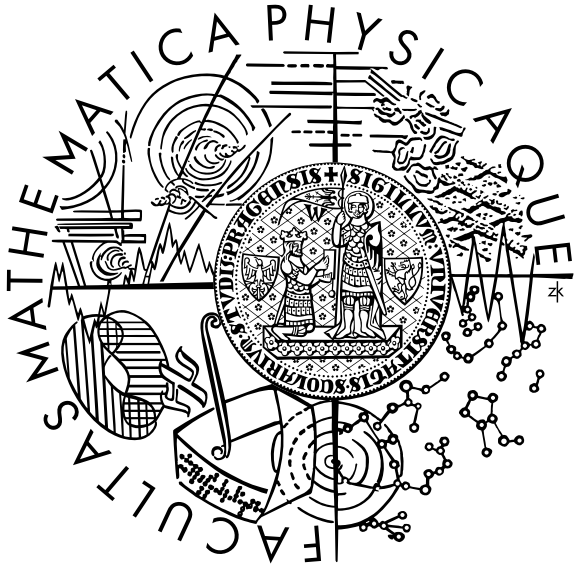
Charles University, Prague, Czech Republic
Faculty of Mathematics and Physics

MASTER THESIS



Pavel Šafrata

Infrastructure for Deployment of Heterogeneous Component-based Applications

Department of Software Engineering
Supervisor: Ing. Lubomír Bulej
Study program: Computer Science, Software Systems

I would like to thank my advisor, Lubomír Bulej, for his extraordinary attention paid to my work, for providing me a lot of his experience in the expert field and for many discussions on the elaborated issues. I also thank Michal Malohlava, Tomáš Poch and Ondřej Šerý for the various valuable help they provided me and Markéta Kolářová for checking correctness of the text grammar. Last but not least I thank my family for supporting me during my work.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with making this thesis publicly available.

In Prague on 15^{th} January 2007 Pavel Šafrata

Contents

Název práce: Infrastructure for Deployment of Heterogeneous Component-based Applications
Autor: Pavel Šafrata
Katedra: Katedra Softwarového Inženýrství
Vedoucí diplomové práce: Lubomír Bulej
e-mail vedoucího: bulej@nenya.ms.mff.cuni.cz
Abstrakt:

Nasazení aplikace je proces zahrnující všechny činnosti prováděné s aplikací od momentu jejího vydání. Různé komponentové modely řeší tyto aktivity odděleně (pokud vůbec), přestože koncepce je většinou stejná. Formální kodument Deployment and Configuration of Component-based Distributed Applications Specification vydaný organizací OMG navrhuje jednotné řešení, které může být přizpůsobeno pro různé komponentové modely.

Tato práce se soustředí na část problému týkající se spouštění aplikací a prezentuje jednotnou infrastrukturu založenou na uvedené specifikaci. Hlavním cílem je prozkoumat možnosti podpory více komponentových modelů a následně heterogenních aplikací, které sestávají z komponent implementovaných v různých komponentových modelech. Toho bylo dosaženo navržením systému rozšíření umožnujících odstínit specifika jednotlivých komponentových modelů od společné infrastruktury.

Přestože zmíněná specifikace nebyla určena pro podporu heterogenních aplikací, implementace se od ní odchyluje jen v několika málo bodech. Ve všech takových případech je prezentována analýza situace a odůvodnění příslušné odchylky.

Klíčová slova:

komponentové aplikace, heterogenní komponentové aplikace, nasazení komponent

Title: Infrastructure for Deployment of Heterogeneous Component-based Applications
Author: Pavel Šafrata
Department: Department of Software Engineering
Supervisor: Lubomír Bulej
Supervisor's e-mail address: bulej@nenya.ms.mff.cuni.cz
Abstract:

Deployment is a process which involves all actions performed with an application after it is released. Traditionally, deployment has been addressed for each component model separately (if at all), even though most of the concepts are the same. The Deployment and Configuration of Component-based Applications Specification released by OMG proposes a unified approach that can be tailored to different component models.

This thesis focuses on the execution phases of the deployment process. It presents a generic deployment runtime based on the OMG specification. The main objective is to elaborate support for multiple component models and subsequently support for heterogeneous applications consisting of components implemented in different component models. This has been achieved through a system of extensions which allows isolating component model specifics from the runtime.

Even though the OMG specification was not originally intended to support heterogeneous applications, the implementation deviates from it only in a few points. In all such cases, the thesis presents an analysis of the situation and rationale for the deviation.

Keywords:

component-based applications, heterogeneous component-based applications, component deployment

1 Introduction

1.1 Component-based Programming

As the size of applications grows, the need to define them modularly becomes more urgent. Also the reusability of code units having some bordered capabilities bears a great emphasis nowadays. The component-based programming goes towards these requirements.

Using the component-based approach, the application is built from entities called *components*. There are many views, but we will consider component to be a black-box communicating to its neighbours through interfaces. Two types of interfaces are distinguished. The *provided* (or *server*) interfaces are similar to the interfaces known from object-based programming – they provide an access to a functionality implemented by the component. The *required* (or *client*) interfaces indicate functionality required from another components to be able to perform the provided functionality. When the required interfaces are satisfied by provided interfaces of other components, the provided interfaces are working and can be used. The interfaces define the only way for components to communicate to their neighbours, no other channels are allowed.

If all interfaces of the components are exactly defined as well as their behavior, they can be developed separately, by different teams or even different vendors, on the basis of the definition. Putting the components together to create the whole application is then performed easily by connecting the required interfaces to the provided interfaces of other components. A component is therefore a highly modular and reusable part of a system, bringing the encapsulation principle to an excellent level.

The component-based programming goes even further. Each component can be similar to an application, broken down to smaller components, keeping the same approach. Components can be defined recursively, implementing their features by connecting features of several more specialized components together. Such component does not require any specific code, it can use a principle of *interface delegation* – the component contains several subcomponents connected to provide the functionality and each of the parent component interfaces is delegated to an interface of one of the subcomponents. That means incoming calls to such component interface are directly passed to a similar interface of some subcomponent. The same applies in the other direction.

Thanks to this recursive approach, the architecture of the application can be designed and presented in a transparent and highly readable top-down form and the modularity and reusability of the code is even bigger.

Component-based programming is a paradigm and many *component models* exist. Among others, we can mention Fractal [2], SOFA [20], CORBA [15] or EJB [8].

1.2 Deployment

The deployment process is an integral part of the application lifecycle. It involves actions performed after the application is released by the developer – actions like installation, executing, un-installation etc.

Historically, each component model either came with its own solution of the deployment problem, or did not address the problem at all. In some cases, the deployment process even differs among different vendors of the same component model (e.g. the EJB model). But the idea is in most cases quite similar because an execution of component-based application typically consists of instantiation and configuration of application components and interconnection of their interfaces.

The goal of the effort within the Object Management Group (OMG) was to avoid the situation of EJB where a single idea is realized in many different ways. Therefore they have created a formal document called Deployment and Configuration of Component–based Distributed Applications Specification [16], from now on referred to as OMG D&C. This document serves as a specification of the deployment process for the CORBA Component Model (CCM) but it defines a unified approach to solve the deployment issues of component-based applications and is usable generally. The deployment support is first defined with respect to common issues and principles, not taking a particular component models into account, and then specified for the CCM. The general approach can be adopted by different component models and technically further specified to follow the component model specifics.

1.3 Heterogeneous Component Applications

One of the advantages of the component-based programming is the opportunity to develop components separately. Our work goes further in this direction. We believe that it should be possible to develop components even more independently without presumptions made on a language used to write the code and a target distribution of the components. Looking from the other side, it is possible to create an application from components, which have no common designer and were developed absolutely independently.

The most important attribute of such application is *heterogeneity* of the component models used. That means each component can be possibly written in another language using another component model. A component in its abstract shape has its interactions to its neighbours defined so clearly that it does not need to put any technical presumptions on their internal implementation. Should we assume, that particular models bring this idea to work in their implementations, it is then possible to build a system with an ability to connect a heterogeneous application.

One more aspect is related to this approach. A component should indicate functional requirements on its neighbours through required interfaces but it should anticipate neither their implementation nor placement. Looking from the other side, if we're building an application from independently developed components, it is natural that such components use their neighbours as if they were written in the same language using the same component model, and typically they are using direct references to communicate with the other components. But we aim

to deploy the applications to a distributed target environment. Using some well-known middleware techniques, it is possible to provide remote connections among components, transparently to their implementation code.

Let us summarize the mentioned observations. Heterogeneous component-based application is an application built from components that are written in possibly different languages using different component models. Application components typically assume that the other components are written by the same techniques and can be referenced directly. The environment, which the application lives in, should take care of connecting such components – to provide a transparent distribution and an interlayer needed to connect components with different connection principles and maybe different data formats used.

1.4 Problem Statement

Our goal is to allow deployment of heterogeneous applications while the state of art does not allow us to use the existing systems (we cannot deploy one application using multiple deployment systems). The OMG D&C also does not address this problem. It provides a standardized basis for deployment support but not with heterogeneity in mind, not a unified deployment implementation for all models. However, the specification can be considered a good basis for the work because it represents a serious amount of work directed to unify all the deployment approaches.

The problem is to find a way to deploy components using many component models by one deployment system at the same time. This problem is addressed by neither component model vendors nor the OMG.

1.5 Goals of the Thesis

The main goal of the thesis is to design and implement an environment in which we could execute heterogeneous applications. The OMG D&C serves as a basis for our work. The requirement is to adhere to the general part of the specification as much as possible not pulling particular component-model-specific issues into the system, however, without making compromises to deployment of the heterogeneous applications. Support for particular component models is to be realized by extensions which will provide all component-model-specific features. With this approach, adding support for new component models and following the future OMG D&C schema progress will be easier.

An associated goal is to produce an evaluation of the OMG D&C specification. The specification will be examined closely and an implementation based on it will be created. That should be sufficient to produce some erudite summary of the specification usability notes.

1.6 Structure of the Work

In section 2 we provide a general overview of the OMG D&C. We delimit parts relevant to our work and provide a closer overview of these parts.

We aim to provide support for deployment of heterogeneous component-based applications (from now on called also shortly *heterogeneous deployment*), which is not within the scope of the OMG D&C. In section 3 specifics of the heterogeneous deployment support built on the OMG D&C specification are analyzed.

In section 4 there is provided an overview of several component models, whose support is later designed and implemented as a proof of concept. Several patterns common to more component models and relevant for the work are pointed separately and their relation to deployment and to OMG D&C is analyzed.

At this point, the problem is analyzed and a sequence of more specific goals necessary to reach the general objectives can be formulated. This is done in section 5.

In section 6, solution of deployment support for heterogeneous applications is designed, discussing different alternatives. Architecture of the *deployment runtime* providing this functionality is outlined.

Several specific issues related to the heterogeneous deployment support are elaborated in section 7. Data model and interfaces are defined. Support for particular component models is designed.

Having the design, the prototype implementation can be built. In section 8, description of the solution is provided. Concrete implementation of the OMG D&C deployment system extended by the design of heterogeneous deployment functionality is described. Several technical problems are analyzed and reasons for their solution explained. Solutions for the particular component models are described closely, as well as examples provided to test functionality of the system.

One of the goals is to provide an evaluation of the OMG D&C. In section 9, notes on both conceptual and technical usability of the specification collected during the work on the system are provided.

Evaluation of results of this work from the view of the goals, as well as confrontation to another works on a similar topic is provided in section 10.

Finally, section 11 contains a general conclusion of this work.

2 Overview of the OMG D&C Specification

OMG Deployment and Configuration of Component-based Distributed Applications Specification (OMG D&C) [16] attempts to standardize a deployment process of component-based applications. It defines a general schema of a system providing the deployment functionality. This schema is to be further specified for particular component models using the Model Driven Architecture (MDA) [18] approach.

The specification defines the deployment process as a sequence of actions, beginning after a producer of an application packages the application along with its metadata to a package and provides the package to others. The following deployment phases are defined:

- **Installation**: The packaged application is stored into a repository under the deployer's control.
- **Configuration**: Parameters of the installed application are configured to

match the user's preferences.

- **Planning**: Decisions about destination locations of the application parts are done together with resolving application requirements on the target environment. This phase is performed by a *planner*.
- **Preparation**: The target environment is prepared to execute the application on the basis of the planning result.
- **Launch**: The actual execution of the application.

The specification addresses all these phases and is based on the following models:

- **Component Model**: Maintains description of component-based applications.
- **Target Model**: Maintains the target environment.
- **Execution Model**: Maintains the execution of the applications.

Each of these three models is divided into two sub-models - Data Model defining metadata required for the particular purpose and Management Model defining interfaces providing the particular functionality.

This thesis focuses on the Preparation and Launch phases. From now on, these two phases will be collectively called *execution*. In connection with that the Target and Execution models are important for the work. The Execution Model addresses the execution problem itself so it is in center of our interest. It uses directly the Target Model for its work so the Target model has to be taken into account as well. The Component Model is processed in the previous deployment phases and is not used directly from the phases involved within the scope of this thesis.

The specification further defines a deployment process using the defined metadata and interfaces. Parts of this process related to execution of applications are the most important for the work.

The last part of the specification defines a specialization of all the deployment models for the CORBA Component Model which is not much important for this work, but can provide better insight into intentions of the authors.

2.1 Component Model

The Component Data Model defines metadata for describing component-based applications.

In the specification an *application* is nothing special. It is simply a top-level component, which provides some reasonable functionality by itself, executed standalone.

A component has provided and required interfaces in this model called *ports*. Provided interfaces are called *provider ports* and required interfaces are called *user ports*.

A component is either a *monolithic component* or an *assembly*. The monolithic component contains a code implementing its functionality. The assembly functionality is realized by subcomponents. In this model, an assembly is a *virtual* component. That means it has no code and serves only as a logical container for its subcomponents. The subcomponents are connected to each other to provide the functionality and each assembly port is delegated to a port of one of its subcomponents.

The Component Data Model is not so important for understanding the approach presented in this work, therefore we omit further description. Additional details can be found in the original specification [16]. The Component Management Model is totally out of scope of this work, therefore its description is omitted completely.

2.2 Target Model

The Target Model addresses issues related to maintaining *target environments* – computing systems on which applications are executed. The model is split into Target Data Model and Target Management Model. The Target Data Model defines metadata for describing distributed target systems (systems that are to be used to deploy the applications) and their capabilities. The Target Management Model defines interfaces for collecting and retrieving the metadata defined by the Target Data Model.

2.2.1 Target Data Model

A simplified view of the Target Data Model is provided in figure 1. The target system consists of `Node`s interconnected to a `Domain`, which is a top-level entity. `Node` is a target for execution of component instances, e.g. a PC. For interconnection of the `Node`s is defined an `Interconnect` entity representing a direct connection between `Node`s and a `Bridge` representing an indirect connection between `Interconnect`s. `Interconnect` is the target for deployment of inter-component connections.

Each of these entities has its `Resource`s that are matched against application requirements. The last basic entity is a `SharedResource` which represents a named resource shared by multiple `Node`s. The model defines several types of resources which determine the algorithm of their matching with requirements.

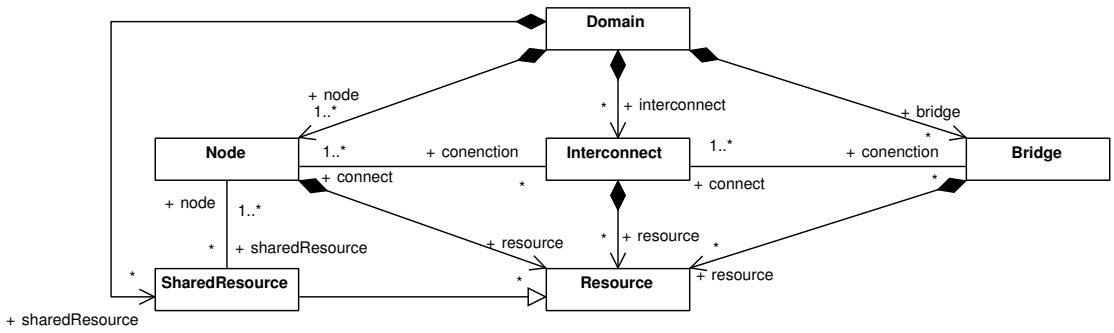


Figure 1: Simplified OMG D&C Target Data Model

2.2.2 Target Management model

The main entity of the model is a `TargetManager`. It has knowledge about a target domain and all resources provided by members of the domain. It can be asked to create a `ResourceCommitmentManager` object which maintains commitment (reservation) of resources.

2.3 Execution Model

The Execution Model addresses issues related to execution of deployed applications. The model is split into Execution Data Model and Execution Management Model. The Execution Data Model defines metadata necessary for execution of a component-based application in a distributed target system. The Execution Management Model defines *deployment runtime* support needed on the target environment to perform the execution – interfaces for performing the `Prepare` and `Launch` phases of the deployment process using the metadata defined in the Execution Data Model.

2.3.1 Execution Data Model

The Execution Data Model realizes an output of the Planning phase and input for the deployment runtime performing the Prepare and Launch phases. The data contains a full description of an application needed for its deployment, including its requirements and the placement decisions made during planning phase.

The component structure of the application is flattened – assemblies are broken to monolithic subcomponents and delegated connections are connected directly. Assemblies are important at the design phase as a useful abstraction, but delegate all their functionality to subcomponents, so they do not have to be instantiated physically.

A simplified view of the Execution Data Model is provided in figure 2. The top-level entity of this model is a `DeploymentPlan`. It contains all information required to deploy an application. To make the following text more readable, we will reference the entities by their roles instead of their names, as the role names are more intuitive. The `DeploymentPlan` contains set of `instance`s, representing particular component instances of the application. Each `instance` has an `implementation`, which describes the monolithic implementation of the component. An `implementation` contains a set of `artifact`s referencing the actual code artifacts implementing the component. Furthermore, the `DeploymentPlan` contains a set of `connection`s, describing interconnects among component instances. The `instance`s and `connection`s contain `deployRequirement`s, that are matched against target system resources. The `DeploymentPlan` also contains a `realizes` section which describes a component interface implemented by the plan, including a set of `port` entities describing particular ports.[1]

The `artifact`s and `implementation`s have a set of `execParameter`s used for controlling the component instantiation process. The `instance` has a set of `configProperty` entities used for configuring the instantiated component. Each

[1] Entities of the `realizes` section are defined in the Component Data Model.

of these parameters and properties is a name-value pair whose value has type `Any` which represents any data type present in the component model.

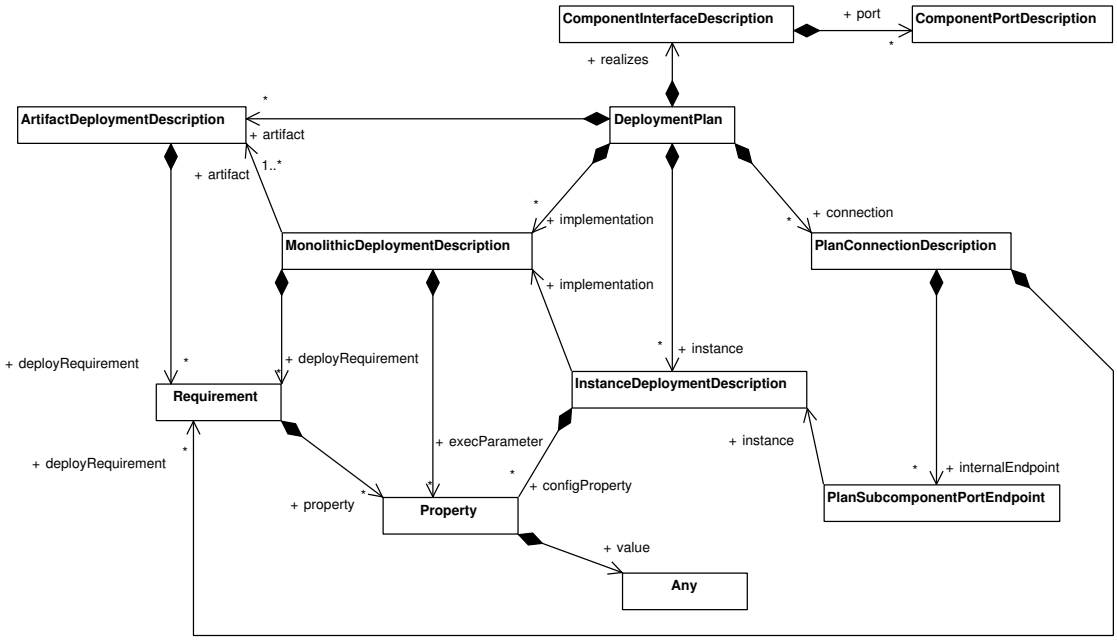


Figure 2: Simplified OMG D&C Execution Data Model

2.3.2 Execution Management Model

The Execution Management Model defines interfaces of a deployment runtime. A runtime implementing these interfaces accepts a `DeploymentPlan`, created by the planner. On the basis of this information it instantiates the application components, interconnects them and starts the application. This is the most important model for our work.

A simplified view of the Target Management Model is provided in figure 3. First top-level entity of this model is an `ExecutionManager`. It is a domain-wide starting point for execution of an application. Second top-level entity is a `NodeManager` which is controlled by the `ExecutionManager` and maintains execution on a particular node. The `ExecutionManager` maintains the `NodeManager`s to be connected, composing the domain. The `ExecutionManager` also commits resources on the `TargetManager`.

The `ExecutionManager` creates a `DomainApplicationManager` which provides a domain-wide control over the deployed application. It communicates with `NodeApplicationManager`s (created by the `NodeManager`s), which control parts of the application deployed to the particular nodes.

Bottom-most entities are `DomainApplication` and `NodeApplication` created by the application managers, representing the deployed application itself (domain-wide and node-wide).

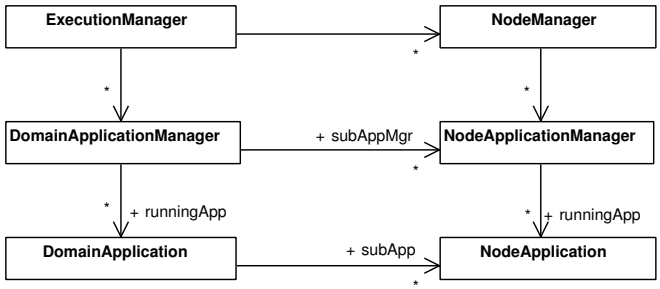


Figure 3: Simplified OMG D&C Execution Management Model

2.4 Execution Process

When the target environment is established and the runtime is running, applications can be executed. Input for the execution process represents a Deployment Plan which was created during previous deployment phase by the planner. An application is executed in three steps:

- **Prepare Plan**: In this step, the application `DeploymentPlan` is given to the `ExecutionManager`, which initiates the deployment system to prepare for launch of the application described in the plan (resources are committed, if it was not done by the planner, artifacts are downloaded to the nodes etc.). Result of this action is a new `DomainApplicationManager` which is connected to `NodeApplicationManager`s maintaining the application on the nodes.

- **Start Launch**: This is a first step of the `Launch` phase, initiated by the `DomainApplicationManager`. In this step, the application is executed, but not started yet. The application components are instantiated and configured and their provided ports are retrieved. As this step may be done repeatedly on one `DomainApplicationManager`, there can be a necessity to commit another resources. Result of this step is a `DomainApplication` which is connected to `NodeApplication`s representing the application on the nodes.

- **Finish Launch**: Second step of the `Launch` phase is initiated by the `DomainApplication`. The references collected in the previous step are passed to the components to connect their required ports. After that, the application is interconnected and is ready to run. Finally, the application is started.

The `DomainApplicationManager` provides also functionality for exiting the running applications.

2.5 Vendor Boundaries

The specification defines *vendor boundaries* which delimit parts of the whole system and have precisely defined interactions. This allows multiple vendors to develop parts of the system independently.

The specification identifies three parts of the deployment system that are suitable for independent development. The Target Management Model represents one of them. The Execution Management Model is divided by a vendor boundary into two parts - Execution Manager and Node Manager. From this view, `DomainApplicationManager` and `DomainApplication` entities are part of the `ExecutionManager`, they are created and maintained by it. Similarly, `NodeApplicationManager` and `NodeApplication` are parts of the `NodeManager`. Finally, `ResourceCommitmentManager` is part of the `TargetManager`. Communication between these three sets of objects runs through interfaces exactly defined in the OMG D&C, the interfaces inside of each group can be defined by vendor. That means, each of these sets have a common developer, possibly different from the other sets.

2.6 Model Driven Architecture

Model Driven Architecture [18] is a standard created by OMG which defines an approach to design software systems. It attempts to separate abstract, general design from implementation and technical issues. A system is developed incrementally by creating models to which model transformations are applied. First, a Platform Independent Model is created. This model contains only general ideas and should be absolutely independent of the platform, that can be used to implement the system. After that, model transformations are defined to incrementally transform the Platform Independent Model to more and more platform-dependent and better specified models, called Platform Specific Models. The transformations can go as deep as desired, possibly finishing at the bottom by an executable code.

The OMG D&C specification is compliant with this approach. The core of the specification defines a Platform Independent Model which contains metadata and interfaces as described in previous sections. Besides that, the specification contains model transformation from the Platform Independent Model to a specific model for CORBA component system. Further transformations are defined from the CORBA Platform Specific Model to Platform Specific Models for IDL (Interface Definition Language) and XSD. Transformations for the other component models are left on users of the specification.

Benefits and drawbacks of this approach for our work are examined in section 3.2.

3 Specifics of Heterogeneous Deployment

Deployment system presented in the previous section was intended for deployment of homogeneous component applications. The objective to deploy heterogeneous applications puts additional requirements on the system:

1. **Metadata** The metadata have to carry description of components using different component models.

2. **Deployment Runtime** The runtime has to be able to process the data and execute components built on different component models.

3. **Communication among Components** The components can communicate over various middleware platforms and use incompatible type systems.

3.1 Metadata (Data Model)

The OMG D&C specification defines a schema of component metadata (using MDA). The schema is quite general and is to be further specified for any particular component model. However, a separate metadata schema for each particular model is not suitable for heterogeneous deployment. Component models differ from each other, and we need to represent many models in one metadata schema – the deployment system needs to use this metadata to execute components built on different component models.

One possible approach is explored by Petr Hnětynka in [9]. In this work, many component models are examined and their features analyzed. Then a "unified component supermodel" is defined. This unified schema defines metadata containing all patterns present in any of the examined component models.

Our approach is different, adhering to the metadata schema defined in the OMG D&C more closely than the described unified model does. The metadata schema defined in the OMG D&C specification should contain all basic patterns needed for "common" component models (models, which use component architecture similar to the architecture described in section 1.1). If a particular component model needs more extending data, it should be insignificant for the deployment runtime. This information can be hidden in some properties and the runtime should not need to understand it.

3.2 Heterogeneous Deployment Runtime (Management Model)

The OMG D&C specification is compliant with the Model Driven Architecture which is not very suitable for objectives of this thesis. The main idea of this approach is the gradual specification from general to specific models. From one, general model can be derived several more specific models, different from each other.

Using MDA as proposed in OMG D&C, we would get a different deployment system for each component model. But we propose that there should be one system for many component models. In the specification, the only deployment model independent of the component models is the general Platform Independent Model. However, it is too general to be used as it is. It contains several entities that are to be defined by each Platform Specific Model (i.e. for each component model separately).

To reach the goals, one MDA step is to be proceeded, from the Platform Independent Model to a Platform Specific Model for Heterogeneous Deployment. This model should extend the defined Platform Independent Model as few as possible, getting close to a direct implementation of it. The unspecified parts should be, to achieve the goals, specified rather minimalistically. They are to be defined rather

as a subset of all models with availability to add some properties than a superset of all models. This applies to the metadata as well.

3.3 Communication among Components

When components of a heterogeneous application are instantiated on nodes, they need to be connected so that they can communicate. This is actually quite complex process, because components of a heterogeneous application may not know about the heterogeneity and the distribution, so they can expect their neighbours to be in the same address space and using the same communication patterns, which needn't be true.

In classical models a component contains a business code and a middleware-specific communication code (figure 4). When connecting components, the server component first provides a middleware-specific reference to its provided port. This reference is passed to the client component's required port, which is *bound* to the provider, using the given middleware.

Connectors proposed in [1] separate these two parts. A component can contain only a business code, while the communication middleware-specific code is moved to a connector which realizes a connection to another component. A connector consists of *connector units* present on particular ends of the connection (an example of a simple connection is shown in figure 5). The component code is cleaner and the component can be connected using various middleware platforms, because connectors can be generated accordingly. This approach provides a transparent remote communication among components. It fits requirements of distributed heterogeneous deployment and is adopted by this thesis.

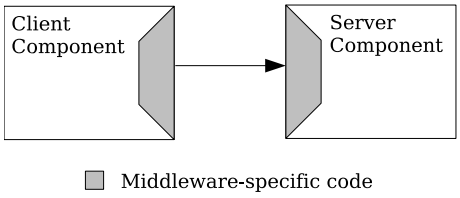


Figure 4: Simple connection between components

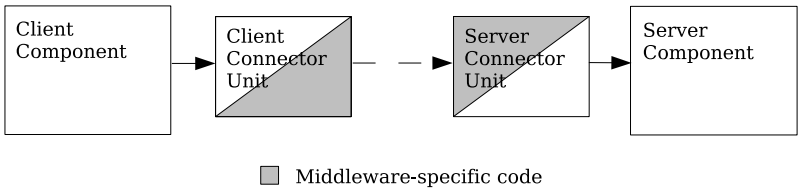


Figure 5: Connection between components with a generated connector

Using connectors does not necessarily break the compliance with OMG D&C even though the concept of connectors is not explicitely supported. They are expected to be generated during the Planning phase of the deployment process and

added to a deployment plan by a planner. From the view of deployment runtime, they can be treated as standard application components.

3.3.1 Interconnecting the Components

As described in the OMG D&C, the interconnecting of component instances has two steps – in the first step, provided ports are retrieved from all components and in the second step, required ports are filled by the references collected in the first step. The references have to be passed from all nodes remotely to the Domain Application common for the domain and then back to the nodes. The references are passed remotely but the application components may not know about it and typically use direct references. Therefore, the system has to be able to pass references remotely without loss of their meaning.

Moreover, the component can pass multiple references for one port. A component can for example provide a local and a remote reference. If client component of this port is deployed in the same address space, it can use the local reference which will be direct and fast. If it is deployed elsewhere, it will have to use the remote reference, which is much slower, but provides the remote communication. This applies mainly to the connectors – generally they may want to pass multiple references to the same port, using different middleware platforms. This approach makes the application more adaptable.

3.3.2 Connectors

Connectors provide transparent connections among components using various middleware platforms. They contain the code maintaining the communication so they are also suitable for implementing the heterogeneity (e.g. bridging the type systems of two component models).

Components still contain only the business code and both distribution and heterogeneity is implemented by connectors. The component simply expects its neighbours to be directly connected to its ports and to be built on the same component model. For each connection, a connector is generated according to the connection interface and models of both components. The connector units serve as proxy objects. For example, a simple procedure call from a client component to a server component is provided by two connector units (figure 5). The client component is connected to a client connector unit instead of the server component. On the other side, the server component has its provided port bound to a server connector unit. The units are then connected using the generated interface.

Components are always connected directly to local connector units. Note that even so the references are collected in the domain and sent back so the deployment runtime has to be able to pass them remotely.

Each connector unit is built on the same component model as the component it is attached to. Therefore the component does not recognize that its neighbouring components are built on another component models.

Heterogeneous application requires method invocation among components to

be fully transparent, independent of distribution and heterogeneity of the application components. The described connectors fulfill this requirement.

The remote communication has to solve several well-known problems, most important of which is passing references. When a reference is passed through a connector, the connector has ability to instantiate a new connector, serving as a proxy of the referenced object. Method invocations on this object will then produce another remote call, again fully hidden to the application components.

Connectors [1, 4, 3] have a hierarchical structure, composed from so called *elements*. A connector unit has a top-level element, that can contain a number of subelements, performing different tasks. The system is prepared for reconfiguration of the structure of the elements – elements can be removed or added to a data path.

Connectors are maintained by objects called Dock Connector Manager and Global Connector Manager. These managers handle lifecycle of connectors, the first one locally on a node and the second one globally for a whole system. They also provide the functionality for creating new connectors for remotely passed references.

During the generation of connectors a Remote Binding Map is created – a structure describing remote bindings among connector units.

3.3.3 Connection Reconfiguration

Connectors have an ability of requesting *reconfiguration* of their connections. This feature allows elements, that are no longer used to be excluded from the data flow, and new elements to be added. In case that all elements of a connector are switched off, the whole connector is excluded from the data flow and components are connected directly. This feature is also used at application startup for connection ordering – a component (or a unit), which needs to provide its provided port after a required port is set, uses the reconfiguration mechanism when it is able to provide the reference.

The usage of the reconfiguration feature during the application startup can be demonstrated on a following example: an application contains two connected components, built on the same component model. The planner generates a connector between these two components. The components happen to be deployed to the same node. After interconnecting of the application, the connector can recognize that both components can be connected directly and swith off elements providing the remote communication. If there are no other active elements, the connector is useless – it only slows down the data exchange. Such connector should have a possibility to exclude itself from the way and to initiate the reconfiguration of the connection resulting to direct connection of the two components.

The connectors [3] are prepared to get a callback object from their environment, on which they can request the reconfiguration. After such request, the specified connection is invalidated and connected again according to the new conditions.

However, this feature is incompatible with the OMG D&C. The drawback of the two-step interconnection defined in the specification for the reconfiguration purpose can be demonstrated on the case with a useless connector. The simplified idea

is that the server unit provides two references. The first reference is a remote reference to the server connector unit itself. The second reference is a local reference to the component (the server component of the connection represented by the connector). If the client unit finds the local reference usable in its address space, it will know that it is deployed locally and can pass the target component as its provided port. This connects directly the two components and the connector is not used any more.

The reason why the reconfiguration has to be addressed is that neither connector unit has its required ports bound when asked for provided ports. That means the server unit does not have a reference to the target component so it cannot provide the local reference to it. The client unit does not have a reference to the server unit, so it cannot provide anything else than reference to itself.

Using the starting process as described in the OMG D&C strictly, the connector elimination is not possible. The goal is to find the least-evil way to achieve this functionality from the view of the goal to adhere the OMG D&C.

4 Supported Component Models

There are many component models but we are especially interested in the Fractal [2] and SOFA [20] component models. This is mainly given by the context of another projects interacting with our work.

4.1 Fractal

Fractal component model [2] defines components with a hierarchical structure. Component can be either *primitive* (which corresponds to the monolithic component from OMG D&C) or *composite* (which corresponds to an assembly). The composite components delegate their ports to subcomponents and can be further nested in each other.

The model also supports *shared components* – several distinct components can have a common subcomponent. Then the shared subcomponent has several direct parents. However, using this feature makes the application architecture less readable.

Each component besides its business interfaces has also control interfaces, called *controllers*. They provide access to component state and properties. The most important controllers are the Lifecycle Controller (starting or stopping), Binding Controller (connections among components), Containment Controller (changing composite's subcomponents) and Name Controller (setting name of the component).

Delegation of composite's business interfaces to its subcomponents is realized through using *internal interfaces*. Component's common provided and required interfaces are called *external*. In case of composite component, each of these interfaces corresponds to an internal interface of the same type, but opposite direction (client/server). External interfaces of subcomponents can then be bound to internal

ports of their parent component. External and internal ports are mapped one-to-one and delegate the invocations on each other.

Binding between components is defined as a connection between client and server external interfaces of two components on the same level of nesting, or between client and server interfaces, one of them internal of a composite component and the other external of its direct subcomponent. In the binding, types of the interfaces do not need to have exactly the same definition – the server interface has to be of a subtype of the client interface to be able to accept all invocations. The system allows to define a *collection* interface which represents an (unbounded) array of these interfaces. Bindings among more than two ports are possible, but not as a core principle – they are realized by components dedicated to communication.

There is no deployment runtime. Each application is expected to have its own launcher which builds the architecture and starts the application. There is also a generic launcher for applications with a Fractal Architecture Description Language (ADL) specificaction. ADL provides metadata which describe the application architecture. It contains component implementation description in case of primitive components and containment hierarchy description in case of composite components, configuration information for all components, binding information etc. It does not take the distribution into account.

Next to that, a Fractal RMI is provided. It is an additional functionality providing transparent remote binding. It uses similar principles as the java RMI, but has its own implementation. The remote objects are not identified by the java RMI interfaces, but by Fractal interfaces of standard components. Both stubs and skeletons are generated automatically at runtime.

4.2 SOFA

SOFA is a component model with a hierarchical structure of components. Type of a component is represented by a *frame* which describes its external interfaces. A concrete realization of a frame is represented by an *architecture* which describes either monolithic implementation or subcomponents and bindings among them. The subcomponents can be described either by their frames, or further by their architectures. An application is described by an *assembly descriptor* which assigns architectures to particular frames.

A connection among interfaces is represented by a *binding*, which has three types – *connector* represents an ordinary binding among provided and required interfaces, *delegation* represents delegation of component interface to one of its subcomponents and *subsumption* represents delegation of parent component interface to the component. Components are physically interconnected by *connectors*, which contain a middleware-specific communication code and can provide transparent distribution (this applies to delegations and subsumptions as well). A connector can also provide binding among more than two interfaces.

Each component has business interfaces and controllers (similarily to Fractal). Collections of interfaces are also possible. Internal architecture of a component also employs a set of *microcomponents*. Microcomponents are objects with various

functionality. Each component has three chains of microcomponents – for provided interfaces, required interfaces, and control interfaces. Besides that, a component can have a number of standalone microcomponents. Invocations on component interfaces are passed through all microcomponents in the appropriate chain and then to component content which means either a monolithic implementation or subcomponents.

Components that are able to create other components at runtime are marked as *factories* and the components they produce are described as *dynamic components*.

There is no ADL – the metadata are stored in a repository generated from a metamodel and possibly serialized to a XML file.

The SOFA component model is designed to provide multiple deployment backends. There is only one implemented backend so far. It deploys an application to a distributed environment. The deployment system consists of a *repository* which contains all information about applications and of *deployment docks* which represent the actual target environment. When the deployment process starts, an assembly descriptor and architectures are used to create a *deployment plan* into which connectors are generated. This plan is passed to a deployment dock which hosts a top-level component. The application components are instantiated and bound recursively according to the component hierarchy by the deployment docks.

4.3 Common Concepts

There are several common concepts in the examined models. However, one of them deserves to be elaborated closer – component containment hierarchy. This concept is natural for both models and is important for them but the OMG D&C Execution Model suppresses this pattern.

4.3.1 Containment

Containment is a common pattern for most of component models (see section 1.1). A component can consist of several subcomponents, delegate its ports to them and let them process all the work.

The OMG D&C specification adopts the containment pattern in the Component Data Model. However, the Deployment Plan, which serves as the only input of the deployment process, is flat, with assemblies broken down to leaves in the containment hierarchy.

Besides the monolithic components and assemblies, we have considered another type of component – an assembly, which contains beyond the subcomponents also a code, thus a non-virtual composite component. The code typically modifies somehow the data coming from external ports to the subcomponents and back. This can be demonstrated on an example of a composite component translating a single configuration value to separate configuration values for several subcomponents. Even such component can be broken down, its code can be projected to the deployment plan as a monolithic component and ports can be connected accordingly.

However, in some cases, the containment information is required inside an

application. For example in Fractal, a component can browse its subcomponents and call methods on them (althoug only controllers may be called, no business methods). This can be important for the application. The containment can be also important during start and stop of the application as it determines an order of changing the state of particular components. In other models, the containment information can play even more important role. Therefore, it may be required to preserve the containment information throughout the deployment process.

The first option is to extend the Deployment Plan schema to preserve the containment information. This is quite straightforward solution but it is not necessary to extend the model and it breaks the goal to adhere to the OMG D&C specification.

From that point of view, it is much better to hide this information to component properties, not understood by deployment runtime. An assembly is then represented in the plan as a monolithic component without any code, only forwarding its ports and having some special properties carrying the containment information.

In a distributed application, subcomponents of a component can be deployed remotely, i.e. two subcomponents of one assembly can be deployed to different nodes. Some component models do not take this into account or their solution is incompatible with our approach. The containment relation can be viewed as a special kind of connection and thus connectors can be used to achieve the distribution. We call such connectors *containment connectors* – they maintain the distributed containment transparently by the presented principle of connectors.

5 Goals revisited

We have analyzed an OMG D&C which serves as a basis for our work. Then, we have discussed issues introduced by the support for heterogeneous components and finally we have introduced two component models. Now we can formulate a sequence of more specific goals required to achieve the top-level goals.

The top-level goal is to elaborate and prove the concept of heterogeneous deployment. We aim to adhere to the OMG D&C specification closely in the work. As mentioned before, the scope of this thesis involves the Prepare and Launch deployment phases. These phases are addressed by the Execution model, which serves as a basis for the work, and Target model, which is directly required by the Execution model. Both Data and Management parts of both models are important for the work. We aim to find a way how to extend these models with the heterogeneity. Outputs of previous deployment phases are to be simulated by hand because we have no running implementation to use.

5.1 Deployment Repository

During the deployment process, data have to be exchanged among parts of the system. Therefore, we need a repository for the Data Models. The data basis is necessary for the Management Models as they operate on it.

Platform specific Data Model for heterogeneous deployment has to be defined. Then a way to store the designed metadata has to be found. We aim to find a tool to load the metadata from a persistent form to a memory and write them back. It has to be done with emphasis for projectability of future development of the OMG D&C to the repository.

5.2 Deployment Runtime

Second step is the Deployment Runtime, thus, the Target Management Model and the Execution Management Model with support for heterogeneity. We aim to design a system of extensions implementing functionality of particular component models. This goal can be divided into two subgoals:

5.2.1 Common Core of the Heterogeneous Deployment Runtime

We aim to create a basic implementation of the OMG D&C Target and Execution Management Models. The implementation should be compliant as much as possible with the OMG D&C specification. Similarly to the data models, some entities have to be specified in a Platform Specific Model for Heterogeneous Deployment. The emphasis lies on the heterogeneity, therefore, rather simple implementation of the model is sufficient, creating a platform for a research on the field of heterogeneity.

5.2.2 Support for Different Component Models

The objective is to design a system of extensions adding particular component-model functionality to the runtime. We aim to isolate the system of extensions from the OMG D&C interfaces and to adhere to the vendor boundaries principle. To prove the concept, we planned to implement Fractal and SOFA extensions. However, before we could realize it, the SOFA project had been stopped and development of a new model SOFA2 started. At the time of writing this thesis, SOFA2 is not yet ready to be used. Therefore, we can provide only an analysis of the SOFA2 support and propose a solution without the actual implementation. The Fractal support can be fully implemented.

6 Deployment Runtime

A simplified architecture of the Deployment Runtime according to the OMG D&C is shown in figure 6.

The Deployment Plan entity represents the data input to the execution part of the deployment process produced by the Planner. The plan structure adheres the OMG D&C design and the Platform Specific Model for Heterogeneous Deployment defined in section 7.1. Representation of the plan is implementation specific. The managers are the top-level entities of the Target and Execution management models.

A part of the architecture suitable for the support for heterogeneity has to be identified. The natural choice is to hide the functionality behind interface of

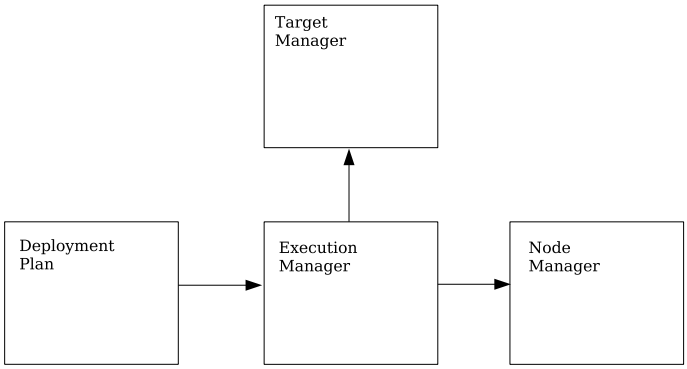


Figure 6: Architecture of the deployment runtime

the Node Manager because the Node Manager is the closest entity to boundaries of deployed applications. This also allows us to adhere to runtime interfaces defined in the OMG D&C.

Only a basic implementation of the Execution and Target managers adhering to the specification is required. Therefore we do not elaborate them closer. Design of the Node Manager is fundamental from this point of view and its architecture, supporting the heterogeneous deployment, is addressed in the next section.

6.1 Node Manager

The Node Manager has to adhere to the defined external interfaces and provide the features required for heterogeneous deployment. According to the specification, the Node Application Manager and Node Application entities are considered parts of the Node Manager.

6.1.1 Requirements on Node Manager Architecture

One of the goals is to provide a system of extensions providing support for particular component models. This system has to be designed as a part of the Node Manager.

Each component model vendor should have the possibility to implement support for its model and to add it to the deployment system. That means that the component model extensions are expected to be developed by a third party. Vendor boundaries have to be defined accordingly.

There are two options of where deployed applications will be instantiated. The Node Manager can directly instantiate application components in the same environment where it lives, or they can be instantiated more separately, in another environment. One of these options has to be chosen.

6.1.2 Node Manager Architecture

The architecture of the Node Manager is illustrated in figure 7.

When choosing a place where components will be instantiated, efficiency and stability is to be discussed. The first option of where the instantiated components

will live is directly the Node Manager's environment. This is the easiest, fastest and least memory-consuming way. The alternative option is to create a separated environment dedicated to host the deployed applications, called *execution environment*. An example of such environment can be a virtual machine. This consumes more memory and time and is not so straight-forward. On the other hand, it brings huge stability advantage – in case of crash of an application or in case of a malign application, there is dramatically smaller possibility of damaging the deployment runtime.

We have decided to instantiate applications in a separated execution environment because the stability advantage gained is more important for us than the loss of effectivity. Moreover, improving this approach a little, we can provide to the applications an option to request their own execution environment. This is useful because important applications can be isolated from influence of the other applications.

There is one more benefit of this approach. If we design and implement support for execution environments general enough, various environments can be employed to run components even on different platforms.

The extensions with support for particular component models are called *component model plugins*. A plugin is the only part of the system that knows the specifics of the component model required by an application component. This implies that the plugin has to be as close to deployed components as possible. Therefore support for plugins is put inside the execution environments.

To manage lifecycle of execution environments, Environment Manager is present in the architecture. It is responsible for creating execution environments upon request and shutting down unused environments. It keeps a list of running environments and provides them upon request. The Node Manager uses the Environment Manager for maintaining environments and uses the environments for maintaining application components, and it is isolated from the heterogeneity issues (i.e. component model specifics etc.). The execution environment maintains loading of component model plugins.

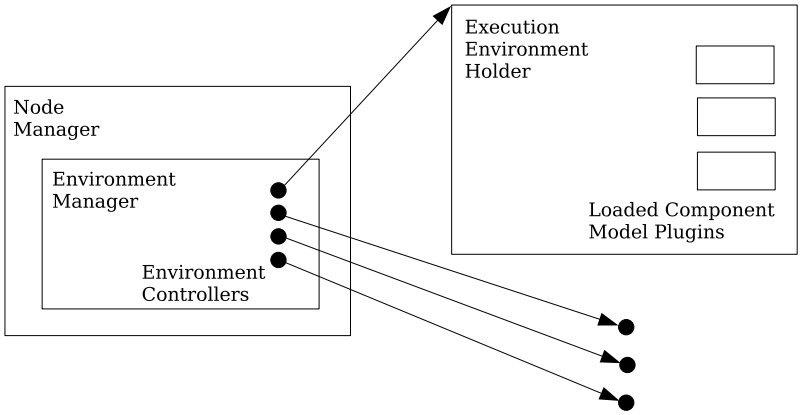


Figure 7: Architecture of the Node Manager

6.1.3 Execution Environments

Support for an execution environment has two parts - Environment Controller and Environment Holder. The Environment Controller provides communication to the environment and is present in the Node Manager. The Environment Holder is injected to the environment, communicates with the Environment Controller and maintains applications' components.

The Node Manager needs to deploy components into an execution environment. To allow that, the Environment Controller provides functions such as instantiating component, interconnecting component, starting component etc.

The Environment Holder lives inside the execution environment and controls application components according to requests sent by the Environment Controller. Next to that it manages plugins. The Environment Holder is not able to work directly with applications, it is still part of the common core of the deployment runtime and is not aware of specifics of particular component models. In fact, all it can do, is to load the required plugins and forward requests to them (as we mentioned before, the plugin is the closest entity to the application components).

6.1.4 Component Model Plugins

Component Model Plugin is an independent part of the deployment system. All the specifics of a particular component model are implemented inside a plugin and remain hidden to the core of deployment runtime. A plugin understands to all the properties containing data specific for the component model and uses them to maintain components.

The functionality of a plugin is similar to that of the environment – it provides functions for maintaining component lifecycle. Plugins do not participate on data transfers during the Prepare phase, their only purpose is to take care of components built on the particular component model – to prepare environment according to requirements of the model and to provide functionality needed to create, connect, configure and start the components.

6.2 Communication among Components

Application components will be connected using connectors. The deployment runtime has to connect the components to the connector units and the connector units to each other. To achieve this, it has to be able to pass references remotely and to pass multiple references for the same port.

The connector model corresponds to another project and level of integration of its implementation has to be chosen.

Special care has to be taken to support connection reconfiguration which is not supported by the OMG D&C.

6.2.1 Interconnecting the Components

A component typically communicates with its neighbours using direct references to their ports. Since we use connectors to implement communication among components, a component is always connected locally to a connector unit which is built on the same component model. That is, what the component expects. But the runtime collects provided references from all nodes in the domain and then sends them back so the references given by the components have to be modified to a remotely passable form and then modified back before delivering them to the required ports.

Connector units may need to pass multiple references for the same port. They may want to recognize whether they are deployed locally or not. Moreover, they may want to provide references for several different kinds of middleware platforms. But the connector units are aware of the heterogeneity and distribution so we can let them build the remotely passable references on their own.

Design is implied by the requirements. The translation of the direct references to the references passable remotely is a responsibility of plugins. A plugin provides communication between the deployment runtime and the component, so it is the most suitable place. The multiple reference building is a responsibility of connectors. However, they use their own data format. The translation from the connector data format to the deployment runtime data format is a subject of a general problem of translation between connector interfaces and deployment runtime which is addressed in section 6.3.

6.2.2 Connectors

We expect that the connector generator [5] is used by planner to generate connectors for an application. The connector generator implementation is in stage of prototype. Development of the planner is in progress and the planner is not ready to be used yet. Therefore the prototype of the deployment runtime uses hand-written input which should be normally provided by the planner.

For demonstrational purpose, we have decided to create simple connectors by hand. Doing that, we put emphasis to simulate functionality of the generated connectors as much as possible – their interfaces and external behavior are adopted. From the view of the system, the generated and hand-written connectors should be indistinguishable. This prepares the way for future merge of both projects.

6.2.3 Connection Reconfiguration

Connector units may want to reconfigure their connections after they are established, for example to reject a connector from where it is not needed. But the OMG D&C specification connects components in two steps (collecting provided references and setting required references) and does not address the reconfiguration problem.

The first solution is to extend the number of starting steps for exchanging references. Number of such steps has to be sufficient to give to each component a

chance to publish its provided ports after its required ports are bound. In the example of useless connector, that means: get provided port from the server component, give the reference to the required port of the server connector unit, get provided port of the server connector unit, give the reference to the required port of the client connector unit, get provided port of the client connector unit and give the reference to the required port of the client component. All these actions are in a causal dependence, none of them can be removed, therefore six steps are required instead of two. This is a strong diversion from the OMG D&C specification and means three times more remote calls.

The above solution can be improved by performing more rounds for exchanging references in a node, upon a single request from the domain. This can reduce the number of remote calls. In an extreme case, we can connect as much ports of components as possible in one node during each step. In the case from the above example, the first three steps can be done at once and the second three steps as well. So the connection can be connected in two steps as defined in the OMG D&C. In case of local deployment of both components, the interconnection with elimination of the connector can be even done in one step. However, the two steps from the specification are roughly redefined. In the first step, only provided ports should be retrieved from each component but several connecting rounds are processed in each node, components with local connections are connected. The same applies for the second step – instead of simply passing provided references to the required ports, number of components is connected in several rounds. Thus, this solution adheres to the number of two steps defined in the OMG D&C (from the view of remote calls from the domain) but it does not adhere to the defined semantics of the steps.

Last considered solution is to extend the deployment runtime with support for the reconfiguration callback. This solution adheres to the launching steps as defined in the OMG D&C. Besides this, another communication channel is added to allow reconfiguration calls during the second step (during the first step, the components do not get any new information so they do not have any reason to reconfigure the connections). The number of remote calls in this case depends on the number of reconfigured connections. Also in this case, it can be reduced by solving the callback on a node, if the connection is local. By this approach, we have adhered to the OMG D&C and added an extending functionality.

We have decided to employ the support for the connection reconfiguration callback – we can provide a deployment runtime which adheres to the OMG D&C and has an extra extension. If a deployment runtime entity does not support this extension, the system falls back to the behavior defined by the specification. This means, reconfiguration will not be processed. The vendor of a part of the runtime can rely on behavior of the other parts, which adheres to the specification, thus contracts defined by vendor boundaries are not broken. If all parts used to connect a connection implement the reconfiguration extension, the runtime will provide the reconfiguration functionality for the connection. The number of remote calls will be probably higher than in the other cases but the mentioned advantages are more important.

Using the reconfiguration callback, we can possibly provide connection reconfiguration during the application run, if the particular component model allows it. This is a strong new feature, not considered in the OMG D&C specification, and not provided by the other two solutions.

6.3 Vendor Boundaries

The deployment system according to the OMG D&C specification has several parts divided by vendor boundaries – Deployment Planner, Target Manager, Execution Manager, Node Manager. We have added several entities to the model and vendor boundaries among them have to be defined. Dependencies among the parts have to be elaborated and kept as weak as possible.

The parts separated by vendor boundaries are connectors and component model plugins. Execution environments belong to the Node Manager. They could be also considered a standalone part, a system of extensions with execution environments adding support for other platforms could be introduced. However, it would extend the scope of this thesis specified in section 5. The support can be added in future by implementing an Environment Manager providing this feature. The last part naturally separated by a vendor boundary are the deployed applications, as they will obviously have different vendors. Dependencies among these parts are explained in the rest of this section.

First, it is obvious that the applications should be independent of the deployment system. They have to provide metadata understood by the deployment runtime but the application code is totally independent. Moreover, the metadata can be created by the planner using another metadata format, e.g. format specific for some component model. On the other side, the deployment has to be independent of the applications, it is obvious as well.

Connectors are developed as a standalone project and should be independent of the runtime. Naturally, they define their own control interfaces. Therefore somewhere has to be a *bridge* from the connector interfaces to the deployment runtime ones. As we have described, connectors figure as components built on a particular component model. This implies the solution which is to put the bridge to component model plugins. Plugin will communicate outside to the runtime using runtime data format and interfaces and inside to the connector using connector data format and interfaces. That means that plugins depend on connectors. Planner also depends strongly on connectors, as it has to generate connectors and put them to a deployment plan. There are no more dependencies involving connectors except these two.

Concerning component model plugins, a dependency to connectors has been described. Plugins also depend on runtime. The deployment runtime provides support for plugins and thus defines rules for plugins to be usable. This dependency has to be kept as weak as possible, requirements on plugins have to be minimal and well defined. No other dependencies involving plugins are present.

If the environments were implemented also separately, they would depend on runtime, similar to the dependency from plugins to runtime. In that case, plugins

would depend on environments instead of runtime.

7 Heterogeneous Deployment Support Elaborated

In the previous sections, we have outlined a general design of the system. Before we can create a prototype implementation, several issues have to be elaborated more closely. First, a Platform Specific Model for Heterogeneous Deployment has to be defined. Second, naming conventions have to be provided to complete the contracts on vendor boundaries. Finally a solution of component model support has to be designed for each particular model.

7.1 Platform Specific Model for Heterogeneous Deployment

Several entities are not specified by the OMG D&C Platform Independent Model and are to be defined by the Platform Specific Models. Moreover, several changes in the model are required for the heterogeneous deployment. We propose following definition of the model.

7.1.1 Component Interfaces

Deployment Plan in the Platform Independent Model contains one general entity `realizes`, describing external component interface of the application. But a description of interface of each component is needed for its instantiation. Therefore, a `realizes` entity is added also to component `implementation`. This provides specification of interface of each component.

7.1.2 Port Properties

Particular component models can define specific kinds of component interfaces or may need to influence their behavior. To allow them to pass the needed extra information for each port, an `execParameter` is added to the `port` entity.

7.1.3 Logging

The Platform Independent Model introduces an optional entity `Logger` allowing central domain logging facility for Execution Manager and Node Manager. Neither the entity nor its interface is specified.

We have defined a simple interface, containing methods `debug`, `info`, `warn`, `error` and `fatal`. This is a general logging interface differentiating types of messages.

7.1.4 Connections

When interconnecting application components, references to their ports are transferred through the runtime. For this purpose, entities `Connection` and `Endpoint` are defined but not specified. The `Connection` entity represents connections among ports of components and contains a number of `Endpoints` representing the actual ports.

We have specified two requirements on these entities. First, they have to be passable remotely without loss of their meaning. Second, they have to provide a way to pass multiple references for each port.

The solution is straightforward. Each reference is translated to a string, which can be later translated back to the original reference. Such string is called *stringified reference*. For each port, a bundle of stringified references is passed. Each reference has a type (describing type of the reference – local/remote, used middleware..) and the stringified reference itself.

To be able to distribute the references correctly, the entities must have a unique identification within the scope of an application. The `Connection` entity can adopt unique name from its originating entity in the Deployment Plan. However, ports do not have any unique name in the Deployment Plan. To identify the `Endpoint`, we have considered several alternatives. The pair <connection name, port name> is not unique because one connection can contain more similarly named ports of different components. A better solution is the pair <component instance name, port name> – the instance name is unique within the scope of an application and the port name within the scope of the component instance. However, in the Fractal component model, a composite component has two ports of the same name, one internal and one external. The ports are always of different directions. Final solution is the triplet <instance name, port name, port direction (provider/user)> which is sufficient for all common models. A model allowing a component to have multiple ports of the same name and direction is considered "uncommon" and the deployment system will probably not be able to support it. This triplet is used in the whole system as unique port identification, during interconnecting and matching ports, reconfiguring connections etc.

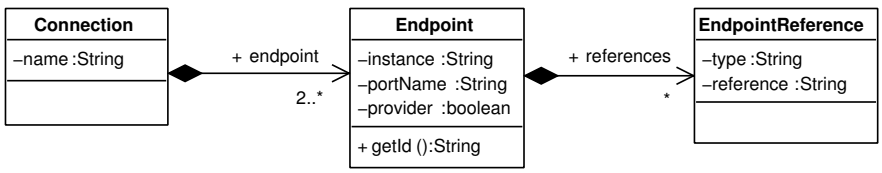


Figure 8: Platform Specific Model for Heterogeneous Deployment: Connection

From this point, the definition of the entities is straightforward. The diagram is presented in figure 8. The `Connection` contains a unique string name and a list of `Endpoints`. The `Endpoint` contains its unique identification, thus string name of the component instance, string name of the port and boolean value provider/user.

Then it contains a bundle of references, which is represented as a map from the string reference type to an element`EndpointReference`. The `EndpointReference` contains the string reference type and the string reference itself. Presence of the `EndpointReference` entity has a simple reason: it is sometimes useful to track the reference type from the reference itself, which would be difficult with the string-only form.

7.1.5 Any

One more entity is left unspecified by the OMG D&C Platform Independent Model – `Any`, which represents any data value from the particular component model. In the Platform Specific Model for CORBA it is defined as a pair <Data Type, Data Value> where Data Type is a bundle of entities allowing representation of any data type of the CORBA Component Model and the Data Value is another bundle of entities representing any value of these data types. However, since the core of the Deployment Runtime is common for all component models, the definition of `Any` cannot be specific to a particular component model. Therefore, we have decided to provide several common data types which the deployment runtime may need to recognize. The component-model-specific data types are encapsulated to strings.

The data type is represented by a `DataType` entity, the value is contained directly in the `Any` entity.

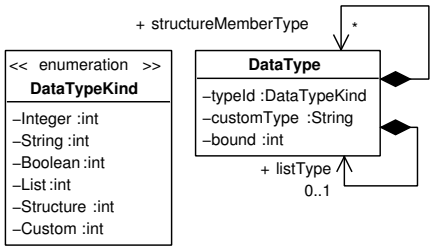


Figure 9: Platform Specific Model for Heterogeneous Deployment: DataType

A UML schema of the `DataType` definition is provided in figure 9. An enumeration entity `DataTypeKind` distinguishes the basic data types – it contains values `Integer`, `String`, `Boolean`, `List`, `Structure` and `Custom`. The `DataType` entity contains a value of the `DataTypeKind` type, describing the type. For the `Integer`, `String` and `Boolean`, this is enough. For the `List`, it contains another entity of the `DataType` type, representing the type of the list members. Optionally, the list's length boundary can be added. For the `Structure`, it contains multiple entities of the `DataType` type, describing data types of particular structure members. Finally for the `Custom` type, which represents any other more specific type, the string name of the type is present.

A UML schema of the Any definition is provided in figure 10. Structure of the `Any` entity representing a value corresponds to the structure of the `DataType`. It contains a `DataType` entity describing type of the value. Direct values for the `Integer`, `String` and `Boolean` data types can be used. For lists and structures, multiple

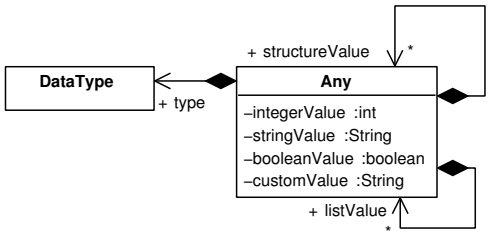


Figure 10: Platform Specific Model for Heterogeneous Deployment: Any

`Any` entities are contained. Finally, a value of the `Custom` type is encapsulated to a string.

The three basic types and two complex types are sufficient for matching application requirements to system resources, which is directly implied by the specification. Support for other types can be added to the model in future, if necessary. The encapsulation of the other types to string values is a responsibility of a plugin developer.

7.2 Naming Conventions

When the deployment system is starting up, its parts have to connect to each other so that they can communicate. To achieve that, naming conventions for these parts are defined, which allow them to find each other in a naming service.

- **Execution Manager** has to register itself in the naming service under the name `ExecutionManager`. The name can be this simple because the Execution Manager has only one instance in a domain.

- **Node Manager** has to register itself under the name `NodeManager/<name of the node>`. The <name of the node> part is to be substituted by the name, which will appear in the domain description and in the deployment plan.

- **Target Manager** has to register itself under the name `TargetManager`. The name can be this simple because the Target Manager has only one instance in a domain.

7.3 Fractal Component Model

For the Fractal component model three main issues have to be elaborated: suitability of Fractal component types for requirements of the deployment, containment hierarchy of application components and metadata containing fractal-specific deployment information.

7.3.1 Component Types

The Fractal primitive and composite components correspond to the monolithic components and assemblies from the OMG D&C. In section 4.3.1 we have identified another type of component – a composite component, which is not clearly virtural, thus has both code and subcomponents. Such components are not present in the Fractal model but we have decided to extend its scope and provide such option to applications. We have defined a new type of Fractal component called *hybrid*. Such component does typically not adhere to the one-to-one mapping of external and internal ports, defined for composite components. Therefore, a hybrid component has only external ports. It can modify the passing data anyhow. If some ports are only delegated to subcomponents, this delegation has to be implemented manually by sending the incoming data from one external port to another external port. In other words, hybrid component can be viewed as a primitive component with the containment of subcomponents extension.

7.3.2 Containment

Each Fractal component has knowledge about its content (subcomponents structure in case of a composite component, implementation in case of a primitive component[2]) and its parents. To retrieve and change subcomponents and parents, a `ContentController` and `SuperController` are present in each component. A component cannot use references to subcomponents and parents to invoke methods on their business interfaces, for this purpose a binding has to be defined. It can only call controllers on such components.

The deployment framework described in this work allows distribution of an application to different nodes. Therefore, it has to handle a situation that two subcomponents of one component are deployed to different nodes. A component can be deployed to different node than its subcomponents and parents.

The containment relation can be viewed as a special kind of connection between components, thus we have solved the problem by connectors. We have created a *containment connector* providing the remote containment, thus, the proxy object of component controllers.

For each containment relation, two such connectors have to be instantiated by the plugin. One connector as proxy of child for parent, second as proxy of parent for child. The containment connector is the same for all Fractal components with the same controllers. Controllers are generated to components in a plugin which holds control over them. Therefore, containment connector implementation is part of a plugin and the plugin can instantiate the connectors transparently. There is a problem with connecting of units of these connectors. If the containment connector is instantiated transparently, the units and the connection among them are not present in the Deployment Plan so the deployment runtime does not connect them. That means that the standard way to pass the server unit reference to the client

[2]The hybrid components have both contents. However, this is not a collision of principles, but only collision of names. In the code, this doesn't make problems.

unit during application launch cannot be used. There are several ways to address this problem.

First solution is to add a connection during the launch of the application and use the normal way to connect the connector units. The containment connectors have to be instantiated by Fractal plugin, no other part knows about them. But it is not safe to assume that all parts of the deployment runtime can handle the changed set of connections which is required for this solution to be successful.

An alternative solution is to connect the units without passing any reference, using some object registry. This approach requires an object registry accessible from both sides. The solution can be explained on a simple example of one parent and child. We call the relation "being a child of" *childhood* and a relation "being a parent of" *parenthood*. When a parent component is instantiated, the plugin finds child's name among component's properties. It instantiates client connector unit for the childhood relation (and sets it to the parent component as child) and server connector unit for the parenthood relation (and sets the parent component as a target to the unit). On the child's side mirror actions are performed creating the pair units. Server connector units on both sides are registered to the registry under a name constructed using exactly defined rules. During the second step of interconnecting of application components (Finish Launch) we can be sure that all application components are instantiated, as well as the connector units. Thus, during this step, plugin on each side can use the rules to make up the name of the server connector unit instantiated by the plugin on the other side, retrieve the reference from the registry and connect the connector units.

Another solution is to create the containment connection directly to a Deployment Plan together with special "parent" and "child" ports of components. From the view of Deployment Runtime there are ordinary ports and connections (the containment is actually a kind of connection), a plugin internally handles these ports as containment, not as business interfaces. This approach makes the plan more complicated and the connectors are not instantiated transparently. However, the units can be connected by the common mechanism for connecting components' ports. Moreover, with this approach, a connection reconfiguration mechanism can be used for containment, because there is a normal connection from the view of Deployment Runtime.

The solution with an added connection puts extraordinary requirements on the deployment runtime, which breaks the contract on the vendor boundary – the runtime should be independent of plugins. We have decided to provide support for the two letter solutions. The object registry solves the problem inside the plugin, not involving the deployment runtime. An object registry is already present since it is required for exchange of port references (it binds direct to strigified references). The solution with containment connectors added to Deployment Plan puts more requirements on a Planner but is most flexible.

Note that if an application uses containment, regardless on the chosen solution, it is not safe to start the application during the Finish Launch step as the specification allows. At the end of the Finish Launch step on a node, all required ports of all components on the node are bound but some provided ports can be

unbound yet (their users are on different nodes which have not finished launch yet). Normally, it does not constitute a problem. But in case of containment, this situation may cause an error because the containment relation is mutual. A started parent component can have a reference to a subcomponent which does not have the reference to the parent component yet. This may confuse the component model implementation. Therefore, it is strongly recommended to start the application after the global Finish Launch operation has finished in case of presence of containment.

7.3.3 Fractal-specific Deployment Metadata

All information needed for execution of a Fractal component has to be stored in the Deployment Plan. The generic Fractal launcher executes Fractal applications using metadata provided by ADL. Our system uses a metadata format based on the OMG D&C. However, if a Fractal application comes with an ADL, it will be possible to rewrite the ADL metadata to the metadata format required by the deployment framework.

The overall structure of the metadata is similar to our model. Components are defined with their interfaces and implementations, as well as connections among them. However, there are several specific features that have to be addressed.

In Fractal, composite components are also instantiated. They have knowledge about their subcomponents and parents. This can be used to set parameters to subcomponents or parents and it is useful for ordering of starting and stopping of application components. Moreover, a composite component can change the hierarchy structure during its run. Therefore, an application may need the containment to be preserved.

We have decided to provide support for two solutions to implement containment. For the first solution, the containment information has to be recorded in the Deployment Plan as a property. The most suitable place for this is the `configProperty` of a component `instance` – names of child and parent component instances can be passed as these properties. To represent a shared component, entering multiple parents is sufficent. The properties are named `parentComponent` and `childComponent`. The Fractal plugin can build the containment hierarchy using these properties.

The second supported solution realizing containment is to explicitely provide special ports and connections among them. In this case, a component has special ports, which have an interface of a Fractal component, and are marked by a property named `type` with value either `parenthood` or `childhood`, according to a relation realized by the port. These ports are connected by a standard connection. A containment connector units can be defined as components and used to connect these ports remotely.

For each containment relation, one of these representations is to be used, not both of them.

In Fractal, primitive and composite components differ in generated controllers. Type of each component has to be defined in the metadata. This information is recorded as an `execParameter` of component `implementation` named `controllers`.

It can have one of values `primitive`, `composite` and `hybrid`. If this parameter is omitted, the default value is `primitive` because an application flattened as expected by the specification contains only primitive components so this is natural. If an application needs another type of component than primitive, it has to request it explicitly.

For each component, its interfaces and implementation have to be defined so that it can be executed. This information is stored in the `realizes` part of the component `implementation`. For a Fractal component, it has to contain a `specificType` value specifying name of the class containing the actual component content code (the value is omitted in case of composite components, as they have no content code). Then, a list of `port`s has to be provided. Each `port` has to define its `name` to be identified inside the component, its `specificType` specifying name of the class defining the interface, a `provider` value to distinguish direction of the interface and an `optional` value to distinguish between the mandatory and optional interfaces. For each `port`, `exclusiveUser` and `exclusiveProvider` values have to be set to `true` as only bindings between two ports are allowed. Bindings among multiple ports have to be realized by components dedicated to communication and we suppose these components figure in the plan normally.

The `realizes` section should also contain a `label` value to let the Fractal plugin produce sane human-readable messages.

A Fractal composite component has internal ports. These ports are not present in Fractal ADL and are created automatically according to the external ports. We cannot adopt this approach because the deployment runtime is not aware of the creation of internal ports and would consider connections using nonexistent ports. Therefore the internal ports have to be present in the Deployment Plan as well. Naturally, they have to have the same definition as their external opposites, except of the direction. To distinguish the internal ports, an `execParameter` with the name `type` and value `internal` has to be added to each of these `port`s. For this parameter, also `external` value is allowed which is default. Other possible values are `parenthood` and `childhood` (see above).

The Fractal model allows defining a collection of interfaces as a component port. When an interface collection `Foo` is defined, the component has interfaces `Foo1`, `Foo2`, `Foo3` etc. The sequence is possibly unbounded. The deployment runtime has to distinguish these ports from each other, therefore, they cannot be represented by one `port` entry. On the other hand, the Fractal plugin has to create the true collection because an unbounded sequence of ports cannot be created one by one.

For each member of a port collection, which is used in a connection, a `port` has to be defined. To express its assignment to an interface collection, an `execParameter` named `memberOfCollection` whose value is the name of the collection has to be added. When instantiating such a component, the Fractal plugin merges ports with the same collection name into a collection. It has to recognize which port parameters are to be set for the collection (although the only thing besides their names they can differ is the `optional` value). Therefore, one port of the collection has to have

the parameter `pivotOfCollection` instead of `memberOfCollection`. Parameters of this port will be used for the collection.

7.4 SOFA Component Model

There are two main options how to implement deployment support for SOFA components. First, the existing SOFA deployment runtime can be used, plugins on nodes can encapsulate SOFA deployment docks. In this solution, two deployment systems would duplicate each other – the SOFA deployment docks will run on their own, bypassing our deployment runtime structure. This breaks the concept of common deployment infrastructure.

A better solution is to connect our system to SOFA as another deployment backend and implement the support for SOFA using the deployment runtime infrastructure and SOFA core API. To implement this solution, data needed for deployment of SOFA application have to be identified and their mapping to the Deployment Plan defined. The same issues as in case of Fractal are to be elaborated.

7.4.1 Component Types

The difference between monolithic and composite components is not as strong as in Fractal. Each component has its frame and microcomponents, the difference is in the content, which is realized either by monolithic code, or by subcomponents. Therefore, a composite component has always its own code, realizing the microcomponents.

7.4.2 Containment

In the SOFA component model, there are no implicit connections among components and subcomponents. That means, that a component has no reference to its subcomponents or parent, except the business interfaces. The control interfaces of contained components are not connected, therefore, the containment does not have to be elaborated as in case of Fractal.

7.4.3 SOFA-specific Deployment Metadata

Type of a component is described by a `Frame` with `Interface`s. These entities are mapped to the `realizes` and `port` sections. All contained attributes can be represented as properties. Collection of interfaces can be handled the same way as in Fractal (see section 7.3.3).

The `implementation` section is created from an `Architecture`. In case of monolithic component, name of the implementing class is copied. In case of composite components, an empty `implementation` is created and references to subcomponents are ignored (no physical references are created).

The `instance` is created from `InstanceAssemblyDescription` which is part of the assembly descriptor.

The SOFA repository stores compiled code of classes and interfaces directly in `CodeBundle` entities. This approach does not fit the Deployment Plan. These data have to be stored in classes packaged as artifacts and referenced by class names. The plugin can load them to the `CodeBundle` objects.

Each component contains a system of microcomponents. Their structure is quite complex for storing into properties. More practical solution is to pass this information in special artifacts. The description of microcomponents can be serialized into a XML file and passed as an artifact with a parameter identifying the artifact purpose. The plugin can deserialize the XML into objects, which can be passed to SOFA API.

The Deployment Runtime handles connector units as any other components and plugins have to provide a bridge to connector interfaces. In SOFA component model, connectors are first class entities, represented separately in the metadata. However, this approach does not prevent representation of connectors as components in the Deployment Plan. Each connector unit is represented as a component instance, the plugin can handle it as it needs. As application components expect to be connected to connectors (not to ordinary components as in Fractal), the connector bridge has an easier role – it does not have to create a real components from connectors.

Several types of bindings have to be considered – the ordinary client-server bindings, delegations, subsumptions and remote bindigs from a Remote Binding Map describing connections among connector units. Each of the described bindings can be represented as a connection. The only problem is to distinguish the role of a port in a binding because one component port can be for example bound to another component by a connector and bound to a subcomponent by a delegation.

A port can be used in connector, delegation or subsumption on the outside and used in delegation, subsumption or directly by the content on the inside. Delegation can be distinguished from subsumption by the direction of ports, therefore, these bindings can be collectively called *delegationOrSubsumption*. On the other hand, an external delegationOrSubsumption to a parent component has to be distinguished from an internal delegationOrSubsumption to a subcomponent. Therefore, each port has a property named `role`, which contains value either `connector` or `delegationOrSubsumption` (where `connector` is default and does not have to be present). If a port is delegated or subsumed on the inside, another port has to be defined, similarly to Fractal internal ports. Such port has the `role` set to `delegationOrSubsumptionInternal`. The plugin does not create such interface, it checks that it matches the external interface and then delegates or subsumes the external interface appropriately.

The SOFA model defines *property mapping* – each component property can be mapped to a property of one of its subcomponents. This pattern is present in the OMG Component Data Model. However, since the Deployment Plan is flat, the mapping is not present here and has to be represented another way. The mapping can be viewed as a special type of connection – connection which delegates operations

on a property. Therefore, each mapped property can be represented as two special ports and a connection between them. To distinguish these ports, they must have the property `role` set to `propertyMapping`. Moreover, they have to have a property `mappedPropertyName` set to name of the mapped property. The plugin can then handle these special ports appropriately and create the real property mapping.

8 Prototype Implementation

The prototype implementation should prove the concept of heterogeneous deployment. The goal is to create the deployment runtime independently on particular component models (as it aspires to be general). Then, maximum amount of functionality of component models have to be brought to work. The runtime has to be able to deploy applications to a distributed target environment using connectors.

8.1 Overview of the Implementation

The implementation contains only one component model plugin (reasons are explained in section 5.2.2) and therefore it is not yet ready to execute a real heterogeneous application. However, the system is general and support for one classical component model is fully implemented. Since components are executed separately and communication among them is provided exclusively by connectors and since we assume that the heterogeneity will be transparently hidden to the connectors, the implementation can be still considered as proof of the concept of heterogeneous deployment.

The deployment runtime itself is a minimalistic implementation of the OMG D&C specification – the focus has been on proving the concept of heterogeneous deployment. It is a console application which works only with a static domain.

For the implementation java language is used. This choice is given by the other projects which collaborate with this work.

The planner is under development and not prepared yet so it is not easy to get the input for the deployment runtime. Therefore, input data are created manually. Also demonstrational applications are written manually, because we cannot create in this way plans and connectors for a large application.

The project is divided into five basic parts:

- **Metadata Repository** contains metadata repository implementation and generation facility.
- **API** contains basic interfaces of runtime entities.
- **Runtime** contains deployment runtime implementation itself.
- **Plugins** contains implementation of plugins.

- **Applications** contains demo applications.

The API depends on the Metadata Repository because the metadata are used in the interfaces. The Runtime depends on the Metadata Repository and API. The Plugins depend on all three runtime parts and the Applications are independent.

8.2 Metadata

Before we can implement the runtime itself, we have to define exact format of the data used by the system. The deployment runtime operates on the Data part of the Platform Specific Model for Heterogeneous Deployment. A support for storing, exchanging and browsing of these data has to be implemented.

8.2.1 Metadata Repository Requirements

A physical representation of the metadata and their mapping into the memory are required for storing and loading the data. A repository is required for browsing and exchanging the data. A repository is expected to be a set of java classes representing the data, allowing browsing them, with the ability to load them from a persistent form and to store them back. It has to be created in a reasonable time and has to be maintainable in future. There are tools generating a data repository on basis of a data description. Using one of these tools fits the mentioned requirements better than writing a repository by hand (confrontation of hand-written and generated repositories is elaborated in [9]).

We have several requirements on the tool and the resulting repository:

- **Ease of Use**: The repository should be easy to use – without many unnecessary indirections, long unintuitive names etc.

- **Java 1.5**: We strongly prefer java 1.5 repository code, which relates to the ease-of-use requirement, but which mainly allows to write more readable and safe code of the rest of the system.

- **XML**: We prefer XML as the persistent data format.

- **Maintainability**: The repository should be reasonably maintainable according to future changes in the OMG D&C. The specification development is in progress and some changes in the schema will appear. Their projection to the repository should be possible with a reasonable effort, without hand-rewriting the generated code etc.

- **Maintained Tool**: Last but not least, the tool should be live and maintained project, from obvious reasons.

8.2.2 Repository Generators

ModFact [14] is a tool based on the Meta-Object Facility (MOF) [17] which is a meta-modeling specification provided by OMG. The ModFact tool provides generation of java repository from a UML metamodel. This tool is used in [9]. The tool, however, fails to meet majority of the requirements. The resulting repository is not very intuitive, its structure is quite complicated. Java 1.5 is not supported by the tool. Generally, the tool is unmaintained (last update of project web page was in 2004) and not very robust. On the other hand, the XML persistence format requirement is fulfilled. Future changes in the OMG D&C specification have to be projected to the UML model and the repository has to be regenerated which meets the easy maintenance requirement.

Eclipse Modeling Framework (EMF) [7] is a tool providing a java code generation from XMI metamodel. It actually implements the MOF specification. The resulting code is also not very intuitive. Moreover, it provides many features, which we do not need, and which make the code even more complex and less intuitive. With the support for the java 1.5 it is also not good – the project seems to be refusing 1.5 features to allow maximal flexibility and not to generate version mismatch problems with other parts of the Eclipse framework. Changes in the OMG D&C specification have to be projected to the XMI and the repository has to be regenerated. As the XMI is typically generated from a UML model, it is the same case as in ModFact. The project is live and maintained.

The EMF tool seems to be more suitable than the MOF tool but it has only a few advantages (mainly the liveness of the project).

Java Architecture for XML Binding (JAXB) [10] is a product of Sun, integrated in Java Web Services. It provides java repository generation from a XSD schema. The java repository represents a XML data conforming to the XSD schema. The tool produces very clear, simple and intuitive repository. It generates java 1.5 code. To project future OMG D&C changes to the repository, the XSD schema has to be adjusted and the repository regenerated. It is not as intuitive as changing the UML model but can be also done quite easily. The JAXB project is live and maintained, without any doubt.

Considering the described alternatives in the light of the introduced requirements, we have decided to build the repository on top of the JAXB backend. The repository is definitely most intuitive, clean and easy to use. It uses pure java 1.5. The changes in the OMG D&C can be reflected to the repository with effort comparable to using the other tools. The XML data format is common to all considered alternatives. The JAXB tool is well-supported, not loosing there either. Apparently, the JAXB tool meets the requirements the best.

8.2.3 Metadata Repository Implementation

Metamodel input for JAXB has a XSD form. The OMG D&C defines MDA transformations coming to the XSD schema for CORBA Component Model, providing the resulting XSD. We have considered the option to use this XSD as a clue in process of creating the model-independent XSD. The CORBA Component Model is not within the scope of this work but the provided XSD represents the intent of the specification authors how such code can look like, which is valuable information. However, we have identified following issues with the CCM schema.

The XSD proposed by OMG validates great amount of nonsense XML files. The problem can be demonstrated on a little example. Considering a simplified element `Node`, which contains a name of type `String` and a set of `connection`s of type `Interconnect`, in the XSD from OMG D&C specification it would be defined as follows:

```
<xsd:complexType name="Node">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="name" type="String"/>
        <xsd:element name="connection" type="Interconnect"/>
    </xsd:choice>
</xsd:complexType>
```

This schema validates obscure nodes either without name or with multiple names. This approach does not allow restricting the number of connections (e.g. cannot forbid standalone, not interconnected node). It allows for arbitrary ordering of the subelements, which is general, but does not contribute to the XML readability. Another problem is, that all boolean elements are defined to be of a `String` type which allows passing nonsense values.

Considering these problems, we have decided to write our own, independent XSD schema according to the Platform Specific Model for Heterogeneous Deployment. The above example is rewritten as follows:

```
<xsd:complexType name="Node">
    <xsd:sequence>
        <xsd:element name="name" type="String"/>
        <xsd:element name="connection" type="Interconnect"
            minOccurs=0 maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
```

We have also specified all the boolean types as true booleans. These modifications solve all problems described above. The resulting schema validates smaller set of XML files, but their format is not influenced.

However, another problem has surfaced with element referencing. Using the previous example, the `connection` is a standalone entity, which is not contained in the `Node`, but is defined at the same level as the `Node` and should be only referenced from here. In the XSD from the OMG specification, the connection is of type

`Interconnect`, which is a complex type defining the connection, not a reference. This means, the XSD does not differentiate elements themselves and references to elements.

Complex types defined in the XSD (including the `Interconnect` from the example) allow an attribute `idref`, which can point to another element. There are several problems with that. Instead of the reference, we have a whole `Interconnect` element which can reference another one. This is an superfluous indirection, effectively doubling number of indirection steps when browsing the repository. Moreover, the `Interconnect` element can contain both data and reference to another element, and is hard to decide, which of the two data sets is valid. Finally, there is no type checking of the referenced element.

Many different options to solve this problem were considered, several of them implemented. The cleanest solution is to explicitly distinguish data from references. Where a reference is present in the model, an element of `xsd:IDREF` type is used with JAXB specific annotations to provide type control. We have clearly separated the data definition from the data reference. Note that this modification changes structure of the validated XML files.

The JAXB output is very clean and usable, yet because it is a beta version, it contains wrong annotations in case of the element references with added type control. Code is generated with the right types but the types are not projected to the annotations. This makes the code unusable for loading and storing the XML files. We have decided to create a tool which corrects this problem. This can be done easily – the java sources generated by the JAXB tool are passed to an *annotator* tool. For each reference element, the specific type is read from the class member definition and the general type in the annotation is rewritten by the specific type. Then the code behaves correctly.

8.3 Node Manager

The overall structure of the Node Manager implementation including Component Model Plugins is presented in figure 11. The Node Manager uses an Environment Manager to instantiate execution environments. In the environments component model plugins are loaded. The plugins then maintain components of deployed applications. The implementation adds an entity called Component Model Plugin Manager to the architecture to separate the code responsible for lifecycle of component model plugins. This entity is used by execution environments.

Instantiated application components are by default placed in an automatically started *common* environment. A new execution environment can be started on demand, either for an application, or even for one component. For that purpose, the Environment Manager is passed to the Node Applications because they are aware of component requests. Therefore the Environment Manager, one per node, is used mainly by the Node Applications.

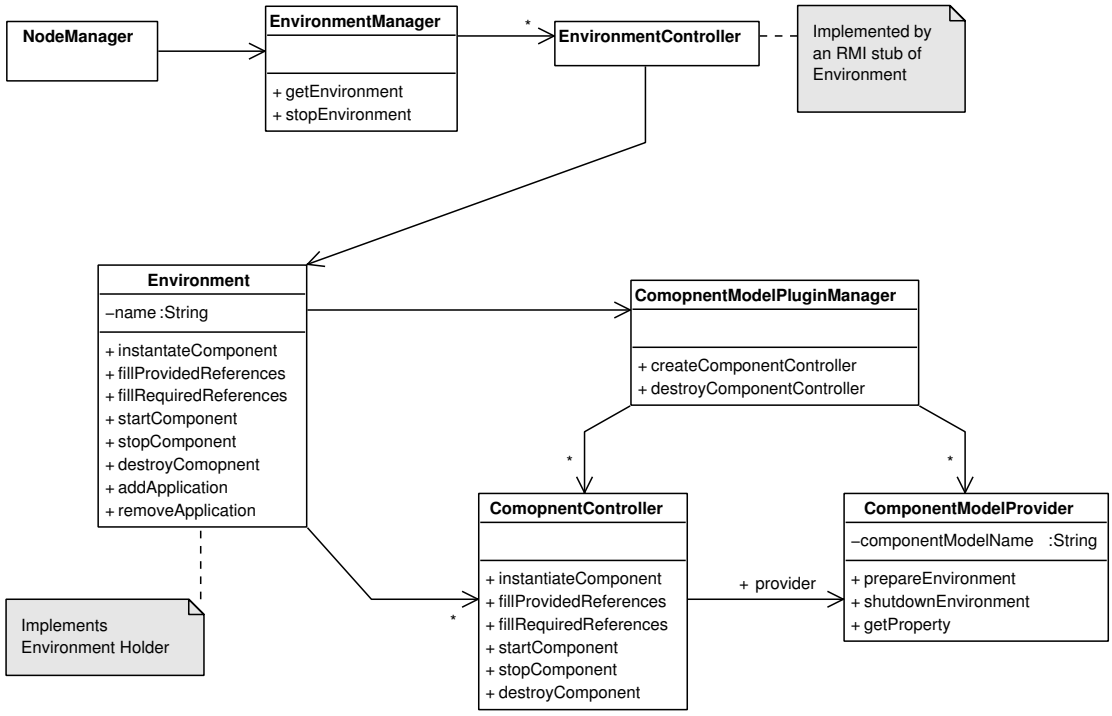


Figure 11: Node Manager Implementation

8.3.1 Execution Environments

We have implemented one type of environment, representing a Java Virtual Machine dedicated to host the deployed applications.

The execution environment purpose is to maintain components of deployed applications. An environment represents another JVM, so the components cannot be identified by references. Therefore a stringified identifier is created for each instantiated component.

The interface of an execution environment (both Environment Controller and Environment Holder) provides these operations:

- **instantiateComponent**: Takes all information needed to instantiate a component, instantiates it, and creates the identifier.
- **fillProvidedReferences**: Retrieves provided references from a component.
- **fillRequiredReferences**: Sets required references to a component.
- **startComponent, stopComponent**: Changes lifecycle state of a component.
- **destroyComponent**: Destroys a component.
- **addApplication, removeApplication**: The execution environment needs to know how the components belong to the applications and may need to track

some application-related information. These methods provide the information about the added and removed applications.

An execution environment has two parts: Environment Holder (controlling the environment from inside) and Environment Controller (controlling the environment from outside, from the Node Manager). We have created an Environment Holder which registers itself to an RMI registry during the environment startup. Stub of this object, generated by the RMI, serves as the Environment Controller.

Both Node Manager and execution environments launched by it run on the same node. Each Node Manager (more precisely the Environment Manager) starts its own RMI registry used to connect to execution environments so that it does not perform a network communication for the local connection.

The execution environment only delegates requests to plugins. When instantiating a component, a property identifying component model is read to recognize which plugin has to be used. The rest of the work is delegated to the plugin.

We have separated the work around loading plugins and loading components through the plugins off to a special object called Plugin Manager. This object is in fact part of the execution environment but its functionality is described in section 8.3.3 because it is tied to the plugin system described in next section.

8.3.2 Component Model Plugins

The implementation follows the requirements set forth in section 6.1.4.

A plugin providing support for a component model has two basic objectives. The first objective is to adjust environment to satisfy all requirements of the component model. The second objective is to manage components built on the component model. We have decided to divide support for these two objectives into separated objects – Component Model Provider and Component Controller.

This decision is influenced mainly by classloading issues. Briefly introduced, for improved robustness and performance, the plugin should not accumulate classpaths of all applications. Therefore, the plugin part providing the general support related only to the component model and the part tied to the application have to be separated. This issue is elaborated in more detail in section 8.5.3.

The first, common part of plugin architecture is the Component Model Provider. It has only one instance in the execution environment, created upon loading of plugin. Its interface is simple:

- **componentModelName**: Returns name of the implemented component model.
- **prepareEnvironment**: Prepares the environment to provide everything required by the component model. This method is to be called after the plugin is loaded to the system.

- **shutdownEnvironment**: Cleans the environment from everything created by `prepareEnvironment`. This method is to be called before the plugin is unloaded from the system.

- **getProperty**: Gives to the Component Controller part of the plugin anything from node-wide component model properties.

The second part of plugin architecture is the Component Controller. Each Component Controller instance manages one component. Thus, this part will have as many instances as many components built on this component model will run. The controller is instantiated at the same moment as the controlled component. An execution environment uses the Component Controller as a handle of a component – it keeps a one-to-one relation between Component Controllers and the component identifiers provided to the Node Manager. When a request from the Node Manager comes to an environment with an identifier of a component, the corresponding Component Controller is found and the request is passed to it. Therefore, the interface of the Component Controller corresponds to the environment interface:

- **setProvider**: Sets a Component Model Provider reference. Information and objects required by the component model can be retrieved from this object by the `getProperty` method.

- **instantiateComponent**: Instantiates a component. This method goes together with creation of the controller, as the only purpose of the controller is to manage the component.

- **fillProvidedReferences**: Retrieves provided port references from the managed component.

- **fillRequiredReferences**: Sets required port references to the managed component.

- **startComponent, stopComponent**: Changes lifecycle state of the managed component.

- **destroyComponent**: Destroys the managed component. This method goes together with destruction of the controller.

To keep the boundary between a plugin and the deployment runtime clear, dependencies to the other parts of the system were minimalized. The Component Controller manages the component according to requirements and conventions of the particular component model. It can use the standard deployment information passed from the deployment runtime (which is well-defined), extending component-model-related information, which traversed through the runtime in properties (which are defined by the plugin vendor) and the information retrieved from the Component Model Provider (which is implemented by the plugin vendor).

Data format of a component model plugin is defined as a single jar file containing all plugin classes and all classes implementing the component model. Each plugin has to contain in the jar top-level directory a file named `plugin.inf`, which is used to determine which classes are to be loaded. The file contains name of the supported component model, name of a Component Model Provider implementation class and name of a Component Controller implementation class. The `plugin.inf` file has a format of a common config file – each line contains either a name-value pair, or a comment beginning with `#`, or is empty. Properties named `model`, `provider` and `controller` have to be present.

8.3.3 Component Model Plugin Manager

Plugin Manager is a separated part of the execution environment. Its purpose is to manage lifecycle of component model plugins. It manages loading and destroying of both Component Model Providers and Component Controllers. It takes care of proper settings of classpaths and codebase, which is not an easy task, and is elaborated in section 8.5.3.

The implementation of Plugin Manager loads plugins on demand and supports reloading of new plugin versions and unloading of unused plugins.

Plugin is loaded when it is needed to instantiate a component. If a new version of a loaded plugin is found, the plugin is marked as expired. It is not considered safe to run multiple versions of the same plugin at once. Therefore, the manager does nothing with the expired plugin while any components are running under it. As soon as the plugin temporarily does not host any component, the old version is unloaded and the new one is loaded. If there is no component using the plugin for a long time, the plugin is unloaded. The expiration time is configurable from a configuration file.

8.4 Communication among Components

To allow communication among deployed components, they have to be interconnected by remotely passed references. To perform the communication, connectors are used and have to be implemented. Their behavior has to be similar to behavior of generated connectors, because they are to be replaced in future. A connection reconfiguration functionality has to be provided to connectors.

8.4.1 Interconnecting the Components

Component port references have to be stringified so that they can be passed remotely. Components implementing connector units need to have an option of passing multiple references for one port.

The deployment runtime provides a Local Port Registry, which can be used to translate native references of components to a stringified references and back. The registry is a singleton object existing once per a JVM. A reference can be registered

in the registry and bound to a string name. This name can be then passed through remote calls without damage and then the object can be looked up in the registry.

A problem can appear when the registries on different nodes contain references bound to the same name. In this case, the name can be passed to another JVM and there an unexpected object can be found in the registry. A global uniqueness of local references is left on an object invoking the registration. The deployment runtime has enough information to be able to create the unique names.

Remote references can be stringified using a global RMI registry provided by the deployment runtime. When using the remote registry, the stringified reference contains both address of the registry and name to which the object is bound.

A connector unit may need to create both local and remote references. To be able to create the remote reference, address of the global RMI registry is given to each unit. The Local Port Registry is a part of the runtime and to keep the vendor boundary between connectors and plugins clear, it should not be used by connectors. Connectors have their own local registry providing a similar service. Our plugin provides a bridge between these two registries (keeps a one-to-one mapping between them).

8.4.2 Connectors

The Planner generates connectors and adds them to a Deployment Plan. The deployment runtime then is not aware of the connectors and sees them as any other components. In the end, the component model plugin provides a bridge from connector interfaces to runtime data format.

The connectors implement interfaces defined in the connector project [5]. The behavior of connectors has to be simulated from the external view of the deployment runtime, it does not need to follow the internal hierarchical structure of elements. Therefore, each connector unit has only a monolithic implementation of its top-level element.

Each connector element may need some properties or references to be set. In the original architecture, they are passed to a constructor by a superior element or by a Connector Manager. Therefore, the original connector interfaces do not allow setting these properties from outside which is necessary to configure the standalone connectors from deployment plan. The connectors generated to a deployment plan act as any other components, and it may be difficult to pass anything to the constructor. To solve this problem, we have added the necessary interfaces which allow setting these properties to a unit's top-level element. These interfaces are to be added to the connectors project.

The connectors have to instantiate new connectors for references passed through a remote call. We have decided to employ a Dock Connector Manager for this purpose. However, the original interface of the manager is tied to the connector generation facility which is not used in our system. Therefore, we have decided not to use the original universal Dock Connector Manager. We have covered both Dock Connector Manager interface and implementation. Our interface is simple and allows only to instantiate a new connector for a given interface, the implementation

contains a hard-coded facility to instantiate one of the hand-written connectors.

The hand-written connectors employ the connection reconfiguration feature. Each connector tests if it is deployed locally and in this case tries to reconfigure the connection to reject itself from the data path.

The connectors handle multiple redirections optimally. This feature can be explained on an example of three components `A`, `B` and `C` connected `A` to `B` and `B` to `C` remotely through connectors. If each of these components is deployed to another node and the component `A` asks component `B` for reference to `C`, a direct connector from `A` to `C` is returned instead of an indirect connector from `A` to `B` to `C`. If the components `A` and `C` are deployed to the same node, even a direct reference is returned, without any needless connector. This holds regardless of how many nodes were passed by the call.

There is a requirement to keep the system mergeable with the connector project. The hand-written connectors are replaceable by the original connectors, because they adhere to the interfaces and external behavior. The changed connector manager does not make problems either. Its interface is used only from inside of the connectors. The manager is instantiated by a component model plugin (because the dock connector manager is plugin-specific) and is passed to connectors. During the application run, the connectors use this interface to instantiate another connectors on demand. Therefore, when the connectors are replaced, the connector manager can be replaced as well, together with its interface[3]. Using the Global Connector Manager is out of scope of this thesis.

8.4.3 Connection Reconfiguration

We have decided to provide to components an option to reconfigure connections containing their ports through a callback. There are several issues with this approach: callback interfaces and their usage have to be defined, connection reconfiguration running concurrently with the natural connection during application launch has to be addressed and impact of the solution on adherence to the OMG D&C has to be minimal.

The standard interconnecting process is initiated by a Domain Application. In the first step it collects references to provided ports of all components, in the second step it sets the references to required ports of all components[4]. During each of these steps, the call has to traverse from the Domain Application to the component itself. The call runs over these entities:

[3]In fact, the modified Dock Connector Manager interface is used from one more place – from the plugin, to instantiate containment connectors. This prevents code duplicity in the system. When merging the projects, the old connector manager code should be copied to the place where it is used and after that it can be replaced. If the new connector manager will be able to create containment connectors, it can be used instead from that place.

[4]The first step is in fact initiated by a Domain Application Manager. However, internally (according to the specification) the Domain Application Manager creates a Domain Application and lets it to collect the references. Thus, both steps are actually performed by the Domain Application.

Domain Application → Node Application → Execution Environment → Component Controller (Plugin) → Component

A reconfiguration callback initiated by a component has to run up to the Domain Application because the connection may have endpoints on multiple nodes and the Domain Application is the only entity on the path that has references to the other nodes. Therefore, all these entities have to be employed to provide the callback to components. The callback then runs upwards to the Domain Application, which repeats the two connection steps for the particular connection – gets the provided ports again and resets them to the required ports.

Interfaces realizing the reconfiguration are presented in figure 12.

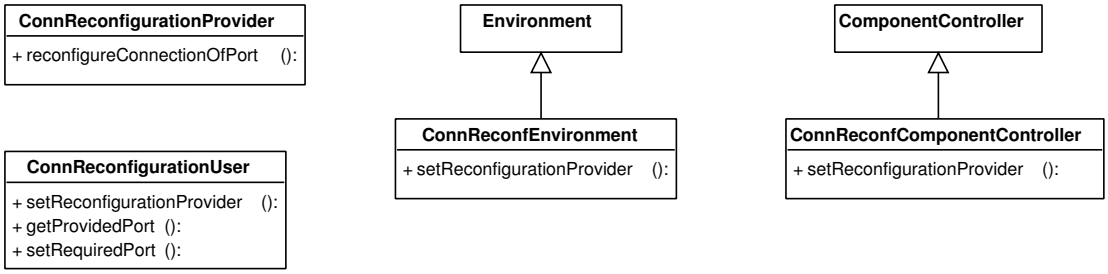


Figure 12: Connection Reconfiguration Interfaces

The connection reconfiguration callback is realized by two interfaces. The Connection Reconfiguration Provider interface realizes the callback object and has the only method – it reconfigures a given connection. The Connection Reconfiguration User interface realizes the user of the callback. It has three methods – the first sets the reconfiguration provider, the second gets a provided port contained in a reconfigured connection and the third sets a required port contained in a reconfigured connection. The provider set by the first method is used to invoke the reconfiguration, the other two methods are used by the provider to perform the reconfiguration.

The callback sequence has to be connected over all entities on the path from a Domain Application to a Component. However, the callback itself does not have to perform such many steps. In the implementation, a Domain Application provides the reconfiguration callback to a Node Application and the Node Application provides it directly to a Component Controller. This makes the callback faster as it is not passed through the Execution Environment entity[5]. To achieve this, the interface of Execution Environment has been extended to allow passing of a reconfiguration provider further to Component Controllers. The interface of Component Controller has been also extended to allow setting of the reconfiguration provider. The getter for provided ports and setter for required ports present in the Connection Reconfiguration User interface are not needed in these two entities, because the Execution Environment will not be involved in the callback path and the Component Controller has these features already.

[5]As the environment only redirects requests to plugins, it does not need to track the callbacks. Therefore, it can pass a callback provider to plugins and reject itself from the callback path.

To let the connection reconfiguration work, an implementation of Component Controller has to provide a bridge to reconfiguration interfaces of connectors.

Each entity on the path from a Domain Application to a Component is responsible for creating the next entity on the path. Therefore, each entity can set the reconfiguration provider to the next entity during its instantiation. This keeps all callback paths connected at any time they are needed.

Components may request connection reconfiguration as soon as their required ports are bound. This happens during the second interconnecting step (Finish Launch) and the application does not have to be fully interconnected yet. Therefore, a reconfiguration request can come for a not-yet-configured connection.

It is dangerous to delay somehow the request because of a risk of deadlocks. Therefore, the implementation always handles the reconfiguration request immediately. During the Finish Launch step, each Node Application creates a queue of application connections and then removes one by one from the queue and connects it. If a reconfiguration request comes for a connection that is still in the queue, it is removed, otherwise it would be later connected back to the state before the reconfiguration. From the view of a component, both natural connection and connection reconfiguration is a simple bind request so replacement of one by the other does not constitute any problem.

We have added two interfaces to the deployment runtime and extended another two. The new interfaces break compliance with the OMG D&C. Therefore, we have separated all the reconfiguration-related functionality to special classes. The interfaces are extended by implementing subinterfaces, implementation of the connection reconfiguration functionality is placed to subclasses of the original entities. This also contributes to code readability.

We have implemented a Runtime Entity Factory which creates entities of the deployment runtime. According to a configuration file, either objects with or without the reconfiguration functionality are instantiated. Thus, the system is still compliant with the specification and, moreover, a reconfiguration extension is provided.

During the connection reconfiguration process, each involved entity is checked for the reconfiguration support. The reconfiguration is performed, if all entities on the reconfiguration path provide this functionality.

8.5 Technical Issues

8.5.1 Starting the Deployment Runtime

The deployment runtime, according to the OMG D&C, has the following standalone parts: Target Manager, Execution Manager and Node Managers. In the implementation each of these parts runs in its own JVM. For each of these entities, besides the implementing class, a class with the name suffix `Server` is present. This class holds the main thread of the JVM. When started, it reads a configuration file, prepares everything needed and creates the manager itself. Implementation of the managers contains only the business code.

The deployment runtime parts are interconnected through java RMI. The Target Manager starts an RMI registry and registers itself. The other parts get address of the registry from a configuration file. During their start, they register themselves to this registry as well. As their names are well known (defined in section 7.2), they can find each other. In fact, the only one who searches the registry is the Execution Manager – the Execution Manager commits resources on the Target Manager and the Execution Manager initiates starting of applications on the Node Managers.

One technical problem is related to the usage of RMI. The java RMI registry forbids registration of an object from a remote host due to security reasons. But for the system it is necessary to register the managers remotely. Because security is not within the scope of this work, we have solved this problem by reimplementation of the registry. The implementation replaces the java RMI registry transparently and allows the remote binds (without solving the security issues). Moreover, this registry provides an option of a blocking lookup – the method passively waits till the searched object is registered and then returns it. It is useful during the system startup as the parts of the runtime can wait for each other passively.

8.5.2 Artifact Management

Each application consists of artifacts, that are supplied as jar files, accessible from addresses given in a Deployment Plan. Each component then requires a subset of these artifacts.

The artifacts are managed by an Artifact Manager which is instantiated per Node Application. During the Start Launch operation, for each component, Artifact Manager is called to retrieve the component artifacts. It downloads all the artifacts from their target locations listed in the Deployment Plan and stores them to a temporary directory on a local drive. The manager ensures, that each artifact (possibly shared) is downloaded only once. It also maintains dependencies among components and artifacts. When a component is being instantiated, the Artifact Manager is asked for all artifacts, required for the component. The manager returns the local paths to all artifacts which are then used to load the component.

After an application finishes, the Artifact Manager is called to clean its artifacts. It deletes all the artifacts stored on the local drive.

8.5.3 Code Management

Applications and plugins are dynamically loaded by the deployment runtime. Each component of one heterogeneous application can demand different plugin. On the other hand, under each plugin can run components of different applications. A system of classloaders providing a necessary support is required. There are several requirements on such system.

Most general requirement is to avoid superfluous number of classloaders to keep memory and time requirements in reasonable limits. Endless accumulation of classpaths added to one classloader should be avoided for the same reason (we do not have a classloader enabling removal of classpaths).

Different components of one application deployed to the same node have to be loaded by the same classloader to be compatible with each other. Components of one application deployed on different nodes are connected through connectors using RMI, and their compatibility is reached by a proper set up of codebase.

On the other hand, components of different applications deployed to the same node have to be loaded by different classloaders, otherwise classpaths of all applications launched during the node's life will accumulate.

There are similar conditions with plugins. In case of a common classloader, classpaths of all plugins accumulate. Moreover, reloading of a plugin in case of installation of a new version can be done only by changing plugin classloader so this feature requires separate classloaders.

A decision implied by the requirements is to use one classloader per application and one classloader per plugin.

Number of loaded plugins (and used classloaders) is expected to be quite small. Number of applications can be larger but in comparison with a deployment runtime overhead for launching an application, the overhead of one classloader is insignificant.

Plugin classloaders have in their classpath only classes of the plugin, it is not growing. To instantiate a component, an application classloader needs classpaths to all component artifacts and to a plugin needed by the component. So the classpath of application classloaders grows but contains only classes of the application and of the needed plugins. It stops growing when a whole application is instantiated and it is discarded at the application finish.

There are also negatives of this approach. The two plugin parts are loaded by different classloaders – Component Model Provider is loaded by a plugin classloader and Component Controllers are loaded by an application classloader. Therefore, these parts can communicate to each other only using interfaces loaded by a common parent loader – standard java interfaces, deployment runtime interfaces and connector interfaces. In other words, it is impossible for a plugin developer to instantiate an object in a Component Model Provider, pass it to a Component Controller and use it there, unless it has an interface loaded by the parent loader.

This is quite complex problem to solve. The application possibly uses several plugins, and possibly multiple applications are running under one plugin. Therefore there is no way to make one classloader parent of the other. The only way to make classes of the plugin loaded in both parts compatible is to create a common parent loader which loads at least the interfaces needed to be passed from the Component Model Provider to the Component Controller. With this approach, we have to abandon the requirement not to accumulate classpaths endlessly because the common parent has to accumulate classes from all plugins. Also reloading of plugin versions is problematic in that case. We have decided not to make these compromises until they are really necessary.

If applications use different versions of the same component, name clashes between different classes may occur. If this occurs in different applications, the classes are loaded by different classloaders which solve the problem. However, the

system is not able to execute a single application which uses different versions of one class. The thesis [9] proposes a solution based on byte code manipulations, which is quite complex and is outside the scope of this thesis.

8.5.4 Logging

We have employed a log4j logging system [13] to provide logging functionality because it is easy to use and highly configurable without code rebuilding. Each entity in the deployment runtime has its own, local instance of the log. Moreover, the domain central log is also provided.

For the Execution Manager, the local log is sufficient because it is in fact also central within the scope of a domain. The Node Managers have, next to the local log, an access to the central log provided by the Execution Manager. The implementation of the central log is a wrapper over the log4j system providing remote access to the log. Node Managers use the local logs for usual logging and the central log for printing messages relevant to a domain, e.g. changes in the domain, fatal errors etc.

8.5.5 Requirements on Plugins

To add support for a new component model to the deployment runtime, one has to do two things. First, component-model-specific metadata needed for deployment have to be identified and their storage to the Deployment Plan has to be defined. Second, a component model plugin has to be implemented.

The metadata have to fit the general Deployment Plan and each component-model-specific piece of information has to be stored in a property. Creating a Deployment Plan according to these definitions is not a responsibility of the deployment runtime, it has to be a product of previous deployment phases.

Requirements on component model plugins are dispersed through this thesis, a summary is provided below.

A plugin has to implement two interfaces – the Component Model Provider and the Component Controller. A file `plugin.inf` listing the implementation classes has to be added (the plugin architecture is described in section 8.3.2). These parts have to take into account that they will be loaded by different classloaders. A communication between them is possible but only standard interfaces have to be used, no plugin-specific will work (reasons for that are explained in section 8.5.3).

A plugin has to implement a bridge to connector interfaces. That typically means, it has to be able to create a component from an implementation of connector interfaces (the connector should be seen as a natural component by application components). Reasons for this requirement are described in section 6.3. In addition to that, the Component Model Provider has to instantiate a component-model-specific Dock Connector Manager.

Natural direct port references used by components have to be translated to a stringified form and back. The Local Port Registry can be used for that. This requirement is presented in section 6.2.1 and technically described in section 8.4.1.

Optionally a connection reconfiguration feature can be provided. This can

be done by implementing of an extended interface of the Component Controller. Through this interface, a Reconfiguration Provider is set to the Component Controller which has to provide a bridge between this provider and connector reconfiguration interface. Technical details are described in section 8.4.3.

8.5.6 Requirements on Deployment Plan

We have provided an XSD file defining format of a Deployment Plan. Requirements on Deployment Plan content given by the deployment runtime implementation are described below.

The `DeploymentPlan` has to contain a `UUID` value to identify the plan and to create names of applications launched from the plan. Also a `label` should be provided to let the deployment runtime print clear messages.

For each `artifact` defined in the plan a `name` has to be provided for its identification. Then a list of `location`s in an URL format[6] is required for retrieval of the artifact. For the `file` URLs, the path can be either absolute or relative to the location of the plan.

For component `implementation`s, `name` value is required as an identifier. A list of required `artifact`s for the component has to be provided. A `realizes` section defining the component interface has to contain list of component ports with all values set. Implementation's `execParameter`s are there to contain a component-model-specific data. However, several values are recognized by the system. Each component has to define an execution parameter named `component-model`, containing a name of the component model used by the component. Optionally a parameter named `environment` can be added, with a string value `common`, `application` or `component`. This parameter requests an execution environment in which the component is to be placed. The `common` value is default and the component is placed to an environment common for all applications. The `application` value requests a special environment for the application, all components of the application with this value are placed in it. In case of `component` value, a special environment for the only component is created.

Each `instance` has to contain a `name` for its identification and a `node` name to identify placement of the component. An `implementation` of the instance has to be referenced. Instance's `configProperty` elements can provide component-model-specific data.

Each `connection` element has to contain a unique `name` and a list of `internalEndpoint`s. Each endpoint has three mandatory values: `instance` referencing a component, `portName` identifying port of the component and `provider` distinguishing direction of the port.

[6] The implementation in this moment supports only `file` and `http` urls.

8.6 Component Model Plugins

8.6.1 Fractal

We have implemented support for the Fractal component model. Several issues are to be addressed together with the description of the implementation.

General Issues: The Fractal component model is rather a specification and its implementation has to be chosen. We use Julia [11], which is a major implementation, and is compliant with Fractal at the highest level.

To create a full-featured component using Julia, one has to implement only a content of the component, thus the business code. The Julia takes this content implementation and creates a component by generating a component frame. This frame implements all standard component interfaces, including all controllers. Next to the business code, the content implementation can contain also an implementation of some controllers, the generated frame then delegates calls to it.

We have decided not to use the Fractal RMI. In the system, connectors are used to achieve transparent distribution of components and using Fractal RMI would bring a mixture of two solutions of the same problem. Moreover, it would employ a middleware platform which is not fully under our control.

Fractal Component Model Provider: The Component Model Provider part of a plugin has to adjust environment to provide everything required by the component model. The implementation sets system properties instructing Fractal which implementation to use and how to configure it. The Fractal RMI registry is started on its default port to allow applications use it internally (although it is not recommended). Finally, a Fractal Dock Connector Manager is started.

Fractal Component Controller: The Component Controller part of a plugin manages a single component. The implementation uses a Julia dynamic loader to instantiate the component. Two special controllers with their implementation are added to each component. They encapsulate data translation functionality directly to the component so it can communicate using the deployment runtime data format, regardless of its content (Fractal or connector). This approach makes it possible to use some content's interfaces (e.g. connector's configuration interfaces) before the component starts, which is crucial, as Fractal forbids using component business interfaces when the component is not running. It also makes the Component Controller's code cleaner because the data translation functionality is separated to another classes.

The first added controller is a Port Controller. It implements a port binding functionality including connection reconfiguration. The controller provides binding methods using the deployment runtime data format. In case of Fractal content, a standard Binding Controller is used, provided references are stringified using the

Local Port Registry and required references unstringified back. In case of connector content, a bridge for binding and connection reconfiguration interfaces of the connector is provided. Local references are copied from the connector local registry to the Local Port Registry and back, the other references are used as they are (connectors are using a similar reference data format with a bundle of strings). Note that also component contents providing their own implementation of the Port Controller interface are supported, method invocations are then delegated to this implementation. This can be used to implement a component profitting from knowledge of the deployment runtime data format. However, this is an additional functionality, which is not currently used.

The second added controller is an Element Controller. It is applicable only to connectors. It allows setting of properties needed by a connector unit.

After instantiation of a component, its Name Controller is used to set its name and then execution properties are processed. The properties contain component attributes and containment information. The attributes are set to the component through its Attribute Controller. The containment issue is more complicated and is addressed in the next paragraph. Component Ports are bound through component's Port Controller and component lifecycle is managed through component's Lifecycle Controller.

Containment: Remote containment is realized by containment connectors. The containment connector implements common connector interfaces and behave similarly to the other implemented connectors. We have decided to provide two solutions of their instantiation and interconnection. In case of transparent instantiation, their units have to be connected without passing a reference from one to the other. An object registry and a rule to construct their names have to be used. In case of explicit storage of containment connectors in the Deployment Plan, special ports have to be handled.

To connect units of a containment connector which is not stored in the Deployment Plan, two registries are used – an RMI registry and a Local Port Registry. If the server unit reference is found by the client unit in the local registry, the containment is connected directly, without the connector. In the other case, the RMI reference is used to connect the units. Name of each unit has the following form:

```
<application name>.<client component instance name>_
<server component instance name>_
{childhood|parenthood}_{server|client}
```

This name is unique and can be built on both sides (on each side, the instance name of the other component comes as the property defining containment, the other parts are known naturally).

When containment information is present in a Deployment Plan as a connection, the special component ports are handled appropriately. The containment connector also cannot be fully handled as any other connector. A connector bridge implemented in a plugin creates a component frame to a connector implementation.

However, the containment connector has to provide a special component interface implementation, which forwards all calls to a target component, because it serves as a proxy for component interface. Therefore, a simple wrapper is created for a containment connector unit. The wrapper is a connector unit which creates internally a containment connector unit and delegates all connector interfaces to it. The plugin creates a component frame to the wrapper and communicates to the unit through the wrapper. But the wrapper provides to components a reference to the contained unit with the special component interface, not to itself.

A containment connector needs to create new containment connectors when a component reference is passed through the connector (e.g. the component asks its subcomponents for subcomponents). When the deployment runtime is merged with the connector project, the new Dock Connector Manager implementation has to be able to instantiate such connectors[7], or the plugin has to keep giving to containment connectors the current implementation of the connector manager.

In a containment connector implementation are present methods removing a component from a list (e.g. when removing a subcomponent). However, the component to remove can be a connector and in the list can be another connector to the same component. Therefore, the connector searches the component in the list according to the component's name.

The presented approach provides a distributed containment transparently to components, except one thing. When the Julia Binding Controller binds two external ports of two components, it checks that they have the same parent. This check can fail because parents can be either different (different proxy objects for the same component) or may not be present at all (the Deployment Plan is flattened). Therefore, Julia is configured not to perform this check.

8.6.2 SOFA

The component model plugin for SOFA component model is not implemented for reasons explained in section 5.2.2.

8.7 Demo Applications

We have implemented several demo applications which are designed to test utmost features of the implemented deployment runtime.

To run the system, we need a target environment. For the demo purposes, we have prepared a simple target system containing two nodes and an interconnect between them.

[7] In this case, also the connector implementation is expected to be replaced by the generated connector.

8.7.1 Local Demo

Local demo is a simple application verifying basic deployment ability of the system. The application contains a simple logging system built on Fractal. The demo architecture is presented in figure 13. It has several entities of four types – Message Producer, Message Arbitrator, Log Factory and Log. The basic architecture can be described as follows: the Message Arbitrator uses Log Factories to create Logs and passes these Logs to the Message Producer which produces messages to these Logs.

The Message Producer cyclically produces log messages containing a sequence number and writes them to a log which has been set through the producer's only business interface. The log is set by the arbitrator and is time to time changed.

The Message Arbitrator is connected to several Log Factories. Each of these factories has an ability to create any number of Log objects. The Log object accepts a message and prints it to a standard output, together with an identification of the Log Factory which have created the Log and identification of the Log itself. The Message Arbitrator periodically requests randomly chosen factory to create a new log and sets this log to the producer.

An output of the application therefore consists of a sequence of lines containing messages with incremental numbers, time to time changing the identification of the Log (and the factory as well).

The Deployment Plan is flat, as expected by the specification. The whole application is deployed to one node, no connectors are present.

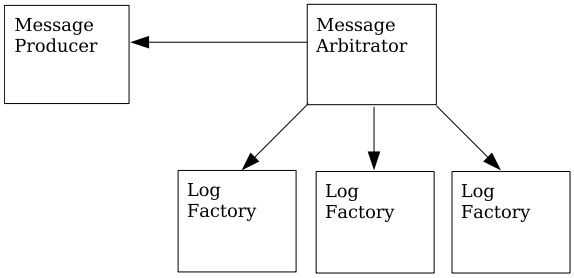


Figure 13: Architecture of the Local Demo application

This demo tests particular sanity of the system. To run the demo, the deployment runtime has to be able to create all needed objects and to distribute data as required. It has to be able to load the Fractal plugin and use it to instantiate, interconnect and start the application components.

Also basic features of the Fractal plugin are verified. It has to be able to instantiate primitive components and set their properties (this is tested by Log Factories – an identifier is set to each of them). To connect components, it has to be able to communicate to components properly and to translate port natural references to a stringified form and back. It has to be able to start and stop each component.

The principle of representation of a port collection by particular ports with properties is verified, because the arbitrator is connected to the log factories through a port collection.

The application requests a separate execution environment, thus, also this feature of the runtime is tested. The Message Arbitrator has a requirement for an amount of memory of the target Node which tests resource allocation.

8.7.2 Distributed Demo

Distributed demo is similar to the local demo, it has an identical code. However, it is distributed into two nodes, using connectors. The demo architecture is presented in figure 14 (the Deployment Boundary separates particular nodes). The Message Producer is deployed to one node, the Message Arbitrator to the other. Log Factories are distributed to both nodes. Connectors are written for each of the three connection types of the application: Log Configuration used by the arbitrator to set log targets to the producer, Log Factory used by the arbitrator to request log creation from factories and Log connector instantiated when passing Log target remotely through one of the first two connectors.

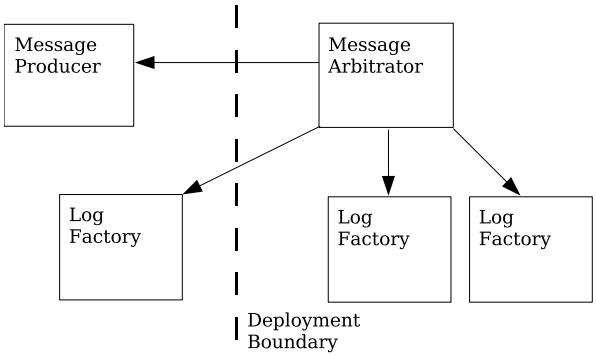


Figure 14: Architecture of the Distributed Demo application

This demo tests (over all the features tested by the Local Demo) a distribution sanity of the system. It proves the concept of passing connectors in the Deployment Plan as any other components, indistinguishably for the deployment runtime. It proves that the hand-written connectors can provide distributed connections transparently for application components because the code of the Local Demo and the Distributed Demo differs only in presence of connectors.

The ability of the Execution Manager to split a Deployment Plan to parts for particular nodes is tested.

The Fractal plugin's ability to provide a bridge to connector interfaces is verified because all the connectors implement only the connector interfaces, not using either runtime or Fractal interfaces or data formats.

Basic functionality of the hand-written connectors is tested. The ability to instantiate a new connector is tested when passing a log target from the arbitrator to the producer, a connector for the log has to be created (each of these entities is deployed to different node). An optimization of multiple remote references is also tested. In the situation that a log factory living on the same node as the producer is asked for a new log, the log is passed remotely from the factory to the arbitrator (which gets a new connector to the log), then remotely back to the

producer. Without this feature, the producer would get a connector to another node referencing a connector back to the log living on the same node as the producer. But it gets a direct local reference to the log, without superfluous connectors.

This application also tests connection reconfiguration functionality. In the plan, connectors are used for each connection. If a log factory is deployed to the same node as the arbitrator, the connector between them is not needed and the connection is reconfigured to be direct. We also provide a mutation of the Deployment Plan, which is similar to the first one but deploys everything to one node. In this case, all connections are reconfigured and no connector is used. The demo then behaves similarly to the local demo.

8.7.3 Hierarchical Demo

The hierarchical demo extends the distributed demo and is dedicated to test containment functionality of the Fractal plugin. The demo architecture is presented in figure 15. In the picture, solid lines represent physical placement of components while dashed lines and background colors represent logical component hierarchy. The structure of the application contains all types of components. The primitive components from the previous demos remain and two more components are added. The first one is called Logger. It is a composite component, containing the arbitrator and all factories as subcomponents. Its only port therefore is the Log Configuration port, delegating a communication between the external producer and the internal arbitrator. The second added component is a top-level component called Demo. It contains the Logger and Message Producer as subcomponents. The Demo is a hybrid component without ports. After it starts, it traverses the whole hierarchy of its subcomponents and prints a tree containing their names (which implies usage of a Name Controller of each component and also passing subcomponents of subcomponents, thus, instantiating new containment connectors).

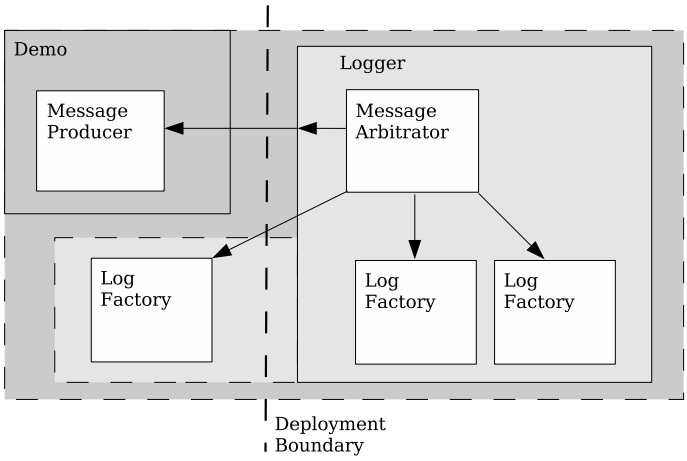


Figure 15: Architecture of the Hierarchical Demo application

This demo proves the concept of passing containment information in properties not understood by the deployment runtime. It proves the concept of distributed containment achieved by connectors.

The demo verifies an ability of the system to employ all three types of components and to handle with the principle of external and internal ports of composite components.

The Fractal plugin is tested to support both solutions of containment. The containment of the Producer and Logger in the Demo component is represented by explicit connections in a Deployment Plan with explicit containment connectors, containment of the Arbitrator and Factories in the Logger component is represented as properties of instances. The Fractal plugin is tested to create implicit containment connectors only where they are needed (the locally deployed components are connected directly) and to handle the explicit units properly by a wrapper. Containment connectors are tested to provide the remote containment feature transparently for application components and for the code generated by Julia. The connection reconfiguration feature is tested for the deployment-plan-explicit connector units – connections of locally deployed units are reconfigured and the containment is connected directly. This demo also tests an ability of the hand-written containment connectors to instantiate new connectors and to handle a multiple redirection – when the Demo component asks the Logger subcomponent (which lives on the other node) for its subcomponents, it gets a direct connector for the Factories living on the other node and direct local reference to those living on the same node.

8.7.4 Executor

To initiate execution of the demo applications in the deployment runtime, an executor is required. We have implemented a simple demo executor which executes one of the three demo applications (chosen by an argument). It performs all the steps defined by the OMG D&C execution processes described in the section 2.4. It lets the application run for a while and then it stops the application and finishes.

In addition, the executor has a "multiple applications" mode which tests an option of launching multiple applications from one application manager. In this mode, the executor executes two applications in parallel and after their finish, executes one more application, still from the same application manager. Each application requires almost all target system resources, therefore, one of the first two applications is not started and throws a "resource not available" error. This tests allocation of resources for these applications.

9 Evaluation of the OMG D&C

We have worked with the version 05-01-07 of the OMG D&C. The specification is well-structured and its usage is mostly clearly explained. Generally, we have found the specification highly usable.

The model is quite large and complicated and many entities are described sequentially. Summarizing diagrams would be appreciated to increase the readability.

The evaluation can be divided into two groups – conceptual and technical.

9.1 Conceptual Evaluation

Conceptually, the specification covers most (if not all) requirements of distributed deployment. However, we have encountered several problems.

9.1.1 Component Interface Description

The most important problem is the lack of description of components. The Deployment Plan contains one general entity `realizes`, describing an external component interface of the application. But we need also a description of the interface of each component, without this information we are not able to instantiate it. This problem has been solved in our Platform Specific Model (see section 7.1).

9.1.2 Stopping of Running Applications

In the Execution Management Model a running application is finished in one step. During this step, the application is stopped and the runtime infrastructure created for the application is removed. We consider this principle insufficient. An application may run on multiple nodes, therefore, its finish is performed asynchronously on its components. During this process, some components of the application are stopped and the others run, which is a situation, that can be handled by the components. But if the infrastructure is removed synchronously with the stopping, the stopped components are immediately destroyed. Then, some components are still running, and the others are already not present in the memory, which can cause the running components to crash. Therefore, we have separated the process into two steps by adding a `stop` methods throughout the deployment runtime entities, similar to the `start` methods present in the OMG D&C model. The application finish runs in two steps – in the first step the application components are stopped by the added methods and in the second step the infrastructure is released using the standard calls defined in the specification.

9.1.3 Splitting the Deployment Plan

The Execution Manager splits the Deployment Plan into separate plans, one for each node. These plans should contain only entries needed for the application parts deployed on the particular nodes. There are several issues around that.

First, it is not clear how the global `realizes` section should be distributed among the subplans. When it is copied as is, the subplans contain a `realizes` section with a content that they do not actually realize. But it would be very difficult or maybe impossible to build an interface description conforming to the actual interface realized by the components that happen to be deployed to the same node. In our implementation, the section is copied to all subplans but the Node Managers do not rely on this behavior.

Second problem is that each Node Manager gets its own deployment plan, built according to rules, which are not defined well. This produces a discontinuity of the application – the application parts on the nodes have no way to identify the global

application they belong to. But sometimes (e.g. when connecting implicitly started containment connectors, see section 7.3.2) a global identification of the application among the nodes is required. We have made the Domain Application to send a global identifier to the Node Applications as a `configProperty` during the start launch step but this solution breaks a little the contract on the vendor boundary between Execution Manager and Node Manager. However, requirements made on the split deployment plans would break it even more. In the case of passing identifier as a config property, if the property is not found, a warning is printed and some extended features (as implicit distributed containment) do not work. Making presumptions on uniqueness of some records in the plan may produce unexpected results which is the greater of two evils.

9.1.4 Component Instantiation Ordering

An ordering of instantiation or interconnection of components is not possible. This feature is typically not needed because components can be typically instantiated and interconnected in any order. However, in some cases, an ordering is required. For example, when executing a hierarchical application, some component models can require subcomponents to be instantiated before their parent component (an API for instanitating a component accepts instances of subcomponents). This requirement is incompatible with OMG D&C. In each step of the three-step-execution approach, each Node Manager operates separately with its part of the Deployment Plan and there is no way to synchronize actions among the nodes. If an ordering is needed, the concept has to be modified. These actions have to be initiated component by component from the Execution Manager.

An ordering of component instantiation is not supported by our implementation. An ordering of connections can be partially simulated by the connection reconfiguration feature – references can be corrected when all prerequisities are ready.

9.1.5 Resource Commitment

The resource commitment solution in the specification has been improved from its previous version but is not excellent yet.

Before an application is executed, its requirements have to be matched against the target environment resources and the resources have to be committed. The commitment can be done during the Planning phase by the planner. The Execution Manager during the Prepare phase either adopts this commitment, or commits the resources by itself, if the planner did not do it. Therefore, a Domain Application Manager is always started with the application resources committed. But there is an option to launch multiple Application instances from one Application Manager which serves as an application factory. But when the application is started multiple times, the resources have to be committed again for each instance. In this case, the resources are committed during the Start Launch step by the Domain Application Manager. This works, but may be confusing from the user's view, because the commitment is performed each time in different phase and may resulting into unexpected errors.

In the Deployment Plan, there is a `ResourceDeploymentDescription` which maps application requirements to target system resources. This entity has a set of properties, which contain "the aspects of the resource actually allocated, if any" [16]. It is not clear what are these properties for because the Execution Manager either gets no Commitment Manager and then commits resources by itself, or gets a Commitment Manager and "then it is assumed that resources were already committed by an online planner" [16]. Thus, the information stored in the properties seems to be superfluous.

One more rather technical problem is related to the resource commitment issue. The `ResourceCommitmentManager`, which provides the commitment feature, provides only methods for commitment and release of a list of resources. Therefore, an ability to build a list of resources required by an application has to be implemented several times or the list has to be passed among the entities somehow. As the resources are committed and released always for one application instance, the set of required resources could be stored in the Commitment Manager.

9.2 Technical Evaluation

The overall technical usability of the schema is good. Again, a few problems have been identified during the work.

The specification defines that the `joinDomain` method on Node Managers is called by the Execution Manager "at startup time or when it is informed of a new node via the `updateDomain` operation" [16]. However, there is no `updateDomain` operation in the Execution Manager, it is a method of the Target Manager. This makes a little confusion in the decision who calls the method actually. As the Execution Manager is mentioned multiple times and is used also in the Platform Specific Model for CORBA Component Model, our implementation uses this interpretation.

The `updateDomain` method of the Target Manager has no error reporting, no exceptions are defined to be thrown. This can be a trouble because the coming domain information can easily contain nonsense.

Several problems were encountered with the XSD schema provided for CORBA Component Model. These problems are elaborated in section 8.2.3.

10 Evaluation and Related Work

10.1 Evaluation of the Work

10.1.1 Specific Goals Achieved

All specific goals set forth in section 5 are achieved by the work:

- **Deployment Repository**: The data model carrying the metadata needed for deployment has been defined. We have considered several tools for creating the

repository, thus, to load the metadata to memory. We have chosen the JAXB backend of Java Web Services. We have created a XSD schema according to our Platform Specific Model and generated a repository from this schema.

- **Heterogeneous Deployment Runtime**: The Execution Management Model and Target Management Model have been implemented. The implementation adhere to the metamodel exactly. Moreover, it provides a connection reconfiguration extension.
- **Support for Different Component Models**: The support for different component models has been designed, it uses Execution Environments and Component Model Plugins to instantiate components and connectors to provide communication among them. This support is hidden behind the interface of the Node Manager. Support for two component models have been analyzed, support for one of them fully implemented.

10.1.2 Top-level Goals Achieved

All top-level goals formulated in section 1.5 are achieved by the work:

- **Support for Execution of Heterogeneous Applications**: The support for execution component based applications have been implemented according to the OMG D&C. The support for different component models is provided by Component Model Plugins. They are added to the system as extensions, possibly developed by a third party.

  Since we have implemented only one component model plugin, no really heterogeneous application can be run yet. But components in our system are deployed separately and use connectors to communicate to the other components. After support for another component model is added, heterogeneous applications can be executed because connector units will comunicate to each other through their natural interfaces and communication between the units and application components is a responsibility of plugins.

  We have proven the concept of passing component-model-specific data in properties through a Deployment Runtime which does not understand them.

- **Adherence to the OMG D&C**: The system adheres to the specification very closely. We have defined the Platform Specific Model for Heterogeneous Deployment which extends the Platform Independent Model by specifying minimalistically the unspecified parts. A few features necessary for deployment of components built on different component models are added to the Data Model but these are only small changes to allow storing extended data. Several standard properties have been defined to control features of the Deployment Runtime. The Management Models were used unchanged.
- **Evaluation of the OMG D&C**: We have created an implementation very closely adhering to the OMG D&C Execution Model, therefore we have proven its usability in practice. All problems with the specification met during the work are listed in section 9.

10.1.3 Summary

The system is prepared to execute applications built on many component models. Support for these models is added to the system in a form of plugins, that can be developed by a third party, for example by vendors of the particular component models. When a component model plugin is added, applications built on the supported component model can be executed, as well as all heterogeneous applications whose each component is supported by one of the installed plugins. The system closely adheres to the OMG D&C which increases its maintainance.

10.2 Related Work

The closest related work is the thesis [9]. It proposes support for deployment of heterogeneous component-based applications based on the OMG D&C. It addresses all deployment phases from installation to execution.

The major part of the work analyzes many component models and proposes a Unified Deployment Component Model (UDCM). This model is based on the OMG D&C Component Data Model and all features of the analyzed component models are added. Support for particular component models is also provided by plugins. At the beginning of the deployment process, a plugin transforms application metadata from the component-model proprietary format to the UDCM. The deployment process runs uniformly over the UDMC and finally a plugin is used to execute the application.

There are several basic differences between the works. First of all, the work [9] uses the OMG D&C specification as a starting point but diverts from it a lot. The data models are extended to contain all features of the component models, while our work adheres to the data models closely and lets plugins hide the extended features to properties. This even applies to a component hierarchy, in contrast to the work [9], we have adopted the flat Deployment Plan.

The work does not adhere to the OMG D&C Execution Management Model structure. There is no Execution Manager, no Node Managers and Application Managers. All their functionality is encapsulated to a domain-central Deployment Manager. Our work adheres to this model exactly.

There is a different approach to component model plugins. In the work [9], plugins are used directly in the Deployment Manager. They accept a whole Deployment Plan and execute the whole application, for which an infrastructure of the particular component model is used. In our work, a common infrastructure is implemented according to the OMG D&C specification and plugins only manage particular components.

The work [9] is only able to deploy heterogeneous applications composed of two component models. For each component model combination, a special plugin has to be provided. Such plugin splits the Deployment Plan into two homogeneous parts, passes them to single component model plugins and maintains communication between the application parts. In our work each component is maintained by an appropriate plugin and communication among them is unified, therefore there are no differences between homogeneous and heterogeneous applications and an appli-

cation can consist of many component models. This quality is achieved thanks to a connector generator and runtime-connector bridges implemented in plugins.

The work [9] is closely tied to the MOF standard [17], from requirements to implementation. We adhere to this standard while specifying model within the frame of OMG D&C but in the implementation, we have considered several metadata representations, the MOF-based as one of them, and have chosen different approach.

Component versioning is deeply elaborated in the work [9]. This problem is not addressed by our work, it is left on vendors of plugins to isolate the versioning issues from the runtime, as any other component-model-specific features. However, our system does not support applications using different versions of the same class (see section 8.5.3). Before such applications can be executed, an approach to loading classes is to be adopted from this work.

The Deployment And Configuration Engine (DAnCE) [6] implements the OMG D&C. However, only the CORBA component model is supported, with its Component Integrated ACE ORB implementation. The implementation of DAnCE does not provide all specified features yet. The generalization for other CCM implementations and other component models is a future work of this project.

The paper [12] proposes a Deployment Environment. Authors identify several existing tools, which solve parts of the deployment process (e.g. an InstallShield Developer) and merge them by wrappers into one Deployment Environment. The paper defines a component model based on CORBA component model. However, the model is too simplified to suffice for the whole deployment process of component-based applications, as proposed by the OMG D&C.

Another deployment environment is presented in paper [19]. Authors use their own component model, which is similar to Fractal component model, and should be suitable for many other component models. An application is described using an ADL language. The work focuses on scalability and fault tolerance. The scalability is achieved by a hierarchical structure of deployment runtime, which follows structure of the deployed application, and by dividing the deployment process into asynchronous tasks. The fault tolerance is achieved by using autonomous agents for each small part of the system and by system of diagnostic and repairing facilities. This work does not address the deployment of heterogeneous applications.

11 Conclusion and Future Work

11.1 Conclusion

In this thesis we have presented the design and implementation of Deployment Runtime which executes heterogeneous component-based applications. It is based on the OMG D&C specification. The system adheres to the specification closely and does not divert from it unless necessary.

A common deployment runtime for all supported component models is proposed, using only component model APIs to manage the lifecycle of particular com-

ponents. Besides the obvious advantages of having one deployment system for multiple component models it enables deployment of heterogeneous applications. Connectors are used to enable communication among components of such applications, without giving up compliance with the OMG D&C specification, which broadens the applicability of the OMG specification even though application heterogeneity and the use of connectors are not specifically supported by the specification.

An extension that deals with connection reconfiguration was introduced into the OMG specification. This feature enables connection ordering at application startup and modifying application structure during application run. This extension was added without breaking contracts defined at vendor boundaries – entities that the deployment runtime consists of can be incorporated to the system regardless of having this extension or not.

The component types present in the OMG D&C are not sufficient for all component models. To enable support for them, the non-virtual assemblies were considered over the scope of the OMG D&C. Support for these compnents was proposed without breaking the compliance with the OMG D&C – they are projected to component types present in the sepcification.

The deployment runtime has been designed to allow adding support for different component models through extensions to the component connection and lifecycle management subsystem. Requirements for supporting the SOFA and Fractal component models within the runtime were analyzed, and support for the Fractal component model has been also implemented.

The work constitutes one of the few number of deployment systems almost completely compliant with the execution part of the OMG specification. It summarizes experience gained when working with the specification, presents several limitations of the specification and proposes a way to eliminate them.

11.2 Future Work

This thesis proposes a support for the execution phases of the deployment process. To provide a complete support for deployment of heterogeneous applications, it requires integration of several projects. The future work is to integrate the system with these projects after they are ready to be used. Most important is the Deployment Planner which generates all inputs for the system. The connectors are to be generated by the planner, however, minor changes are required in the implementation of the deployment runtime to support them. The support for SOFA is to be finished and support for another models added. A more complex executor, which allows to execute any application and to stop it upon request, has to be implemented. To support a dynamic domain, the Execution Manager has to be extended to handle the changes.

References

[1] Bálek, D., Plášil, F.: Software Connectors and Their Role in Component Deployment, In proceedings of DAIS01, Krakow, September 2001

[2] Brunneton, E., Coupaye, T., Stefani, J.B.: The Fractal Component model, http://fractal.objectweb.org/specification/index.html

[3] Bulej, L., Bureš, T.: Eliminating Execution Overhead of Disabled Optional Features in Connectors, Charles University, Prague, Sep 2006

[4] Bureš, T., Plášil, F.: Scalable Element-Based Connectors, Charles University, Prague, Jun 2003

[5] Connector Generator, http://dsrg.mff.cuni.cz/ bures/congen/

[6] Deployment And Configuration Engine (DAnCE), http://www.dre.vanderbilt.edu/˜ schmidt/DOC_ROOT/CIAO/docs/releasenotes/dance.html

[7] Eclipse Modeling Framework, http://www.eclipse.org/emf/

[8] Enterprise Java Beans specification, version 2.1, Sun Microsystems, November 2003

[9] Hnětynka P.: Making Deployment Process of Distributed Component–based Software Unified, Doctoral Thesis, Charles University, Prague, September 2005

[10] Java Architecture for XML Binding (JAXB), http://java.sun.com/webservices/jaxb/index.jsp

[11] Julia, http://fractal.objectweb.org/julia/

[12] Lestideau, V., Belkhatir, N., Cunin, P.-Y.: Towards automated software component configuration and deployment, In proceedings of PDTSD'02, Orlando, Florida, July 2002

[13] Log4j, http://logging.apache.org/log4j/docs/index.html

[14] ModFact, http://modfact.lib6.fr

[15] Object Management Group: CORBA Components, v 3.0, OMG document formal/02-06-65, June 2002

[16] Object Management Group: Deployment and Configuration of Component-based Distributed Applications Specification, OMG document ptc/05-01-07, January 2005

[17] Object Management Group: Meta Object Facility specification, version 1.4, OMG document formal/2002-04-03, April 2002

[18] Object Management Group: Model Driven Architecture (MDA), OMG document ormsc/2001-07-01, July 2001

[19] Quema, V., Balter, R., Bellisard, L: Asynchronous, Hierarchical, and Scalable Deployment of Component-Based Applications, In proceedings of CD 2004, Edinburgh, UK, 2004

[20] SOFA, http://dsrg.mff.cuni.cz/sofa20

A Attached Compact Disc

A compact disc containing the prototype implementation and related resources is attached to the thesis.

A.1 Content of the Disc

The attached disc contains everything needed to run the Deployment Runtime on Linux and Windows platforms. Both binary distribution and source code are present and are independent of each other. Moreover, text of the thesis and generated source code documentation are provided.

The only direct prerequisite to run the system is the *Java Runtime Environment* ver. 1.5 or higher. To run the system from the source code, the *Java Development Kit* and *Ant* are necessary. These systems can be installed from the disc, if necessary. All other pieces of software needed during runtime are present as libraries and are loaded automatically.

Directories on the disc contain `README.txt` files with detailed description of their content. The disc contains following basic directory structure:

- `binary`: Contains a binary distribution of the Deployment Runtime, including execution scripts, demo target environment configuration, demo applications and installed Fractal plugin.
  - `bin`: Contains binary code of the Deployment Runtime and execution scripts for both platforms.
  - `conf`: Contains sample configuration files.
  - `demos`: Contains everything needed for demonstrational purposes – configuration of the system for the demo target environment, configuration and distribution of the demo applications and scripts executing the deployment runtime and the demo applications.
  - `lib`: Contains all external resources required by the system.
  - `plugins`: Contains the installed Fractal plugin.
- `prerequisites`: Contains all prerequisites required to be installed in the operating system to enable work with the other content of the disc.
- `resources`: Contains text of this thesis, source code documentation generated by javadoc and complete XSD description of the Platform Specific Model for Heterogeneous Deployment.
- `source`: Contains complete source code, scripts that facilitate its compilation and execution, libraries needed to run the system and the demo applications.

A.2 Executing the System from the Binary Distribution

To execute the Deployment Runtime and the demo applications from the binary distribution, an installed Java Runtime Environment ver. 1.5 or higher is required. The installers for both platforms can be found in directory `prerequisites`.

The demo target environment and the demo applications are prepared in directory `binary/demos`. Before the software is executed, its configuration can be modified in directory `binary/demos/conf`. Description of the configuration values is present directly in the particular configuration files. By default, the system is configured to run on a single host and is set in a `debug` verbose mode.

The Node Manager needs to store a temporary data to a local drive. By default, it creates a directory `runtime-data` in the distribution's top-level directory. If the distribution is executed from the compact disc, a directory where the data can be temporarily written has to be passed to the Node Manager through an environment variable `DEPLOYMENT_TMP_DIR`.

The execution scripts are stored in directories named by the used platform (`binary/demos/linux` and `binary/demos/windows`). First, the Target Manager, Execution Manager and both Node Managers have to be started. This can be done in any order and it results in the running Deployment Runtime. Then, any of the demo executors can be started.

The demo executor finishes after its work is done, the managers can be shut down by typing `ctrl+c`. The Target Manager should be finished last, because it holds the RMI registry used by the other managers.

A.3 Executing the System from the Source Code

To run the Deployment Runtime and the demo applications from the source code, an installed Java Development Kit ver. 1.5 or higher and Ant are required. The installers for both platforms can be found in directory `prerequisites`.

Everything else required to run the system from the source code is present in directory `source`. The following actions lead to the running system:

- **System compilation**: First, the binary files have to be prepared. This is done by running command `ant compile` subsequently in directories `source/mdr`, `source/api` and `source/runtime`.
- **Prepare distributions of subprojects**: To prepare distributions of the demo applications and of the component model plugins, run command `ant buildSubprojects` from the `source/runtime` directory. This also installs the plugin.
- **Configure the Deployment Runtime**: The configuration files stored in directory `source/conf` can be modified. However, if the whole system is executed on a single host, the default configuration can be sufficient.
- **Execute the Deployment Runtime**: Particular parts of the Deployment Runtime (configured for the demo target environment) are executed by

`ant` targets `runTargetManger`, `runExecutionManger`, `runNodeManager1` and `runNodeManger2` in the `source/runtime` directory. These targets can be run in any order and result to a running Deployment Runtime.

- **Execute a demo application**: The demo applications can be executed by other ant targets from `source/runtime` directory. The targets are `runLocalDemo`, `runDistDemo`, `runDistDemoLocally` and `runHierDemo`.

The demo executor finishes after its work is done, the managers can be shut down by typing `ctrl+c`. The Target Manager should be finished last, because it holds the RMI registry used by the other managers.

Products of the compilation can be removed by running `ant clean` in the three directories, products of the subprojects distribution building process can be removed by running `ant cleanSubprojects` in the `source/runtime` directory.