		LBNewspaperApp.mview	1	0.14%	0	0.00%	1 1.54%
		LBPersonalizedNewspaperApp.mview	34	4.78%	2	7.41%	5 7.69%
	(variation 1b)	NewspaperNPApp.mview	1	0.14%	0	0.00%	1 1.54%
		total (assembly)	166	23.35%	8	29.63%	25 38.46%
Deployment	(variation 2a)	CNN.mview	8	1.13%	0	0.00%	1 1.54%
		NYT.mview	8	1.13%	0	0.00%	1 1.54%
	(variation 2b)	CNN_LB_NewspaperApp.mview	24	3.38%	1	3.70%	3 4.62%
		NYT_LB_NewspaperApp.mview	24	3.38%	1	3.70%	3 4.62%
		CNN_NewspaperApp.mview	16	2.25%	1	3.70%	3 4.62%
		NYT_NewspaperApp.mview	16	2.25%	1	3.70%	3 4.62%
	(variation 2b)	CNN_LB_Staged_NewspaperApp.mview	22	3.09%	3	11.11%	7 10.77%
		NYT_LB_Staged_NewspaperApp.mview	22	3.09%	3	11.11%	7 10.77%
		CNN_NP_StagedNewspaperApp.mview	10	1.41%	1	3.70%	3 4.62%
		NYT_NP_StagedNewspaperApp.mview	10	1.41%	1	3.70%	3 4.62%
		StagedNewspaperApp.mview	14	1.97%	1	3.70%	0 0.00%
		total (deployment)	174	24.47%	13	48.15%	34 52.31%
	TOTAL	[artifact count: 39]	711		27		65
Deployment (each possible variant on its own, as if it was developed individually)		CNN_LB_NewspaperApp	568		15		34
		CNN_LB_Staged_NewspaperApp	604		19		41
		CNN_NP_NewspaperApp	375		7		17
		CNN_NP_StagedNewspaperApp	338		9		18
		NYT_LB_NewspaperApp	467		15		34
		NYT_LB_Staged_NewspaperApp	604		19		41
		NYT_NP_NewspaperApp	375		7		17
		NYT_NP_StagedNewspaperApp	399		9		20
		average	466.25		12.5		27.75
Deployer Effort (the total effort when looking only at the deployer architects)		Overall	174	24.47%	13	48.15%	34 52.31%
		CNN_LB_NewspaperApp	32	4.50%	1	3.70%	4 6.15%
		CNN_LB_Staged_NewspaperApp	68	9.56%	5	18.52%	11 16.92%
		CNN_NP_NewspaperApp	24	3.38%	1	3.70%	4 6.15%
		CNN_NP_StagedNewspaperApp	48	6.75%	3	11.11%	7 10.77%
		NYT_LB_NewspaperApp	32	4.50%	1	3.70%	4 6.15%
		NYT_LB_Staged_NewspaperApp	68	9.56%	5	18.52%	11 16.92%
		NYT_NP_NewspaperApp	24	3.38%	1	3.70%	4 6.15%
		NYT_NP_StagedNewspaperApp	48	6.75%	3	11.11%	7 10.77%

A.2 Table with evaluation data for the *Import* technique.

stage	variation	artifact	#loc		#conns	#imports	
Module		Interfaces.mview	150		0	0	
		AccountingComponent.mview	10		0	0	
		AuthenticationComponent.mview	10		0	0	
		ClientComponent.mview	10		0	0	
		ContentManagementComponent.mview	6		0	0	
		LoadBalancingComponent.mview	10		0	0	
		NewsDeskComponent.mview	13		0	0	
		NewspaperComponent.mview	12		0	0	
		PersonalizationComponent.mview	11		0	0	
		SecurityContextComponent.mview	10		0	0	
		UserCredentialsComponent.mview	6		0	0	
		UserManagementComponent.mview	6		0	0	
		UserTrackingComponent.mview	10		0	0	
		LoadBalancingConnector.mview	18		1	0	
		PersonalizedContentConnector.mview	24		1	0	
		ServiceAccountingConnector.mview	24		1	1	
		ServiceSecurityConnector.mview	24		1	1	
		ServiceTrackingConnector.mview	35		1	1	
		ServiceUsageConnector.mview	15		1	0	
		total (connector)	140	11.94%	6	22.22%	3
		total (module)	404	34.44%	6	22.22%	3
Assembly		LoadBalancedNewspaper.mview	8	0.68%	0	0.00%	1
		NewspaperApplication.mview	14	1.19%	0	0.00%	0
		AccountedNewspaperApplication.mview	33	2.81%	1	3.70%	1
		PersonalizedNewspaperApplication.mview	75	6.39%	2	7.41%	1
		SecuredNewspaperApplication.mview	37	3.15%	1	3.70%	1
	(variation 1a)	LBAccountedNewspaperApp.mview	64	5.46%	2	7.41%	1
		LBNewspaperApp.mview	1	0.09%	0	0.00%	1
		LBPersonalizedNewspaperApp.mview	60	5.12%	2	7.41%	1
	(variation 1b)	NewspaperNPApp.mview	1	0.09%	0	0.00%	1
		total (assembly)	293	24.98%	8	29.63%	8
Deployment	(variation 2a)	CNN.mview	8	0.68%	0	0.00%	1
		NYT.mview	8	0.68%	0	0.00%	1
	(variation 2b)	CNN_LB_NewspaperApp.mview	44	3.75%	1	3.70%	1
		NYT_LB_NewspaperApp.mview	44	3.75%	1	3.70%	1
		CNN_NewspaperApp.mview	36	3.07%	1	3.70%	1
		NYT_NewspaperApp.mview	36	3.07%	1	3.70%	1
	(variation 2b)	CNN_LB_Staged_NewspaperApp.mview	111	9.46%	3	11.11%	1
		NYT_LB_Staged_NewspaperApp.mview	111	9.46%	3	11.11%	1
		CNN_NP_StagedNewspaperApp.mview	32	2.73%	1	3.70%	1
		NYT_NP_StagedNewspaperApp.mview	32	2.73%	1	3.70%	1
		StagedNewspaperApp.mview	14	1.19%	1	3.70%	0
		total (deployment)	476	40.58%	13	48.15%	10
	TOTAL	[artifact count: 39]	1173		27		21
Deployment (each possible variant on its own, as if it was developed individually)		CNN_LB_NewspaperApp	748		15		12
		CNN_LB_Staged_NewspaperApp	873		19		13
		CNN_NP_NewspaperApp	460		7		8
		CNN_NP_StagedNewspaperApp	412		9		10
		NYT_LB_NewspaperApp	614		15		15
		NYT_LB_Staged_NewspaperApp	873		19		13
		NYT_NP_NewspaperApp	460		7		8
		NYT_NP_StagedNewspaperApp	506		9		9
		average	618.2		12.5		11
Deployer Effort (the total effort when looking only at the deployer architects)		Overall	476	40.58%	13	48.15%	10
		CNN_LB_NewspaperApp	52	4.43%	1	3.70%	2
		CNN_LB_Staged_NewspaperApp	177	15.09%	5	18.52%	3
		CNN_NP_NewspaperApp	44	3.75%	1	3.70%	2
		CNN_NP_StagedNewspaperApp	90	7.67%	3	11.11%	3
		NYT_LB_NewspaperApp	52	4.43%	1	3.70%	2
		NYT_LB_Staged_NewspaperApp	177	15.09%	5	18.52%	3
		NYT_NP_NewspaperApp	44	3.75%	1	3.70%	2
		NYT_NP_StagedNewspaperApp	90	7.67%	3	11.11%	3

A.3 Table with evaluation data for the *Flatten* technique.

stage	variation	artifact	#loc		#conns	
Module		Interfaces.mview	150		0	
		AccountingComponent.mview	10		0	
		AuthenticationComponent.mview	10		0	
		ClientComponent.mview	10		0	
		ContentManagementComponent.mview	6		0	
		LoadBalancingComponent.mview	10		0	
		NewsDeskComponent.mview	13		0	
		NewspaperComponent.mview	12		0	
		PersonalizationComponent.mview	11		0	
		SecurityContextComponent.mview	10		0	
		UserCredentialsComponent.mview	6		0	
		UserManagementComponent.mview	6		0	
		UserTrackingComponent.mview	10		0	
		LoadBalancingConnector.mview	18	0.80%	1	1.75%
		PersonalizedContentConnector.mview	24	1.06%	1	1.75%
		ServiceAccountingConnector.mview	24	1.06%	1	1.75%
		ServiceSecurityConnector.mview	24	1.06%	1	1.75%
		ServiceTrackingConnector.mview	35	1.55%	1	1.75%
		total (connector)	125	5.54%	5	8.77%
		total (module)	389	17.24%	5	8.77%
Assembly		LB_NewspaperApp	217	9.61%	6	10.53%
		NP_NewspaperApp	77	3.41%	2	3.51%
		total (assembly)	294	13.03%	8	14.04%
Deployment		CNN_LB_NewspaperApp.mview	237	10.50%	7	12.28%
		CNN_LB_Staged_NewspaperApp.mview	245	10.86%	7	12.28%
		CNN_NP_NewspaperApp.mview	149	6.60%	4	7.02%
		CNN_NP_StagedNewspaperApp.mview	156	6.91%	4	7.02%
		NYT_LB_NewspaperApp.mview	237	10.50%	7	12.28%
		NYT_LB_Staged_NewspaperApp.mview	245	10.86%	7	12.28%
		NYT_NP_NewspaperApp.mview	149	6.60%	4	7.02%
		NYT_NP_StagedNewspaperApp.mview	156	6.91%	4	7.02%
		total (deployment)	1574	69.74%	44	77.19%
	TOTAL	[artifact count: 28]	2257		57	
Deployment (each possible variant on its own, as if it was developed individually)		CNN_LB_NewspaperApp	626		12	
		CNN_LB_Staged_NewspaperApp	634		12	
		CNN_NP_NewspaperApp	538		9	
		CNN_NP_StagedNewspaperApp	545		9	
		NYT_LB_NewspaperApp	626		12	
		NYT_LB_Staged_NewspaperApp	634		12	
		NYT_NP_NewspaperApp	538		9	
		NYT_NP_StagedNewspaperApp	545		9	
		average	585.7		10.5	

A.4 Table with evaluation data for the change scenarios.

change	ripple effect in technique					
	<i>MView</i>		<i>Import</i>		<i>Flatten</i>	
	artifact	loc	#artifacts	loc	#artifacts	loc
NewsRemote	ServiceUsageConnector	15	ServiceUsageConnector	15	ServiceTrackingConnector	35
	ServiceTrackingConnector	24	ServiceTrackingConnector	35	ServiceSecurityConnector	24
			ServiceSecurityConnector	24	ServiceAccountingConnector	24
			ServiceAccountingConnector	24	LB_NewspaperApp	217
			SecuredNewspaperApp	37	NP_NewspaperApp	77
			PersonalizedNewspaperApp	75	CNN_LB_NewspaperApp.mview	237
			AccountedNewspaperApp	33	CNN_LB_StagedNewspaperApp	245
			LB_AccountedNewspaperApp	64	CNN_NP_NewspaperApp	149
			CNN_LBNewspaperApp	44	CNN_NP_StagedNewspaperApp	156
			NYT_LBNewspaperApp	44	NYT_LB_NewspaperApp	237
			CNN_NP_NewspaperApp	36	NYT_LB_StagedNewspaperApp	245
			NYT_NP_NewspaperApp	36	NYT_NP_NewspaperApp	149
			CNN_LB_StagedNewspaperApp	111	NYT_NP_StagedNewspaperApp	156
			NYT_LB_StagedNewspaperApp	111		
			CNN_NP_StagedNewspaperApp	32		
			NYT_NP_StagedNewspaperApp	32		
total	2	39	16	753	13	1951
NewspaperService	AccountedNewspaperApplication	11	AccountedNewspaperApplication	33	LB_NewspaperApp	217
	StagedNewspaperApp	14	StagedNewspaperApp	14	NP_NewspaperApp	77
	LoadBalancedApp	8	LoadBalancedApp	8	CNN_LB_NewspaperApp.mview	237
	NewspaperApplication	14	NewspaperApplication	14	CNN_LB_StagedNewspaperApp	245
	LoadBalancingConnector	18	LoadBalancingConnector	18	CNN_NP_NewspaperApp	149
			LBAccountedNewspaperApp	64	CNN_NP_StagedNewspaperApp	156
			CNN_LB_StagedNewspaperApp	111	NYT_LB_NewspaperApp	237
			NYT_LB_StagedNewspaperApp	111	NYT_LB_StagedNewspaperApp	245
			CNN_NP_StagedNewspaperApp	32	NYT_NP_NewspaperApp	149
			NYT_NP_StagedNewspaperApp	32	NYT_NP_StagedNewspaperApp	156
					LoadBalancingConnector	18
total	5	65	10	437	11	1886
AccountingServer	CNN_LB_NewspaperApp	24	CNN_LB_NewspaperApp	44	CNN_LB_NewspaperApp.mview	237
	CNN_NP_NewspaperApp	16	CNN_NP_NewspaperApp	36	CNN_LB_StagedNewspaperApp	245
	NYT_LB_NewspaperApp	24	NYT_LB_NewspaperApp	44	CNN_NP_NewspaperApp	149
	NYT_NP_NewspaperApp	16	NYT_NewspaperApp	36	CNN_NP_StagedNewspaperApp	156
					NYT_LB_NewspaperApp	237
					NYT_LB_StagedNewspaperApp	245
					NYT_NP_NewspaperApp	149
					NYT_NP_StagedNewspaperApp	156
total	4	80	4	160	8	1574