Towards Integrating Java EE into ProtoCom*

Daria Giacinto, Sebastian Lehrig

University of Paderborn
Zukunftsmeile 1
33102 Paderborn
giacinto@mail.upb.de
sebastian.lehrig@upb.de

Abstract: A key concept of model-driven software development is the transformation of models into other models or source code. ProtoCom is such a transformation that generates a performance prototype from a Palladio Component Model (PCM) instance by means of a model-to-text transformation. The actual supported platform, on which the PCM instance is mapped, is Java SE. Even though related work suggests that multiple platforms should be supported, their concrete integration into ProtoCom is only conceptual. For instance, Java EE has been investigated as a possible target platform, however, ProtoCom lacks its integration as well as the consideration of the current Java EE standard. Therefore, we provide a novel conceptual mapping from PCM to Java EE, thus, allowing to implement a transformation that realizes the mapping. As basis for this implementation, we also provide an initial Java EE reference implementation of a simple example PCM model.

1 Introduction

Model-driven software development deals with the development of software systems on the basis of models. These models are transformed into another model or source code, e.g., an architectural model is transformed into an implementation of the modeled system. One application of model-driven techniques is the automatic generation of performance prototypes. For this, performance engineers transform a model of the architecture into prototyped source code, usable for performance measurements [BDH08]. These engineers subsequently take measurements in a test environment. The prototype indicates how the system, once realized, will behave in the real production environment.

In our preliminary work [LLK13], we argue that performance prototyping should support a wide range of target platforms based on Java SE, Java EE, C#, Python, etc. The reason for this need is that different platforms crucially impact the performance of a system. Also in practice, a high variety of different target platforms is used. For example, in the context of cloud computing, typical target platforms support Java EE (CloudBees, CloudJee, Open-

*The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant no 317704 (CloudScale).

Proc. Kieker/Palladio Days 2013, Nov. 27–29, Karlsruhe, Germany
Available online: http://ceur-ws.org/Vol-1083/

Shift, Oracle Cloud), C# (Microsoft Azure), or Python (Google App Engine). However, current work on performance prototyping considers only a small set of these languages, thus, restricting analysis results to specific target platforms.

Palladio is one approach in the area of model-driven generation of performance prototypes. It facilitates the modeling of component-based architectures with a focus on performance-relevant information. For this modeling, Palladio comprises the Palladio Component Model (PCM), a component-based architecture description language (ADL). This language allows performance engineers for early design-time performance analyses, e.g., via performance prototyping. To generate a performance prototype from a PCM instance, ProtoCom is used. The current ProtoCom version transforms PCM instances into a Java SE implementation. Technically, other platforms are currently unsupported, even though concepts for Java EE have been investigated [Bec08]. Besides Palladio, we are currently unaware of any approach that transforms an ADL to a performance prototype.

To cope with the lack of supported target platforms, we plan and investigate the integration of Java EE into Palladio's ProtoCom. We decided to start with Java EE for multiple reasons. The mapping is already conceptually planned and prototypically implemented in the context of the PHD thesis by Becker [Bec08], thus, providing a point to start with. However, the initial implementation lacks full automation via a model-to-text transformation. Furthermore, Java EE evolved since the publication of Becker's thesis, thus, requiring an adaption of the work. Java EE is also interesting because it provides services that make the transformation more simple compared to Java SE. For example, Java EE's EJB containers support the use of dependency injection. In contrast, the transformation from PCM to Java SE has to use different design patterns to facilitate dependency injection. Another advantage is that Java EE facilitates the development of component-based architectures. Thus, it provides direct concepts to build components. As these concepts are similar to the PCM concepts, they can be adapted more easily and preserve the information of the PCM better [Bec08]. From a practical point of view, several cloud computing providers use Java EE as a platform (see above).

The contribution of this paper is our novel mapping concept from PCM to Java EE performance prototypes. This concept considers the recent Java EE version (Java EE 7) and provides the basis for a full automation. We evaluate the applicability of this concept by a Java EE reference implementation based on a simple toy example. In particular, we use this toy example throughout this paper to illustrate our mapping concept.

This paper is structured as follows. Section 2 provides the foundations for the PCM to Java EE mapping. Section 3 introduces the toy example that is used to illustrate the mapping. The subsequent Sec. 4 describes the mapping from PCM to Java EE in detail. We evaluate the applicability of this mapping by the reference implementation described in Sec. 5. Finally, Sec. 6 summarizes the related work of this paper before Sec. 7 concludes.

2 Foundations

This section gives an introduction about the foundations of the mapping from PCM to Java EE performance prototypes. At first, the performance prototype generator ProtoCom is presented (Sec. 2.1). Afterwards, the current target technology Java SE and the future target technology (Java EE) are explained in Sec. 2.2.

2.1 ProtoCom

Prototyping allows to test in an early development phase whether the system meets extra-functional requirements. Furthermore, it facilitates to analyze the quality, e.g., predict the performance of the system [Bec08]. In performance prototype generation, a transformation generates a directly compilable performance prototype implementation from a model instance. This prototype can be used to test the performance on the target environment. Thereby, the prototype emulates the modeled system's resource demands.

In our case, we use ProtoCom that generates performance prototypes from PCM instances. These performance prototypes can be deployed on realistic hardware environments operating a JVM to simulate resource demands on processing resources [LZ11]. Simulation data can be used to analyze the performance on the target environment. For the prototype generation, ProtoCom uses a model-to-text transformation written in the programming language Xpand.

2.2 Target Technologies

Java SE The current transformation uses Java SE as the target platform. For communication between components, the Java SE prototype uses the Java RMI middleware.

Java EE The Java Enterprise Edition (Java EE) [DS13] supports the development of Java applications that implement enterprise services. These applications are made up of different components that are packed into Java EE modules ready for the deployment on the server. There are different module types available, e.g., Web modules contain servlet class files, EJB modules contain EJB class files and the Enterprise Archive (EAR) module that holds all other modules building the application.

Furthermore, each module can comprise an optional XML deployment descriptor that describes the module's content. These descriptors are optional because a developer can directly add annotations into the Java source code that provide the same information. At deployment, the server configures components according to annotations and/or deployment descriptors, respectively.

Java EE provides the ability to create distributed applications that are not only accessible from modules residing in the same application but also from different applications. The

communication between EJBs deployed on different servers is facilitated by the means of the Remote Method Invocation over Internet Inter-ORB-Protocol (RMI-IIOP) [DS13].

3 Example Scenario: The Alice&Bob-System

We illustrate our toy example, called *Alice&Bob*, in Fig. 1. This system consists of two server instances, for example two Glassfish servers. On one server, the *Alice* component is deployed that provides the interface *IAlice* with the method *callBob()*. The other server deploys the *Bob* component that provides the interface *IBob* with the method *sayHello()*. The *IAlice* interface is provided to a user who can invoke this method through a client-side technology like a browser. This invocation can be received by a servlet on server side that forwards the call to the specific component.

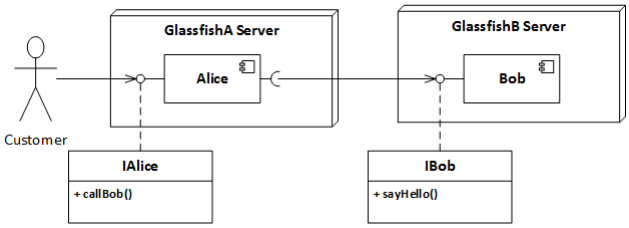

Figure 1: Alice&Bob System

4 PCM to Java EE Mapping

This section describes the mapping of a PCM instance to Java EE step-by-step with the help of the *Alice&Bob* system. Furthermore, the actual transformation to Java SE is explained to show the differences. The concepts of the mapping to Java SE are based on the PHD thesis by Becker [Bec08].

At first, the mapping of PCM's repository (Sec. 4.1), i.e., Interfaces, Basic Components including PCM's Service Effect Specification (SEFF), Provided and Required Roles and Composite Components, is described. Afterwards, the design decisions how to map PCM's assembly (Sec. 4.2), allocation (Sec. 4.3) and usage scenario (Sec. 4.4) are explained. The comparison of the Java SE- and Java EE mapping is summarized in Table 1.

4.1 Repository

The repository model consists of interfaces (Sec. 4.1.1) and basic components (Sec. 4.1.2) that can provide (Sec. 4.1.3) and/or require (Sec. 4.1.4) these interfaces. Furthermore,

PCM Concept	Java SE	Java EE
Interfaces	Java Interface	Java EE Business Interface
Basic Components	Classes with Simulated SEFF	Stateless Classes with Simulated SEFF bundled in an EJB Module, which is packed into an EAR Module
Provided Roles	Port Classes	Port Classes
Required Roles	Context Pattern	Dependency Injection/Annotation
Composite Components	Facade Class	EJB Modules bundled in an EAR Module
Assembly Context	Instance of Component Class	Deployed EAR Module
Assembly Connector	Deployment Descriptor	EJB Deployment Descriptor/Annotations
Internal Actions	Resource Demand Generator	Resource Demand Generator
Call Actions	RMI/SOAP Call	RMI-IIOP
Control Flow	Java Control Flow	Java Control Flow
Data Flow Annotations	Java Control Flow	Java Control Flow
Allocation Context	Manual Deployment	Manual Deployment
Resources	[Uses Physical Resources]	[Uses Physical Resources]
Usage Model	Workload Driver	Workload Driver

Table 1: Comparison of the Current and Future Mapping based on [BDH08, Bec08]

components can be composed to composite components (Sec. 4.1.5). The component developer is responsible for the repository model.

4.1.1 Interfaces

Interfaces specify a set of operations to be required or provided by components. The interface provided by an implementing component specifies which functionality other components can expect from it. The interface required by an component states which functionality is needed from a component implementing the interface. In the PCM, an interface describes the signatures of their operations. An operation consists of an operation name, an ordered list of parameters, a return type, and an unordered list of exceptions. Furthermore, interfaces are independent of a particular component. Interfaces are related to a component by the concept of roles, namely, required, and provided roles. Java interfaces are independent of a class and implementable by several classes. Therefore, they consist of a signature but no implementation.

In the current Java SE mapping, PCM interfaces are directly mapped to Java SE interfaces. In contrast, EJBs can additionally provide a *business interface*. A *business interface* is a standard Java interface that contains only the business methods of an enterprise bean. The difference of the Java EE *business interface* to the Java SE interface is that the Java EE *business interface* defines the client's view of the EJB. The client can only invoke the methods stated in the *business interface*. Method implementations and deployment settings are hidden from the client. This mechanism better facilitates that another component is restricted to component methods that are stated in the interface, even if the component provides further methods, e.g., because it implements additional interfaces. Furthermore, it is possible to annotate the interface as *local* or *remote*. The choice depends on whether

local or remote access to the implementing enterprise bean should be allowed.

For the Java EE mapping of our example system, the *Alice*- and *Bob* are business interfaces annotated with *@remote*. The reason for this annotation is our intention to deploy the implementing components (*Alice* and *Bob*) on two different servers.

4.1.2 Basic Components

Basic Components are components whose implementation cannot be decomposed into further components. Basic Components can be formed by objects in an object-oriented language. One class per component is needed in Java SE/EE as a minimum to mimic the PCM Basic Component.

In our example mapping, there exists an *Alice* class and a *Bob* class. The EJB classes contain the additional annotation *@stateless*, as it is a Palladio assumption that components are stateless. Hence, the EJB saves no information about a state. Each class that constitutes to the Basic Component is packed into an EJB module. An EJB module is bundled in an EAR module. In this way, EJB modules can be reused and directly deployed on the server. The specification of the behavior for each provided operation (PCM's Service Effect Specifications; SEFFs) is mapped to Java EE code emulating the resource demands. They can be adapted from the PCM to Java SE mapping.

4.1.3 Provided Roles

As shortly mentioned in the section about interfaces (Sec. 4.1.1), provided and required roles are the means to relate an interface to a specific component. These roles are used to specify the component's functionality and behavior. The provided role states which interface is provided by the component and, thereby, specifies its offered functionality. Thus, the provided role is equal to the understanding of a class implementing an interface in Java EE and Java SE, respectively.

In our example, the *Alice* component and the *Bob* component have a provided role. The *Alice* component provides the *IAlice* interface and the *Bob* component the *IBob* interface.

In the PCM, it is possible to have a component with different roles that offer the same operation, i.e., the operations have an identical signature. This cannot be expressed by Java. Therefore, a *port* class is introduced as a workaround. This port class realizes the proxy pattern [GHJV95]. For example, the *IAlicePort* implements the *IAlice* interface and forwards the call from the *callBob()* method to the *Alice* class that implements the concrete functionality of this method. The method names in the component class get a unique name, showing the relation to the particular port. These unique names solve the issue of having the same provided operations in one class.

4.1.4 Required Roles

A required role specifies the needed interface implemented by a component. The transformation has to regard that the required roles can be initialized differently, depending on the assembly. Therefore, this information should be implemented outside of the component. In Java SE, there exists no concept of explicit required interfaces. Hence, in the current Java SE mapping, the dependency injection- and context pattern is used by the transformation to guarantee decoupling [Fow04, SVC06].

In contrast to Java SE, Java EE has dedicated support for required interfaces. Required interfaces are specified by annotations. These annotations are then resolved by the EJB container at run time through dependency injection. Dependency injection obtains references to resources at run time without the direct instantiation of them. The declaration of the dependency is stated by an annotation in the source code in front of the use of the resource. This annotation is used as an injection point, e.g., *@EJB*. Thereby, the server can provide the required instance at run time. In an annotation, usually the interface name is stated as the type for the injected instance. This supports the decoupling of the code from a specific implementation. When the required resource is deployed on the same server, annotations can be used to declare the dependency. In the example *Alice* class, the injected bean is given the name *IBob* by means of the *name* attribute of the *EJB* annotation. The annotation is directly followed by the declaration of the type and name of the required bean: the *Alice* class requires an object of type *IBob*.

In case of a remote resource, the remote location has to be specified in the deployment descriptor of the EJB module that contains the component requiring a remote component. As the *Alice* component requires an instance of the remote component implementing the *IBob* interface, the location is specified in the EJB module deployment descriptor containing the *Alice* class. In the descriptor, the name assigned in the EJB annotation to a required instance is resolved to a remote location. The particular required component is identified through a portable Java Naming and Directory Interface (JNDI) name. Remote access to an EJB is based on the Common Object Request Broker Architecture (CORBA). It enables the interaction of applications written in different languages over a network. For this, the provided interfaces of objects are presented by an Interface Definition Language (IDL) that is mapped to the particular programming language. Client- and target Object Request Broker (ORB) typically use the Internet Inter-ORB-Protocol (IIOP) as a common protocol. EJBs deployed on different servers communicate by the means of the Remote Method Invocation over Internet Inter-ORB-Protocol (RMI-IIOP) [DS13]. Therefore, the IP address has to be specified in the deployment descriptor. In combination with the JNDI name it facilitates the access to the remote EJB.

4.1.5 Composite Components

Composite Components consist of a set of inner components that can be Basic Components and Composite Components. These Composite Components combine the functionality of its inner components to offer its own functionality. They are disregarded in the mapping and left as a future work. A first idea how they can be mapped to Java EE would

be to create for each inner component an EJB module and combine them in an EAR module that represents the Composite Component.

4.2 Assembly

Existing components are assembled into a system that can be deployed. A system can have provided- and required roles, too. A system's role is explicitly added and delegated to a provided or required role of its inner components. In Java EE, the inner components of a system are the particular EAR modules, each holding a Basic Component or Composite Component.

In a PCM system, the inner components are connected by *Assembly Connectors*. The interaction between EAR modules is facilitated by JNDI and the deployment descriptor or annotations, respectively. At the moment, the deployment descriptor of the EJB module specifies the concrete connection to a required interface. This setting of the remote location of a bean may be changed and stated in the EAR deployment descriptor because the connection is unknown until the deployment. When it is stated there, the system deployer can easily adapt the deployment descriptor to the designed allocation.

4.3 Allocation

The allocation comprises the connection of components to executing hardware resources. In the current mapping, the component deployer has to do the allocation manually. Namely, the deployer has to export the files constituting one EAR module as an EAR file and deploy it on the server running on the specific hardware node. However, as a future work we consider a full automated deployment.

4.4 Usage Scenario

The usage scenario model of the PCM specifies the behavior of users of the system. It models user interaction and the data that users exchange with the system. In the current mapping, a workload driver emulates the behavior of a user as it is specified in the usage scenario model.

In case of Java EE as a target platform, there exist two ways of generating workload. One possibility is to generate workload over the http interface by the use of a tool like Apache JMeter [Fou]. Another possibility is the use of a client application that generates a workload over the RMI-IIOP interface.

The interaction with the *Alice&Bob* system is facilitated by the use of a servlet that serves as the system's interface. The modeled user invokes the method of the servlet through the browser. This method calls the *callBob()* method of the *IAlice* interface.

5 Reference Implementation

To evaluate the applicability of our mapping concept, we implemented a reference implementation that realizes the *Alice&Bob* system[1]. Based on successfully operating this reference implementation on two Glassfish servers (i.e., in a distributed fashion), we conclude that our mapping concept is indeed applicable.

In contrast to Becker's implementation [Bec08], we provide an implementation fully utilizing Java EE features: business interfaces, annotations, RMI-IIOP communication, and EAR modules. Firstly, Becker used normal Java SE interfaces, thus, neglecting the use of dedicated access restriction mechanisms. Secondly, instead of annotations, Becker applied XML deployments that generally lower the maintainability of EJBs and, consequently, a possible prototype generation. Thirdly, according to the Java EE specification [DS13], RMI-IIOP should be used for communication between EJBs on different servers. Becker used RMI communication that can have a different performance impact and can be an unwanted realization technology (as not standard conform). Finally, Becker lacks support for deployable EAR modules and relies on a manual deployment of implementation code. In contrast, our concept focuses on making prototypes deployable out-of-the-box as EJB modules.

6 Related Work

In his PHD thesis [Bec08], Becker describes a ProtoCom supporting Java SE and Java EE. However, Java EE changed since the publication of his thesis. For example, annotations are now commonly used instead of deployment descriptors. Furthermore, his Java SE and Java EE mappings lack full automation, thus, requiring to manually adjust the prototypes. We revise the Becker's Java EE mapping to cope with these issues.

Lehrig and Zolynski [LZ11] provide an improved version of ProtoCom. They argue that Becker's version suffers from usability issues (incomplete prototypes, inefficient deployment process). To eliminate these usability issues, they improve the transformation to Java SE. This new ProtoCom generates the complete performance prototype and eases the deployment process as no application servers have to be configured. They also removed the support for Java EE as the mapping was outdated and lacked automation. However, because performance prototyping should support a wider range of languages, we work towards integrating Java EE, again. For this integration, we address usability issues similar as Lehrig and Zolynski do for Java SE.

[1]The implementation can be accessed via `https://svnserver.informatik.kit.edu/i43/svn/code/Palladio/Incubation/JEEProtoCom/AliceAndBob` (User: anonymous; Password: anonymous; Visited on 14/11/2013).

7 Conclusions

In this paper, we present a conceptual mapping from PCM to Java EE, evaluated by a reference implementation of a toy example. We also show the differences between mappings to Java SE and Java EE. Our findings serve performance engineers as a starting point for implementing Java EE performance prototypes. Especially our reference implementation is the basis for an automated transformation from PCM to such prototypes.

In our future work, we have to investigate mapping concepts currently disregarded. These concepts are, e.g., SEFFs and Composite Components. Moreover, we plan to fully automate the generation of Java EE performance prototypes.

References

[BDH08] Steffen Becker, Tobias Dencker, and Jens Happe. Model-Driven Generation of Performance Prototypes. In Samuel Kounev, Ian Gorton, and Kai Sachs, editors, *SIPEW*, volume 5119 of *Lecture Notes in Computer Science*, pages 79–98. Springer, 2008.

[Bec08] Steffen Becker. *Coupled model transformations for QoS enabled component-based software design.* PhD thesis, Carl von Ossietzky University of Oldenburg, 2008.

[DS13] Linda DeMichiel and Bill Shannon. Java TM Platform Enterprise Edition 7. Technical report, Orcale, 2013.

[Fou] Apache Foundation. Apache JMeter. `http://jmeter.apache.org/` [Visited on 26/10/13].

[Fow04] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, 2004. `http://martinfowler.com/articles/injection.html` [Visited on 23/10/13].

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[LLK13] Michael Langhammer, Sebastian Lehrig, and Max E. Kramer. Reuse and configuration for code generating architectural refinement transformations. In *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, VAO '13, page 6:16:5, New York, NY, USA, 2013. ACM.

[LZ11] Sebastian Lehrig and Thomas Zolynski. Performance Prototyping with ProtoCom in a Virtualised Environment: A Case Study. In *Proceedings to Palladio Days 2011, 17-18 November 2011, FZI Forschungszentrum Informatik, Karlsruhe, Germany*, 2011.

[SVC06] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.