Using Connectors for Deployment of Heterogeneous Applications in the Context of OMG D&C Specification

Lubomír Bulej [1, 2], Tomáš Bureš [1,2]

[1] Charles University, Faculty of Mathematics and Physics,
Department of Software Engineering
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
{bulej,bures}@nenya.ms.mff.cuni.cz
http://nenya.ms.mff.cuni.cz

[2] Academy of Sciences of the Czech Republic
Institute of Computer Science
{bulej,bures}@cs.cas.cz
http://www.cs.cas.cz

1. Introduction

Component-based software engineering is a paradigm advancing a view of constructing software from reusable building blocks, components. A component is typically a black box with a well defined interface, performing a known function. The concept builds on the techniques well known from modular programming, which encourage the developers to split a large and complex system into smaller and better manageable functional blocks and attempt to minimize dependencies between those blocks.

Several aspects of component-based programming have been embraced by the software development industry and as a result, there are now several component models, such as Enterprise Java Beans [12] by Sun Microsystems, CORBA Component Model [8] by OMG, and .Net [5] by Microsoft, which are extensively used for production of complex software systems.

There are also a large number of other component models, designed and used mainly by the academic community. While most of the academic component models lack the maturity of their industrial counterparts, they aim higher with respect to fulfilling the vision of the component-based software engineering paradigm. This is mainly reflected in support for advanced modeling features, such as component nesting, or connector support. While we are aware of a number of component models used in academia, we are most familiar with SOFA [11,7] and Fractal [6]. Throughout the paper, we will use these models along with EJB as a test-bed for our experiments.

Typically, component applications are modeled as distributed and platform independent, with a particular execution platform selected during development.

However, the current trends in software industry concerning enterprise integration may hint that platform independence on the level of design is not enough.

Our aim is to make possible creating heterogeneous applications which, in addition to the above, can consist of components written using different component models. This brings us two problems that need to be solved – 1) making the different components work together, and 2) deploying the resulting heterogeneous application.

The two problems may seem orthogonal, but in fact they are connected due to the nature of the differences between component models. These differences comprise mainly component packaging format and deployment, component instantiation and lifecycle management, communication middleware, hierarchical composition of components, etc. To make the different components work together and create a truly heterogeneous component application, we need to overcome those differences.

A key problem in making components from different component models work together is communication. Connections in different component models have different semantics and typically use different communication middleware to achieve distribution. Contemporary solutions to this problem usually employ middleware bridges (e.g. BEA WebLogic, Borland Janeva, IONA Orbix and Artix, Intrinsyc J-Integra, ObjectWeb DotNetJ, etc.) to connect components form different component technologies, which only tackles the issue of different middleware and leaves out the issue of different semantics and other (connection related) differences between the component models.

We propose to use software connectors [1] to define the semantics for connections between components from different component models. Based on requirements placed on a specific connection, the implementation of a connector can be automatically generated or, if the semantics allows it, a suitable middleware bridge can be used to mediate the connection.

Deployment of component applications is one of the most burning problems for the majority of component models. The deployment process generally consists of several steps, which have to be performed in order to successfully launch a component application. Without deployment support and tools, a component model is unusable for serious software development.

Most of the component models address the deployment issue in some way, but the differences between various component models have made it difficult to arrive at a common solution. Therefore the deployment process for component applications is specific to a particular component technology and a vendor. Worse, even applications written for a standardized component technology (e.g. EJB) have to be deployed in a vendor specific way using the vendor's proprietary tools.

The above mentioned situation makes the integration of components from different component models and the deployment and maintenance of the resulting application practically impossible.

A promising approach to deployment of heterogeneous component applications is modeling the application in a platform independent manner and mapping the platform independent model into the target environment to ensure interoperability. The first step in this direction has been done by the Object Management Group (the body behind CORBA and the CORBA Component Model [8]), who has published

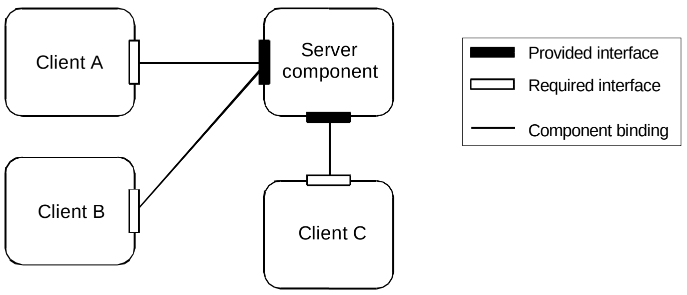


Fig. 1. A model of a simple component application. There is a server component providing two different services with two clients connected to one service and another client connected to the other service.

a specification concerning deployment and configuration of distributed component-based applications [9]. Following the MDA [10] approach, the specification presents platform independent models of the application, target environment, and the deployment process, which are then expected to be transformed to platform specific models suitable for specific component technologies and mapped to particular programming environment.

Upon careful examination, though, the OMG specification stops short of providing support for deployment of heterogeneous component applications. The failure rests with the fact that the OMG expects the platform independent model to be mapped into a single target environment at a time. In effect, this means that the specification can be used to define a number of deployment mechanisms, but each of the deployment mechanisms will only support a single target environment. Interoperability between heterogeneous target environments is only provided at a conceptual level, which is rather insufficient.

We believe that the specification should support deployment of heterogeneous applications, rather than conceptually compatible but functionally incompatible deployment mechanisms for heterogeneous target environments. We extend the OMG specification to support deployment of heterogeneous component applications by introducing connectors as bridges between the heterogeneous parts of an application, and by extending the model to support construction of connectors during deployment.

Throughout the paper, we will use a model of a simple component application depicted in Figure 1 as a running example. The rest of the paper is organized as follows. Section 2 presents an overview of software connectors and their use in overcoming the differences between component models. Section 3 provides a short overview of the relevant parts of the OMG specification, and Section 4 demonstrates the deployment of a heterogeneous application using connectors and the OMG model of deployment process. We discuss related work in Section 5 and conclude the paper in Section 6.

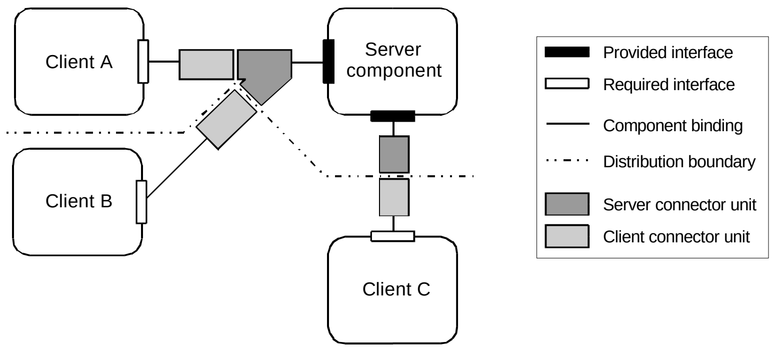


Fig. 2. A model of a simple component application using connectors to capture inter-component interactions. Each connector is split into two parts respecting the distribution boundary. In the case of the procedure call-based connectors (i.e. those used in the example) there is typically one server unit and zero to many client units.

2. Software Connectors

Software connectors are first class entities capturing communication among components. Although connectors may be very complex, modeled by complicated architectures reflecting different communication styles [4], it is sufficient for the purpose of this paper to view a connector as a number of connector units attached to their respective components (see Figure 2).

Apart from modeling and representing inter-component communication, connectors have another important feature – they can be generated automatically based on a high-level description expressed in terms of a communication style and non-functional properties [4]. That allows a developer to concern herself just with components' business logic and not with glue code (often containing middleware dependent parts) used to provide communication among components in distributed environment.

The whole trick of using connectors is to plug an appropriate connector instance between every two components. However, as the concept of connectors is not typically present in current component models, it is often necessary to extend them to support connectors. That is easily done by hooking in the process of component instantiation and binding. Upon instantiation of a component, we create the server connector units and make sure that whenever component interfaces are queried, a connector reference to a corresponding server connector unit is returned (instead of returning a direct reference to a component interface). Similarly, whenever an interface is being connected to another component, a client connector unit is created and bound using the connector reference. In our approach, the connector reference is a container holding a set of server unit identities depending on available transport methods (e.g. Java reference for in-address-space connections, RMI reference for RMI connections, or IOR for CORBA connections).

When instantiating a connector unit at runtime, the information as to what implementation is to be used for that particular connector unit is looked up in a structure called *connector configuration map*, which contains pairs <interface discriminator, connector unit implementation>. The interface discriminator uniquely identifies either a server interface, in which case the discriminator is a pair <component, interface name>, or a client interface, in which case the discriminator is a tuple <client component, client interface name, server dock name, server component, server interface name>. Such a description reflects the fact that a connector unit attached to a server interface is created in advance and can exist on its own, while a connector unit attached to a client interface is created during component binding when the binding target is already known.

3. Overview of the OMG D&C Specification

As mentioned earlier, the OMG Deployment and Configuration Specification is the first step towards unified deployment of component applications. The specification provides three platform independent models, the component model, the target model, and the execution model. These models represent the three major abstractions used during the deployment process, which uses these models to deploy an application.

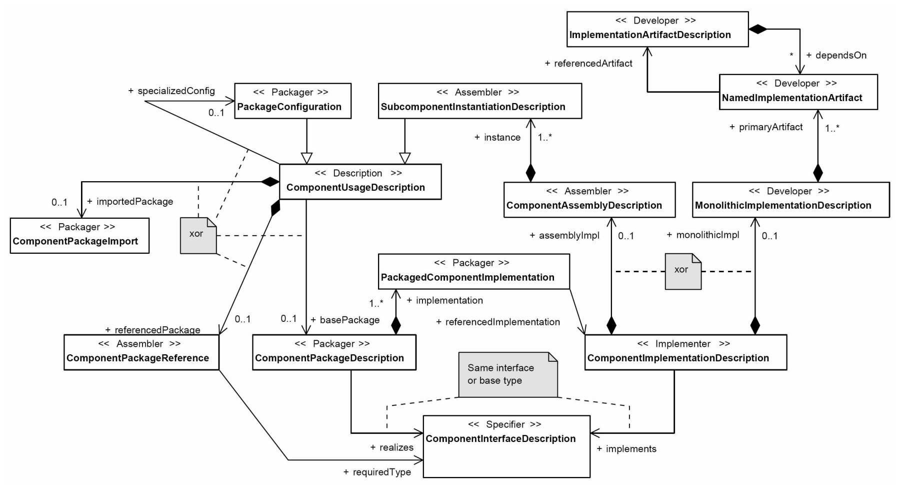
For use with specific component models, the platform independent models should be transformed to platform dependent models, capturing the specifics of the concrete platform. A more detailed overview can be found in [3], and yet more details can be found in the specification itself [9].

To reduce the complexity, the models are split into the data model and the management (runtime) model, with the management models describing runtime entities dealing with the data models. The management models are not important in the scope of this paper; therefore we will only deal with the data models.

3.1 Component Data Model

The component data model captures the logical view of a component application. A high level overview of the component data model is depicted in Figure 3. The key concept is a component package, which represents a reusable work product. A component package is a realization of a specific component interface, and contains possibly multiple implementations of the realized interface. As a reusable product, the package contains configuration of the encapsulated implementations, and selection criteria for choosing an implementation by matching the criteria to the capabilities of the individual implementations.

The implementation of a component interface can be either monolithic, or an assembly of other components. A monolithic implementation consists of a set of implementation artifacts that make up the implementation. The artifacts can depend on each other and can be associated with a set of deployment requirements and execution parameters. The deployment requirements have to be satisfied before an artifact can be deployed.



<< Developer >> ImplementationArtifactDescription
+ referencedArtifact
* + dependsOn
<< Developer >> NamedImplementationArtifact
+ primaryArtifact
1..*
+ specializedConfig
0..1
<< Packager >> PackageConfiguration
<< Assembler >> SubcomponentInstantiationDescription
+ instance
1..*
<< Description >> ComponentUsageDescription
0..1
+ importedPackage
<< Packager >> ComponentPackageImport
xor
<< Assembler >> ComponentAssemblyDescription
<< Developer >> MonolithicImplementationDescription
+ assemblyImpl
0..1
+ monolithicImpl
0..1
<< Packager >> PackagedComponentImplementation
xor
1..*
+ implementation
+ referencedImplementation
+ referencedPackage
0..1
0..1
+ basePackage
<< Assembler >> ComponentPackageReference
<< Packager >> ComponentPackageDescription
<< Implementer >> ComponentImplementationDescription
Same interface or base type
<< Specifier >> ComponentInterfaceDescription
+ realizes
+ implements
+ requiredType

Fig. 3. An overview of the component data model

An assembly of components, depicted in Figure 4, contains references to other component packages to serve as subcomponents of the assembly. The instances of subcomponents are connected using connections between endpoints defined by subcomponents and external endpoints of the assembly. To allow for configuration of an assembly, which in itself does not carry any implementation code, the configuration properties of an assembly are delegated to its subcomponents through a defined mapping.

3.2 Target Data Model

The target data model describes the computational environment into which the application is deployed. The environment, termed domain, consists of computational nodes, interconnects and bridges. Since the target model is not important in the scope of the paper, we will not describe the model in greater detail.

3.3 Execution Data Model

The execution data model depicted in Figure 5 describes the physical structure of a component application. The model represents a flattened view of the original component data model describing the logical structure of an application. The execution data model the application in terms of component instances, connections between endpoints of the instances, and assignment of the instances to computational nodes in the target environment

The component instances carry configuration properties which can be used to influence their behavior, and their implementation is in turn described in terms of implementation artifacts. The artifacts, which are binary files containing implementation code, can carry individual execution parameters, which can be

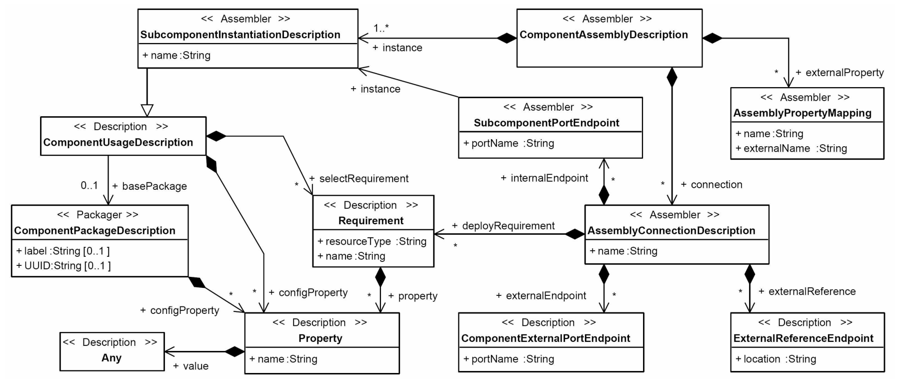


Fig. 4. A detailed view of the component assembly description

used to tell the computational node how to treat a particular implementation artifact. Since the artifacts can be shared by multiple implementations, the execution parameters can be defined per implementation.

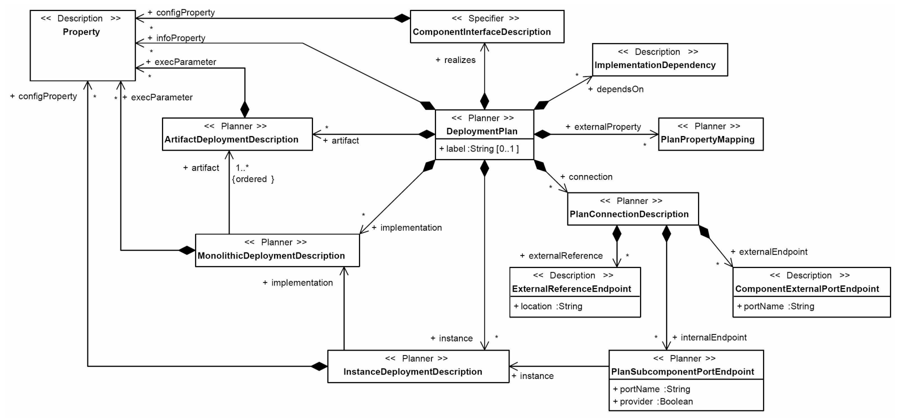
3.4 Deployment Process

Prior to deployment, the component application must be developed, packaged, and published by the provider and obtained by the user. The deployment process defined in the specification then consists of five stages and is performed by a designated actor called deployer.

Installation. During installation, the software package and its component data model is put into a repository, where it will be accessible from other stages of deployment. The location of the repository is not related to the target execution domain. Also, the installation does not involve transfer of binary files to the computational nodes in a domain.

Configuration. When the software is installed in the repository, its functionality can be configured by the deployer. The software can be configured multiple times for different configurations. The configuration stage is meant solely for functional configuration of the software, therefore the configuration should not concern any deployment related decisions or requirements.

Planning. After a software package has been installed into a repository and configured, the deployer can start planning the deployment of the application. The process of planning involves selection of computational nodes the software will run on, the resources it will require for execution, deciding which implementation will be used for component instances, etc. The planning does not have any immediate effects on the environment, but produces a physical description of the application in the execution data model, termed deployment plan.



<< Description >> Property
+ configProperty
+ infoProperty
+ execParameter
<< Specifier >> ComponentInterfaceDescription
<< Description >> ImplementationDependency
+ realizes
+ dependsOn
+ configProperty
+ execParameter
<< Planner >> ArtifactDeploymentDescription
+ artifact
<< Planner >> DeploymentPlan
+ label :String [0..1]
+ externalProperty
<< Planner >> PlanPropertyMapping
+ artifact
1..*
{ordered }
+ connection
<< Planner >> PlanConnectionDescription
+ implementation
<< Planner >> MonolithicDeploymentDescription
+ externalReference
+ externalEndpoint
<< Description >> ExternalReferenceEndpoint
+ location :String
<< Description >> ComponentExternalPortEndpoint
+ portName :String
+ implementation
+ instance
+ internalEndpoint
<< Planner >> InstanceDeploymentDescription
+ instance
<< Planner >> PlanSubcomponentPortEndpoint
+ portName :String
+ provider :Boolean

Fig. 5. An overview of the execution data model

Preparation. Unlike the planning stage, the preparation stage involves performing work in the target environment in order to prepare the environment for execution of the software. The actual transfer of files to computational nodes in the domain can be postponed until the launch of the application.

Launch. After preparation, the application is brought up to the executing state during the launch stage. As planned, instances of components are created and configured on computational nodes in the target environment and the connections among the instances are established. The application then runs until terminated.

4. Integrating Connectors with Deployment

We have presented the two basic concepts we intend to employ to support deployment of heterogeneous component applications. Software connectors, described in Section 2 will be used to overcome the differences between various component models, while OMG D&C Specification, briefly introduced in Section 3 will be used to model the deployment process of a heterogeneous application.

Since the OMG specification does not support the description of heterogeneous component applications and does not directly support connectors, we have to find a way to combine the two approaches. To use connectors with a component application, there are basically two tasks that need to be done, and which need to be integrated with the OMG deployment process:

1. At some point, the implementation of all connectors needs to be generated, which comprises connectors for component bindings a) present in the initial architecture of an application, and b) that can emerge at runtime as a result of passing a reference to component interface
2. The connectors need to be instantiated and bound to their respective components when launching an application. Additionally, the *connector configuration map*

(described at the end of Section 2) must be prepared for each node to allow for later instantiation of connectors that result from reference passing.

4.1. Preparing connectors

To generate a connector, a connector generator needs to have enough information concerning the requirements for the communication the connector is expected to mediate. The specification of connector features has the form of a communication style and non-functional properties. Each connection among instances of components in an assembly can have different requirements.

The planning stage of the deployment process appears to be the most suitable moment for generating connectors. The planning is performed by the deployer using a planner tool. The tool takes as input the component data model, describing the component application, and the target data model, describing the target environment. Using the tool, the deployer assigns instances of components to nodes in the target environment and verifies that an instance can be placed on a particular node. The planner tool has all the information required for generation of connectors, except for the connection requirements.

The specification of connection requirements is not a part of the OMG specification, which therefore needs to be slightly modified. To make the information available, we have extended the AssemblyConnectionDescription in the component data model class with another association named `connectionRequirement`. The association is used to describe the connection requirements.

The connector-aware part of the planner can then communicate with a connector generator [4] and provide the necessary information. For each assembly connection, the generator synthesizes the implementation artifacts and configuration required to instantiate connector and returns the code fragments to the planner. The connector-aware part of the planner then replaces the assembly connections in the component data model with pairs of components encapsulating the connector units connected to the original components.

This step in fact transforms the enhanced component data model (see Figure 6) back into the original plain data model, which can be then transformed to deployment plan according to the original specification, and for which the planner does not need to be modified.

What is important to note, though, that while we transform connectors to components in the context of the OMG specification, connectors are not really components that would be present in the architecture of an application. Connectors are instantiated at runtime, the instance depends on the type of the server a connector unit connects to, and their encapsulation in components as seen in the component data model is merely an implementation convenience.

4.2. Instantiating connectors

The output of the planning stage is a deployment descriptor, describing the physical structure of the application as assigned onto nodes in the target

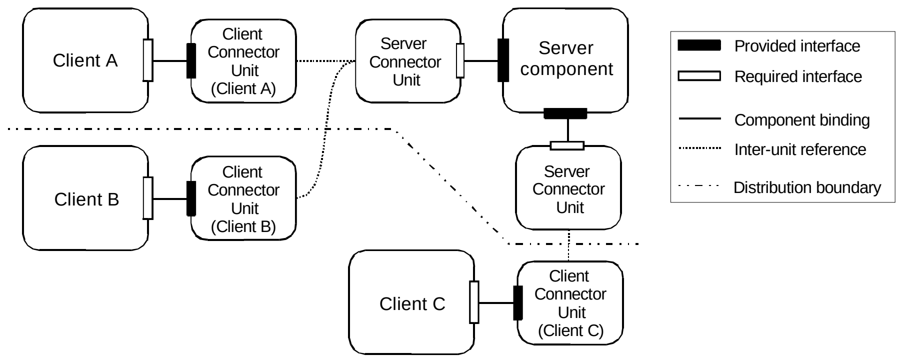


Fig. 6. Application architecture after transformation of the enhanced description

environment. The plan is then broken into pieces with respect to the distribution boundaries and disseminated to individual nodes. The runtime on each of the nodes uses the fragments of the deployment plan to instantiate components and connector units. Depending on the type of the connector unit one of the following actions is taken:

1. Server connector unit is instantiated and bound to its corresponding component. The unit registers its reference (using a name obtained from execution parameters in the deployment plan) in a naming service so as to be accessible by clients.
2. Bound client connector unit (i.e. representing a binding in the initial architecture) is instantiated and its corresponding component is bound to it. The unit retrieves a reference to a previously registered server connector unit from the naming service (using a name obtained from the deployment plan) and establishes the binding.
3. Future client connector unit (i.e. a unit that does not exist in the initial architecture but which can emerge at runtime as the result of reference passing) is stored in the *connector configuration map* for later use.

Since the implementation of a client connector unit depends also on the server component, there can be multiple implementations of a client unit. This is addressed by providing all the implementations in the deployment plan as different artifacts implementing one component and performing the actions 2 or 3 stepwise for the individual artifacts.

5. Evaluation and Related Work

Our approach to deployment of heterogeneous component applications builds upon two major concepts, the concept of software connectors, which is used to define semantics of connections between components from different component models, and the concept proposed by the OMG D&C specification, which unifies deployment of component based applications on conceptual level, but which in itself cannot provide for deployment of heterogeneous applications.

The original component data model present in the specification assumed direct communication between components. That requires that the artifacts providing component endpoints have to be connected together, which makes it impossible to abstract away the middleware technology used for communication. We have made a slight modification to the original OMG specification to enable expression of connection requirements in form of communication styles and nonfunctional properties, which can be used to generate connectors. This allows postponing the selection of communication middleware until the planning stage of the deployment process, or introducing logging, monitoring, or encryption into communication without changing the original application or its description.

We have also described the integration of the connector generator into the deployment process and pointed out places where the planner tool needs to be modified to support the connector generator. By transforming the enhanced component data model to the original component data model, we have avoided excessive modifications to the planner tool. The transformation of component data models is a generic process that can be used to enhance the expressive power of the component data model as long as the advanced constructs can be transformed back to the original data model.

To our knowledge, there is no other work concerning the use of connectors and the OMG specification to support deployment of heterogeneous component applications. There are, however, a number of mature business solutions for interconnecting the leading business component models such as EJB [12], CCM [8], and .NET [5]. A common denominator of these models is the lack of certain features (e.g. component nesting), which makes the problem of their interconnection a matter of middleware bridging. Each of those component models has a native middleware for communication in distributed environment (RMI in case of EJB, CORBA in case of CCM, and .NET remoting in case of .NET).

Even though the bridges represent mature software products, they limit the heterogeneity of the application by prescribing the use of specific communication mechanisms for the components. The connectors, on the other hand, represent a high level view on the connection between components, and allow for the bridges to be employed in the implementation of a connector if necessary.

6. Conclusion

We have presented an approach which we consider a step forward towards deployment of heterogeneous component applications, which allows us to create component applications composed of components implemented in different component models. We have shown how to employ software connectors to overcome the differences between component models and combined the use of connectors with the OMG D&C specification for unified deployment of component applications.

The presented solution is generic and platform independent, and can be used for different component models. We have verified our approach on a prototype implementation, which supports interconnection of components from SOFA, Fractal, and EJB component models, and are currently developing basic tools for deploying and execution of heterogeneous component applications. Mainly due to space constraints, we had to omit some of the details, which can be found in [3].

In the future, we plan to enhance the connector generator and develop more sophisticated tools, mainly the deployment planner and its integration with the connector generator. The work presented in this paper is part of our efforts within the Deployment Framework task of WP2 of the OSMOSE project. The results related to the OMG D&C specification will be submitted to the OMG.

Acknowledgments. The authors would like to thank Petr Tůma for his valuable comments. This work was partially supported by the Grant Agency of the Czech Republic project number 201/05/H014.

References

[1] Balek, D., Plasil, F., Software Connectors and Their Role in Component Deployment, Proceedings of DAIS'01, Krakow, Kluwer, Sep 2001
[2] Bulej, L., Bures, T., A Connector Model Suitable for Automatic Generation of Connectors, Tech. Report No. 2003/1, Dep. of SW Engineering, Charles University, Prague, Jan 2003
[3] Bulej, L., Bures, T., Addressing Heterogeneity in OMG D&C-based Deployment, Tech. Report No. 2004/7, Dep. of SW Engineering, Charles University, Prague, Nov 2004
[4] Bures, T., Plasil, F., Communication Style Driven Connector Configurations, Copyright Springer-Verlag, Berlin, LNCS3026, ISBN: 3-540-21975-7, ISSN: 0302-9743, pp. 102-116, 2004
[5] Microsoft Corporation, .NET, http://www.microsoft.com/net, 2004
[6] ObjectWeb Consortium, Fractal Component Model, http://fractal.objectweb.org, 2004
[7] ObjectWeb Consortium, SOFA Component Model, http://sofa.objectweb.org, 2004
[8] Object Management Group, Corba Components, version 3.0, http://www.omg.org/docs/formal/02-06-65.pdf, Jun 2002
[9] Object Management Group, Deployment and Configuration of Component-based Distributed Applications Specification, http://www.omg.org/docs/ptc/04-08-02.pdf, Feb 2004
[10] Object Management Group, Model Driven Architecture, http://www.omg.org/docs/ormsc/01-07-01.pdf, Jul 2001
[11] Plasil, F., Balek, D., Janecek, R., SOFA/DCUP: Architecture for Component Trading and Dynamic Updating, Proceedings of ICCDS'98, Annapolis, Maryland, USA, IEEE CS Press, May 1998
[12] Sun Microsystems, Inc., Java 2 Platform Enterprise Edition Specification, version 1.4, http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf, Nov 2003