Separation of Concerns in Modeling Distributed Component-based Architectures

Raphaël Marvie Philippe Merle Jean-Marc Geib

Laboratoire d'Informatique Fondamentale de Lille
UPRESA 8022 CNRS
Université des Sciences et Technologies de Lille
Bâtiment M3 – UFR d'IEEA
59655 Villeneuve d'Ascq – France
E-mail: {marvie,merle,geib}@lifl.fr

Abstract

Building component-based distributed applications is a complex task involving a set of cooperating actors like architects, developers, transactions or persistency specialists. For more than ten years, the Object Management Group (OMG) defines open standards to build interoperable distributed applications. First, the Common Object Request Broker Architecture (CORBA) introduced interoperability between heterogeneous distributed objects: An object oriented middleware. Now, the Model Driven Architecture (MDA) introduces interoperability between heterogeneous models: A model oriented middleware. In this context, we advocate the separation of concerns in order to structure the modeling and meta modeling of enterprise distributed component architectures. In the meantime, design related knowledge is most often lost at runtime. Nevertheless, this knowledge could be important to reify architectures of applications at runtime and to support their administration and reconfiguration. Thus, we intend to support separation of concerns from design to runtime of applications, using a meta data repository centric approach. This paper discusses our proposal, CODeX, to structure the definition of meta models in order to offer dedicated points of view of a model to each of the actors of the software engineering process, from architects to application administrators.

1 Introduction

Nowadays, building distributed component-based applications involves several collaborating actors like architects, developers, technical specialists, or administrators. Each of these actors has distinct concerns thus requires different means to perform their task. In the meantime, the definition of standards is required to support the collaboration: Developers need to understand and use the definitions produced by architects. The motivation of our proposal, CODeX, is to support the software engineering process regarding the separation of these concerns in order to allow co-design of applications.

Component models are a well accepted solution to build distributed applications. They provide a set of means to design, develop, and assemble the various pieces of applications. Enterprise Java Beans (EJBs) [12] from Sun Microsystems, the CORBA Component Model (CCM) [18] from the OMG and .Net [26] from Microsoft are the major industrial technologies to build distributed applications and their use is increasing. But, component technologies lack, most of the time, means to properly define architectures. In the meantime, Architecture Definition Languages (ADLs) [13] are a well accepted solution to define distributed application architectures. However, each ADL addresses a limited set of concerns that range from formal specification for behavior validation purposes [9, 1] to application configuration for code generation and to some point as support to the deployment process [14, 2]. This leads to three remarks. First, there is no ADL that addresses all the concerns related to a software engineering process, and very few to be extensible. Second, concerns addressed by ADLs are all mixed up at a same syntactical level. Thus, separation of concerns is not encouraged and co-design capabilities are reduced. Finally, architecture definitions are never reified at runtime, loosing all the information related to the architecture that would be useful for administration purpose for example. CODeX tends to separate concerns in the definition and use of software architectures from design to runtime.

The multiplicity of component models and the strong link between an application and the technological component model used to developed it have an influence on the

future of enterprise solutions. Once chosen, the component model cannot be changed even if the context or the requirements evolve. To address this drawback, the OMG currently shifts from promoting the Common Object Request Broker Architecture (CORBA) [20] to the Model Driven Architecture (MDA) [3, 16]. The later promoting the use of models to generate the implementation of distributed component-based applications. Once defined, the model of applications could be mapped to different component technologies. Following this approach, CODeX is based on the use of meta modeling techniques to support the definition and use of software architectures.

In this context, we do not intend to define a universal extensible ADL that would meet all the potential requirements. Following the MDA, CODeX is a framework to structure the definition of ADLs enabling multiple actors to co-design distributed applications. To reach this goal, CODeX relies on the separation of concerns both in the ADL definition and in its use. Moreover, ADL definitions are performed according to application domain specificities. Then, from these definitions, a dedicated ADL and its environment are produced to support the software engineering process of the domain. Finally, the environment generated is intended to be used from design to runtime, meaning application architectures are reified following separation of concerns for its whole life-cycle.

This paper is structured as follows. Section 2 outlines the evolution of using modeling techniques and its actual frontier. Section 3 discusses our decomposition of the software engineering process as well as the three major industrial component models. Section 4 discusses the separation of concerns in architecture modeling using CODeX. Section 5 presents the use of meta modeling techniques to define an ADL respecting our separation of concerns. Section 6 presents the environments associated with ADLs that are produced using CODeX. Section 7 presents some related works in the context of architecture definition languages. Finally, section 8 concludes this paper and gives some of our future trends.

2 Models Everywhere

2.1 From Hand Craft to Industrial Production

At the beginning was the hammer and the chisel, all concerns of distributed applications—like communications, transactions, persistency, security, and functional ones—were fully hand written.

The introduction of remote procedure calls, like those used in CORBA, was a first step to reduce the amount of code to be written. From object interfaces defined in OMG Interface Definition Language (IDL), communication skeletons and stubs are automatically generated. Thus, one technical concern, communication between objects, is addressed and it is no more necessary to write this part of object implementations. Nevertheless, this approach does not address other concerns like transactions, persistency, or security.

With the advent of component models, like the CCM, the amount of hand written code is again reduced. Through the use of containers and description of non functional properties, concerns like transactions, persistency, and security are adressed and no more required to be hard coded by hand. Open Software Descriptors (OSD) and Component Implementation Definition Language (CIDL) are new means associated to OMG IDL to support the production of applications. However, addressed concerns are most of the time reduced to those three ones and applications are definitely linked to the component technology used. In the meantime, component technologies are multiple and none of them is a ultimate solution for all the contexts.

Finally, the actual vision of software engineering relies on the use of models. To do so, the OMG advocates the use of the Model Driven Architecture (MDA) [3, 16]. The use of models is structured in a two level vision. Platform Independant Models (PIMs) are used to define application models unregarding component technologies that will be used to implement them. Then, PIMs are mapped to Platform Specific Models (PSMs) that represent the application model in the context of a component technology. This later is the basis of the application implementation generation. The Unified Modeling Language (UML) [21] and the Meta Object Facility (MOF) [17] are the new means to support the definition of application models. Hammers and chisels are far from us now, and the software engineering process is now fully industrialized.

2.2 New Emerging Challenges

Separation of concerns The model oriented approach fulfils most of common expectations to produce distributed component-based applications. However, it is now necessary to define methodologies to use models. There is a need to structure PIMs. Separation of Concerns (SoC) is an interesting approach for the structuration already applied in the context of programming languages using Aspect Oriented Programming [6]. Our first intent is to apply SoC in the context of models to structure PIMs and PSMs.

Runtime reification In the meantime, MDA is very similar to CORBA in one aspect that raises a limitation. Using CORBA, the OMG Interface Definition Language is used at design and development time and it most often lost at runtime, the use of the Interface Repository is not very common. Using the MDA, PIMs and PSMs are used at design and development time but seem also

to be lost at runtime, even if MOF repositories that could provide models at runtime exist. Our second intent is to reify PIMs and PSMs at runtime respecting separation of concerns.

Co-design The OMG has specified a set of UML profiles to define application models in a particular context. For example, PIMs will be modeled using the profile for Enterprise Distributed Object Computing (EDOC) [25] or Enterprise Application Integration (EAI) [24]. Similarly, we have specified CODeX in order to define application architectures using separation of concerns and to reify those architectures from design to runtime. The two main objectives are to support the co-design of applications and to support administration and reconfiguration of applications at runtime.

3 Software Engineering

3.1 Actors of the Software Engineering Process

Building distributed component-based applications is a complex task involving several actors that co-design applications [4]. In our context, we have decomposed the software engineering process in six actors that collaborate to define software architectures as depicted in figure 1.

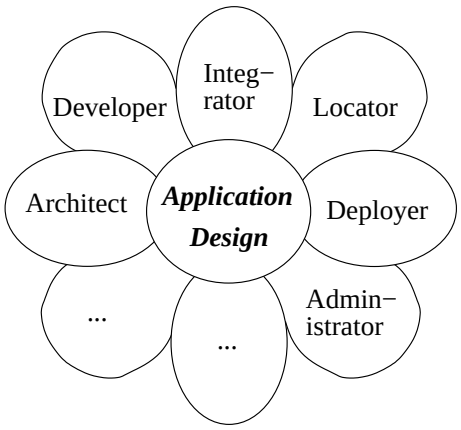

Figure 1. Collaborative Design

Architects define structures of applications, types and assemblies of components, as well as their potential reconfigurations (addition or removal of components and connectors).

Developers realize the functional implementation of component types defined by *architects* that are not already available. They produce binary versions of component types and deliver them as archives.

Integrators define the set of component implementations that can be used, and where they are available. They install archives on archive repositories in order to permit their future download. Then, they create a relationship between the component types defined by *architects* and those archives. This information is added to the application specifications and will be used to perform their deployment.

Locators define the representation of the environment that is used for given applications, the set of hosts and their capabilities, as well as the network connections between hosts. Then, they specify on which host component instances have to run. They contribute to the application specification by enriching *architect* definitions, through the creation of a relationship between components and hosts.

Deployers specify how the deployment of applications has to be performed, and execute it. Reusing the specification produced by *architects* and enriched by *integrators* and *locators* they define / refine the deployment process: They specify the order in which components have to be deployed and guarantee the atomicity of the process (whole the application or nothing has to be deployed).

Administrators supervise and reconfigure running applications. Thus, they need to have access to the whole architecture specification. Moreover, this has to be possible at runtime.

In order to build applications, the various actors are contributing in defining, developing, deploying, or monitoring pieces of software. Other actors related to non functional properties of applications can also be defined introducing separation of concerns related to security, persistence and transaction for example[1]. Component models are a well accepted solution to address this problem of putting the pieces together. The following section discusses the three major industrial component models.

3.2 Component Models

Many component models have been defined in industrial and academic contexts to address specific needs. There is not a particular model that would address any kind of needs, so we have chosen to discuss the three major industrial models that are currently used. This is done in presenting the benefits and limits of those models in the context of the actor taxonomy outlined in the previous section.

Sun's Enterprise Java Beans (EJBs) [12] are server-side components. Applications exist in the context of an application server which can be accessed from remote clients.

[1] But those won't be discussed here.

While component types are defined using Java interfaces, there are no means for *architects* to define architectures as the connectivity between Beans is managed internally (as in objects). Based on these interfaces, *developers* write component implementations using their favorite development environment for Java, and produce Java archives containing both the implementation and an XML description outlining the non functional requirements of the Beans (transaction, persistency, and security). As there is only one programming language and as an application is running on a single host, *integrators* and *locators* roles are very limited, there is no real choice of implementation (only one exists) and all the components are running together. *Deployers* have only to make all the component archives available on the application server and to configure the environment. Component containers, that host the components and manage the non functional properties, are generated according to the XML descriptors of archives. Finally, *administrators* are able to administrate applications according to *ad hoc* tools provided by the application server. There is no knowledge from the design phasis that is available except the interfaces of components through the reflection API of Java.

Microsoft's .Net framework [26] mainly provides a common language runtime (CLR) to run components written using different programming languages. Thus, *developers* can produce component implementation using their favorite language or the one that is the best suited. The various concerns addressed at development time can be found in the byte code produced as meta data, thus available at runtime through reflection. Nevertheless, all these meta data are mixed up. There is no means included in the framework for other actors to perform their job. *Architects* can only define component interfaces if the programming language used by *developers* permit to do so. Moreover, XML assemblies are not as important as they ough to be. .Net is mainly relying on the availability of *ad hoc* tools that would help *deployers* and *administrators* to properly perform their task.

The OMG's CORBA Component Model (CCM) [10, 18, 27], basis of the forthcoming CORBA 3 specification, is a real distributed component model: components of a single application may run on several hosts. Even if being a server-side component model, it may also be used to write client-side components. It provides two distinct means for *architects*: The OMG Interface Definition Language (OMG IDL) to specify component types and an XML DTD to define component assemblies. This provides a basic version of application architectures. As in CORBA 2, component types can be implemented in any language, thus *developers* may choose the best suited one. Moreover, the CCM defines both a packaging standard and a set of XML descriptors to provide information regarding implementations and assemblies. Those packages are used by *integrators* to set the relationship between the application component types and the implementation to be used. As in EJBs, the CCM relies upon containers to run component instances. *Locators* have then to express the properties of such container servers and to set the relationship between component instances and hosts. A basic deployment model is defined. It provides both a set of API and a process for *deployers* to install applications on a network, using their XML descriptors. Finally, *administrators* may find some tools based upon standard CORBA services to supervise applications, but the application architecture, even simple, is not available at runtime as XML descriptors are not reified.

Looking at these three models, actors' concerns are better addressed using the CCM. Even so, one can see first of all that architecture definition is not a major concern of industrial solutions. When means exist, like in the CCM, they are very primitive and very technical. Second, even if various concerns are addressed they are not separated, .Net meta data are completely mixed up once components are compiled. Finally, even if architecture related knowledge exists at design time, like XML descriptors, it is mostly lost at runtime: EJB and CCM only support interface introspection, and .Net provides some meta data. Thus, even if component models are well suited for structuring application development, they lack the specification of application architectures. Architecture Definition Languages (ADLs) are a well accepted means to define software architectures. Next section dicusses how ADLs can be defined in the context of the MDA approach using CODeX.

4 Modeling Architectures and Separation of Concerns

4.1 Overview

CODeX is a framework to structure the definition of ADLs enabling multiple actors to co-design distributed application. To reach this goal, CODeX relies on the separation of concerns both in the ADL definition and in its use. Moreover, ADL definitions are performed according to application domains specificities. Then, from these definitions, a dedicated ADL and its environment are produced to support the software engineering process of the domain. The MDA was primarily defined to produce applications. The idea of CODeX is to apply the MDA to define the meta models of ADLs and to generate their supporting environment.

In order to properly provide means for all the actors mentioned earlier, generated ADL environment are intended to give access to application architecture definitions from design to runtime. In comparison to the Smalltalk system [8], CODeX is expected to be a "living system" where definitions and instances of applications are coexisting. The definitions of architecture elements are not only syntactic, but

reified as objects. Then, instances defined by *architects* are used by other actors and still exist in the system at runtime providing the definition of application architectures. Moreover the separation of concerns is effective not only at design time but also at runtime. To do so, ADL environments are architectured around a meta data repository.

CODeX meta modeling is based on the use of the OMG's Meta Object Facility (MOF) [17]. The MOF specification defines a four levels modeling stack, from the meta meta model level (M3), the MOF itself, to the instance level (M0), where instances of application objects are existing. In the context of CODeX, the use of three levels have been defined. All these three levels are related to application descriptions and not to the application instances themselves (see Figure 2). Looking at the big picture, from meta models to application instances, CODeX has one more level than the MOF has defined. The reason is that CODeX mainly deals with architecture description of applications and not directly with applications themselves. Moreover, CODeX uses the MOF thus the CODeX related definitions begin at the meta model level (M2).

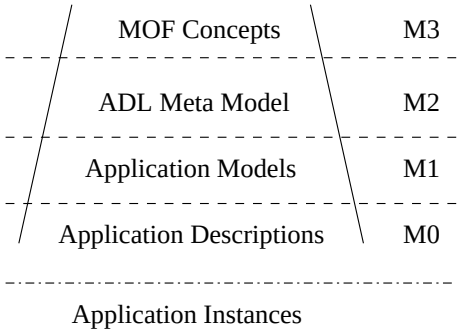

Figure 2. Levels of the CODeX Framework

ADL meta models are defined at the M2 level. This is the level of concept definitions. Application models are defined at the M1 level. These definitions rely on the concepts defined at the M2 level, and are the definitions of application architectures as they would be defined using classic ADLs. Those models can be used to create one or more instances of applications, they are to be compared to classes in Object Oriented languages. The M0 level is not populated of applications instances but of the representation of these instances. For each instance of an application defined at the M1 level, a representation is created. It provides a meta data repository based version of the application architecture that is usable for deployment and administration purposes. Finally, on the side of these three levels, application instances are existing in the context of a particular technology. An interesting aspect of CODeX is that those instances are standard regarding the chosen technology without enhancements nor modifications.

For each instance of the application that is running in the context of a technological model, a meta instance is existing at the M0 level of CODeX, its architectural representation. Moreover, for concepts that are not reified at the application level, like a reference between two component instances, a meta instance exists at the M0 level. Thus, references may be modified, removed or established easily even if the technological model does not provide means to easily do such operations. Finally, at runtime the M0 and M1 levels of CODeX are co-existing in the meta data repository, providing a usable version of the application architecture.

4.2 Approach

The separation of concerns has to be organized for the context of architecture definitions. CODeX deals with this organization at three distinct levels. Figure 3 compares these three levels to common ADLs. In ADLs all the concerns are mixed up at the same syntactical level: the big box on the left of the figure. In CODeX, each concern is defined indepedantly of the others and integrated afterwards: the right part of the figure.

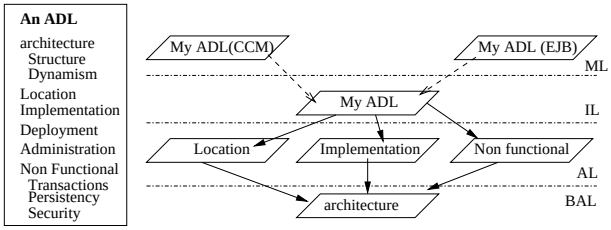

Figure 3. The Three Levels of Abstraction Compared to ADLs

First, the *base architectural level* (BAL) is defined as the heart of the application architectures, their structures. This level is to be used by the *architect* and provides a basis for all the other actors of the software engineering process. It specifies the concepts of the domain to be used in order to build applications: For example the definition of a composite, a component, a connector, a configuration, and the basic architectural operations that can be used on components and composites.

Second, the *annotation level* (AL) is used by actors others than *architects*. A dedicated annotation level is defined for each concern. It enables the refinement of the base architectural level for a specific concern. To do so, the concepts related to the concern are defined, as well as their integration in application architectures. At this level, each concept of the base architectural level is known as it, and the relationship between those concepts and the concepts of the

annotation level are defined. As an example, the annotation level of the *integrator* will specify the concepts like package archive, component implementation, software descriptor, package location, as well as how they are related to the components defined at the base architectural level.

Finally, the *integration level* (IL) integrates together all the concerns defined at the annotation level. The goal is to produce the global version of application architectures without losing the separation of concerns. Then, the global application architecture still can be used from the point of view of a specific concern. This is important regarding the availability of architectures at runtime with the concerns unmixed. Thus, the benefits of separation of concerns are available from design to execution time while it is also possible to have a global vision of the architecture.

Once this last level specified, some mapping have to be defined for the technologic component models that are intended to be used like EJBs or the CCM: the *mapping level* (ML). The three first levels structure PIMs and the mappings are the transformation between PIMs and PSMs.

5 Meta Modeling of ADLs

In order to define ADLs following the separation of concerns, the meta models of the three levels presented in section 4 have to be defined separately. Thus, we are considering three steps in defining the meta model of an ADL: Definition of the base architectural package, definition of each concern meta model, and definition of the concerns integration. To do so, we have chosen to use the Meta Object Facilty (MOF) in order to benefit from its mappings to CORBA to produce meta data repositories.

The MOF allows the definition of meta models using four base concepts: package, to group related definition, class, to define an entity which may contain attributes and operations, association, to define a relationship, and datatype, to define basic or constructed types. As in object oriented context, MOF classes can be in relation through inheritance. In the context of the MOF, packages can be in relation through the use of import but also through inheritance. This means that if package A inherits package B, all the concepts defined in package B are known to A as if defined in A. This last capability is used to define the three levels of CODeX separation of concerns. This section illustrates the definition of an ADL using CODeX.

5.1 Base Architectural Package

The base architectural level is defined first. The core concepts of the application component abstract model have to be specified. In the context of distributed applications, components are currently a technology of choice. Nevertheless, the definition of a component, a connector or a configuration — which are accepted as the three main concepts — may slightly vary from one use to another. Thus, CODeX allows the proper definition of those concepts for a given domain, as well as the basic architecture operations that can be performed on component assemblies like component creation and destruction, and connection establishement and removal. All these concepts are defined in a MOF package.

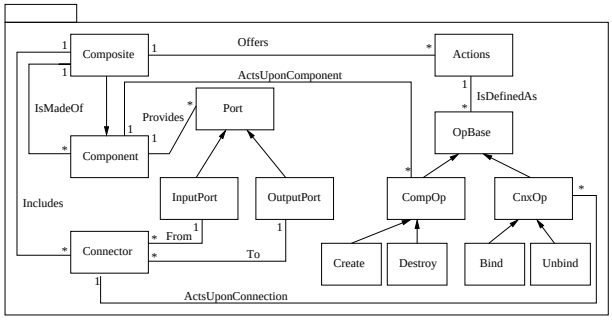

Figure 4. A Base Architectural Package

Figure 4 presents an example of base architectural package defined with CODeX. This package has been defined for experimenting CODeX in the context of the CORBA Component Model (CCM) using our OpenCCM platform[2] [11, 10]. In addition to the concepts existing in the CCM, this package reifies the connections between components and defines *composites* as a means to structure component assemblies. The main goal of this is to be able to act upon assemblies as one would act upon components. Moreover, architectural operations are defined to act on the structure of composites (add / remove components, bind / unbind connections). Those are the basic operations to support the deployment and dynamic reconfiguration of CCM applications using CODeX.

5.2 Concern Packages

Once the base architectural package available, annotation packages are defined. Each concern is separated in two distinct MOF packages. A first one, the base package of a concern, defines the concepts related to the concern, like what is a host or a network connection when defining the meta model of the system infrastructure. A second one, the annotation package of a concern, defines how these concepts will be used to annotate the base architectural level, for example how the relationship between a component and its execution host is specified. This annotation package inherits from the base architectural one to extend its concepts, and imports the base package of the concern to use its concepts. This agrees with our definition of the annotation level

[2] `http://corbaweb.lifl.fr/OpenCCM`

in section 4, and the separation of concerns is respected as each concern meta model is defined independently.

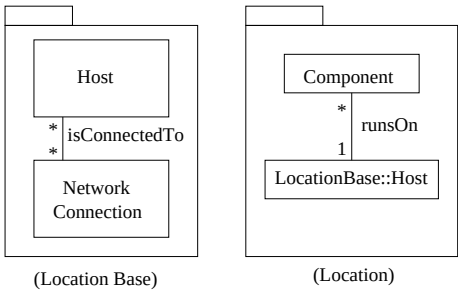

Figure 5. The Location and LocationBase Packages

Figure 5 depicts an example of annotation base and annotation packages, it defines at a high level the *Location* meta model. The two basic concepts are *Host* and *Network Connection*. These two concepts will be used by a *locator* to modelize the execution environment. The location annotation package inherits the base architectural package, thus acquiring all its concepts, only the *Component* concept is represented here. It also imports the location annotation base package, only *LocationBase::Host* is represented as used in our example. In order to create a relation between a component and its host, the only entity defined in the annotation package is a MOF association, named *runsOn* between the inherited concept *Component* and the imported concept *LocationBase::Host*. This represents the technique used to separate concerns in the context of modeling.

5.3 Integration Package

In order to have a complete specification of an ADL for the specific domain, the integration of the various concerns is defined as a last MOF package. Without violating the separation of concerns, this last one defines how all the concerns are coexisting. This last package inherits from all the various annotation packages (see Figure 6). Once its integration package is defined, an ADL is fully defined in the context of CODeX. From this point the MOF mappings can be applied to generate the environment associated to the newly defined ADL.

The result of inheriting annotation packages is a package containing all the concepts defined in the base architectural package, enriched by the associations — with annotation base package concepts — defined in each annotation packages. As an example the concept *Component* will be defined in the integration package as it was in the architectural package, but enriched with associations like *runsOn* coming

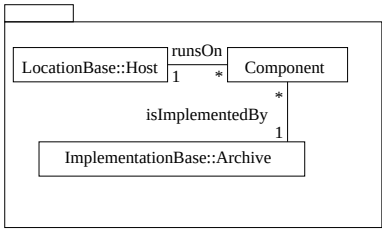

Figure 6. Excerpt of an Integration Package

from the location package (see previous section) and for example *isImplementedBy* (from component implementation package that is not presented in this paper).

5.4 The Big Picture

Figure 7 depicts the big picture of CODeX. It presents the global structuration of ADLs using CODeX. Using the MOF, it represents guidelines to define an ADL. Using UML, a profile provides a set of stereotypes and tagged values that helps ADL designer to perform their task. It summarizes and generalizes the relationships between packages presented in the previous sub-sections.

On the use of package inheritance Package inheritance may not be seen as the most natural approach to implement CODeX separation of concerns. The use of package import and concept inheritance would certainly be used in the context of object oriented technologies. Nevertheless, using the latter option introduces several drawbacks. Firstly, each actor would use different names for the same concepts, for example the architect would use a *component* while a locator would use a *located component* and an integrator would use an *implemented component*. Secondly, in order to integrate all the enriched version of a concept multiple inheritance would have to be used. This may raise problems as two identical attribute names, for example, would break the possibility to use multiple inheritance. Finaly, how would the integration of concept enrichments be named?

The use of package inheritance and concept association is interesting for several reasons. First, according to our wish all the actors use the same concept with the same name. It does not avoid them to enrich this concept using MOF associations between base architectural concepts and base concern concepts. Moreover, there may be as many association defined as required. Second, the integration of all the concern definitions is quite simple as package inheritance of all annotation packages means the integration package contains those definitions. Then, the component defined in the integration package contains both the association to its execution host and the association to its imple-

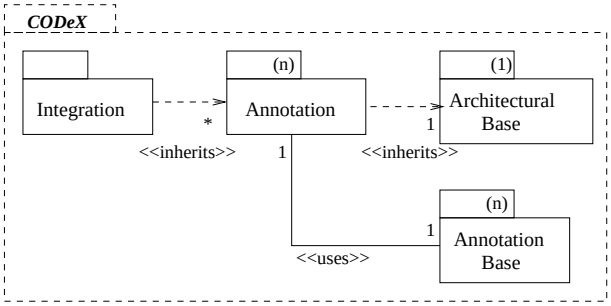

Figure 7. MOF Package Structure of ADL Meta Models

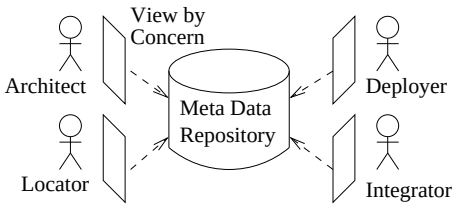

Figure 8. Views by Concerns of the Meta Data Repository

mentation archive. From the actors point of view, a component is always identically defined, there is only the set of associations from / to it that vary from one actor to another. Finaly, beyond the fact that implementing the second approach is easier than implementing package import with concept inheritance, the use of package inheritance and association seems to betterly reify our approach of concept enriching using separation of concerns.

6 CODeX Environment

6.1 Associated Environment

The environment associated with an ADL is architectured around a meta data repository which is intended to contain the definitions of application architectures. Inside this repository, the three levels presented in the previous section are not merged but are available as they were defined. Thus, the same repository is shared amoung the various actors, while each actor may use the information contained in the repository from his own point of view (see Figure 8). All instances reifying the architecture elements of applications are created and 'live' in the repository. These instances could be reached by the various actors of the software engineering process from design to runtime, supporting the co-design of applications.

Using CODeX, *architects* define the structure of applications, the assembly of components, as well as architectural operations in the meta data repository. Once this is done, the various actors may contribute to the definition of the complete architecture also in the repository. Once the components are implemented by *developers*, *integrators* annotate component specifications with the location of their implementation. Meanwhile, *locators* annotate them with their execution host, and *deployers* define the deployment process. When the definition of the architecture is completed, the meta data repository is used to perform the deployment of the application in the system. Then, while the application is running, it can be monitored and controlled by *administrators* using the operations defined in the repository. Thus, the repository allows the dynamic reconfiguration of applications.

6.2 Environment Generation

From a meta model expressed in XML Metadata Interchange (XMI) [23], the MOF defines mappings to generate the OMG IDL interfaces of the meta data repository that is to be used to store instances of the meta model, in our context models of application architectures, as well as the XML DTD specifying how models of application architectures should be written in XML. The means to generate the XMI version of the ADL meta model is not included in CODeX. It may be a simple text editor, a MOF or a UML Case tool able to save meta models as XMI.

In addition to the standards mappings, it is possible to generate the implementation of the meta data repository with the interfaces defined by the MOF specification. It is clear that such repository will not always be fully implementable, generators cannot guess the implementations of all the meta model class operations. But, a base implementation is generated that can be specialized by designers of ADLs or actors like *deployers*. In addition to the repository, a parser, respecting the DTD defined by the MOF mappings, is generated to load application architecture definitions into the repository. This parser is mainly intended to be used for managing application model serialization. Finally, a browser, or a set of browsers, are generated to interact with the repository. All those steps are depicted in Figure 9.

The last step is to define a mapping between the specification of the ADL and the technology that is used for implementing the applications. CODeX first experiments of mappings are realized in the context of the CORBA Component Model. The most important point here is to define the relationship between the instances of applications and their meta instances in the meta data repository, and between some concepts of the application level that are reified in the repository, for example CCM references and their

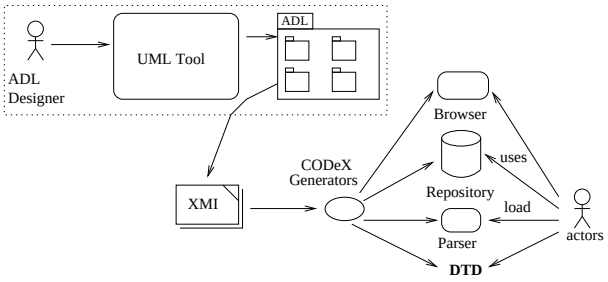

Figure 9. From ADL Definition to its Use

reification as connections in the base architectural package.

At the moment, the repository implementation is fully written in OMG IDLscript [15, 19]. The choice of IDLscript for implementing repositories is driven by the fact that a scripted repository is more flexible than one written in Java for example. Moreover, our jIDLscript[3] implementation offers the same access to CORBA and to Java objects. Thus, such an implementation can be used both for CCM and EJB based applications, CCM being the basis of our first experimentations and EJBs will be the second technology to be used. The implementation of the repository has no effect on the performance of applications. Thus, the choice of a scripting language to realize it is interesting for the flexibility brought without usual drawbacks. The repository can dynamically be enhanced and its behavior modified to match the evolving needs of applications.

6.3 Use of the Environment

Once the environment is produced, it can be used by the various actors to define applications in two ways. First, in the design phasis, actors can use any modeling tool that produces XMI compliant to the generated DTD to produce the model of their application. Then, the model is loaded in the meta data repository using the generated model parser. Second, the most interesting one, actors can directly use the meta data repository with appropriate tools that use it as their data storage. Right now, our OMG IDLscript interpreter is used to interact directly with the repository. Nevertheless, a generic browser or a dedicated tool that is plugged to the repository is a better solution for our repository centric approach. In both cases, the components of the application have to be implemented and the various specifications of concerns finalized. Moreover, application definitions can be finalized using, for example, scripting languages to refine the operations, like deployment ones, contained in the meta data repository.

When the model of the application is completed and the implementation available, the deployment process can be launched. The operations contained in the meta data repository drive this process and create both an instance of the application and a representation of this application in itself. From that point, the repository contains the model and a meta instance of the application, both representing the application that runs. Then, it is possible to administrate the application and make it evolve. For example, if a new end-user of the application appears, its client side can automatically be deployed using actions defined on composites that have been defined for such a task. The client part of the application is a set of components to be connected to the rest of the application which deployment can be automated.

Activities related to administration rely upon the use of the elementary architectural operations defined in the base architectural level: create, destroy, bind, and unbind. Using these basic operations it is possible to define architectural actions that are used to deploy and reconfigure applications. Those actions can be defined both by architects and administrators. Latters can interact dynamically with the repository at runtime to add new actions. Actions are stored in the repository together with their implementations. The evaluation of those actions is performed through the repository: To process an action, an administrator invokes the *eval()* method on the meta object representing the action. The behavior of this evaluation may be modified as the whole repository is implemented in IDLscript. Thus, it is easy to add or replace part of the repository implementation. Without modifying the implementation of existing actions, an administrator can refine the default behavior of an action (s)he has created for new requirements of running applications.

7 Related Works

Lots of ADLs have already been defined [13], each having advantages and disadvantages which may vary from one use to another. From our point of view, each ADL has its own concerns which are limited to the needs or goals of its authors. We have identify several drawbacks that avoid us from reusing existing ADLs as basis of our work.

1. Very few ADLs reify architecture definitions at runtime, implying the lost of architectural vision for administration purposes.

2. All the concerns required by our software engineering process were not met, and very few ADLs support extensibility.

3. Separation of concerns is not supported as all concerns are mixed up at a same syntactical level.

ADLs like Rapide [9] and Wright [1] concerns are related to formal specification, analysis, and to some point

[3] `http://www.lifl.fr/~roos/jidlscipt`

to validation of software architectures. Both are providing means to test the behavior of components and applications. Moreover, they support the definition of application dynamism — changes in the structure of the application, the interconnection of components — and to specify its semantic. Thus, *architects* have proper means to define application architectures. Nevertheless, those ADLs do not provide any means to support code generation, configuration, and deployment of applications. Thus, actors other than *architects* have no means except 'textual information' to perform their task.

In the meantime, ADLs like ACME [7] and xADL [5] provide solutions for the sharing and integration of various software architecture pieces not necessarily defined using the same ADL. Thus *architects* can both define architectures from scratch and reusing existing pieces of applications. A major concern is the extensibility of the ADL both to integrate existing definitions, and to take into account concerns which are not already existing. Those two ADLs are thus customizable to meet special requirements, as an example the expression of QoS may be introduced in ACME using properties. Depending on the tools associated to the ADL extensions, the various actors will have means to express their concerns, and for some of them to be assisted in their task (like *developers* if an extension is written for code generation). However, all the concerns are mixed up at a same syntactic level: Properties expressed as strings for ACME and XML entities for xADL.

Finally, ADLs like C2 [14] and Olan [2] address concerns related to configuration, code generation, and to some point to automated deployment and the administration of distributed applications. Both ADLs permit the separation of definitions related to design from those related to implementation. Thus, they allow *architects* to define abstract architectures, that will be refined / extended to assist *developers* in their task. While C2 provides several frameworks to implement components in C++ and Java, Olan supports the generation of code. Both *integrators* and *locators* find some means to associate implementations to component types and to map a logical architecture to an execution environment. Associated to some tools, code generated using Olan is providing support for both *deployers* and *administrators*. Similarly to most other ADLs, C2 and Olan are not promoting the separation of concerns and are not extensible to introduce easily non foreseen ones. In the meantime, they are not the ADLs to be chosen in order to perform formal verification of application behaviors.

8 Conclusion

Since the standardization of the Unified Modeling Language by the OMG, the use of modeling techniques has evolved. Nowadays, models are not only used as informal specifications of applications but are also a basis of code generation, improving compliance of code to its specification. With the introduction of the Model Driven Architecture, most, if not whole, of the application implementation is generated. In the meantime, it tends to support the reuse of models in different component technologies. Nevertheless, this evolution is not completed. Models still be design time concepts that are very rarely reified at runtime losing a large amount of information related to running applications.

This paper has presented CODeX proposal that tends to address three aspects of building component based applications. First, it has discussed the benefits of runtime reification of application architecture models: support of administration and reconfiguration of applications. Second, it has discusses the separation of concerns in defining application architectures: support of application co-design. Finally, it has presented the use of meta modeling techniques to define both ADLs and their associated environment: definition of dedicated ADL for applicative domains. In all that, CODeX tends to structure both the production of means to define application architectures as well as their use, thus structuring the software engineering process and improving the collaboration of the various actors.

The work presented in this paper is a first step. Actual experiments' first goal was to validate the approach of using meta modeling techniques to define ADLs and associated environnements. We now have to specify some guidelines to define both an ADL and mappings for technical component models. In addition, our experimental framework has to be finalized in order to be widespreadable. We intend to provide some tools to generated dedicated graphical interfaces for concerns that are defined in the ADL instead of only using our OMG IDLscript interpreter. Finally, once responses to the OMG RFP regarding configuration and deployment [22] of distributed applications will be submitted, we intend to generated their associated environment in order to experiment CODeX more deeply.

References

[1] R. Allen, D. Garlan, and J. Ivers. Formal Modeling and Analysis of the HLA Component Integration Standard. In *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, November 1998.

[2] R. Balter, L. Bellissard, F. Boyer, M. Riveill, and J.-Y. Vion-Dury. Architecturing and Configuring Distributed Applications with Olan. In *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England, September 1998.

[3] J. Bézivin. From Objects to Model Transformation with the MDA. In *Proceedings of TOOLS'USA*, volume IEEE-TOOLS 39, Santa Barbara, August 2001.

[4] L. Catledge and C. Potts. Collaboration during conceptual design. In *Proceedings of the 2nd Intl. Conf. Requirements Eng. (ICRE'96)*, Colorado Springs, USA, 1996.

[5] E. Dashofy, A. van der Hoek, and R. Taylor. A Highly-Extensible, XML-Based Architecture Description Language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001)*, Amsterdam, Netherlands, 2001.

[6] G. Kiczales *et al.* Aspect Oriented Programming. In *Proceedings of 11th European Conference on Object Oriented Programming (ECOOP'97)*, LNCS 1241, pages 220–242, Jyväskylä, Finland, June 1997. Springer Verlag.

[7] D. Garlan, R. Monroe, and D. Wile. Acme: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, Novembre 1997.

[8] A. Goldberg and D. Robson. *SmallTalk-80, The Language and its Implementation*. Addition-Westley, 1983. *ISBN: 0-201-11371-6*.

[9] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):336–355, April 1995.

[10] R. Marvie and P. Merle. CORBA Component Model: Discussion and Use with OpenCCM. Technical report, Laboratoire d'Informatique Fondamentale de Lille (LIFL), June 2001.

[11] R. Marvie, P. Merle, and J.-M. Geib. Towards a Dynamic CORBA Component Platform. In *Proceedings of the 2nd International Symposium on Distributed Object Applications (DOA'2000)*, pages 305–314, Antwerpen, Belgium, September 2000. IEEE. *ISBN: 0-7695-0819-7*.

[12] V. Matena and M. Hapner. *Enterprise Java Beans Specification v1.1 - Final Release*. Sun Microsystems, May 1999.

[13] N. Medvidovic and R. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 60–76. Springer–Verlag, 1997.

[14] N. Medvidovic and R. Taylor. Exploiting Architectural Style to Develop a Family of Applications. *IEEE Proceedings Software Engineering*, 1997.

[15] P. Merle, C. Gransart, and J.-M. Geib. Using and Implementing CORBA Objects with CorbaScript. *Object-Oriented Parallel and Distributed Programming*, 2000. Ed. Hermes, *ISBN: 2-7462-0091-0*.

[16] J. Miller and J. Mukerji. *Model Driver Architecture (MDA)*. Object Management Group, July 2001. OMG TC Document ormsc/2001-07-01.

[17] OMG. *Meta Object Facility (MOF) Specification version 1.3*. Object Management Group, March 2000. OMG TC Document formal/00-04-03.

[18] OMG. *CORBA 3 New Components Chapters*. Object Management Group, November 2001. OMG TC Document ptc/2001-11-03.

[19] OMG. *CORBA Scripting Language Specification, v1.0*. Object Management Group, June 2001. OMG TC Document formal/01-06-05.

[20] OMG. *CORBA/IIOP 2.4.2 Specification*. Object Management Group, February 2001. OMG TC Document formal/01-02-01.

[21] OMG. *OMG Unified Modeling Language Specification*. Object Management Group, September 2001. OMG TC Document formal/2001-09-67.

[22] OMG. *Deployment and Configuration of Distributed Applications RFP*. Object Management Group, January 2002. OMG TC Document orbos/2002-01-19.

[23] OMG. *OMG XML Metadata Interchange (XMI) Specification version 1.2*. Object Management Group, January 2002. OMG TC Document formal/02-01-01.

[24] OMG. *UML Profile and Interchange Models for Enterprise Application Integration (EAI) Specification*. Object Management Group, September 2002. OMG TC Document ptc/2002-02-02.

[25] OMG. *UML Profile for Enterprise Distributed Object Computing Specification*. Object Management Group, September 2002. OMG TC Document ptc/2002-02-05.

[26] T. Thai and H. Lam. *.Net Framework Essentials*. O'Reilly, 2001.

[27] N. Wang, D. Schmidt, and C. O'Ryan. *Component-Based Software Engineering: Putting the Pieces Together*, chapter An Overview of the CORBA Component Model. Addison-Westley, 2001.