Transformations from EDOC to EJB by Composition of Mapping Operations

Dariusz Gall

Computer Science and Management Faculty, Wrocław University of Technology, Wybrzeże Wyspiańskiego 27, 50-370 Wrocław, Poland
dariusz.gall@pwr.wroc.pl

Abstract. Transformations from the Enterprise Distributed Object Computing (EDOC) to Enterprise JavaBeans (EJB) are examined herein from the point of view of an efficiency characteristic of result models. Transformations are composed of mapping operations. Mapping operations map an element or a group of elements of the EDOC model into an element or a group of elements of the EJB model. A standard proposal of mapping operations is defined by the Object Management Group (OMG). Alternative mapping operations, which generate EJB models of a better efficiency characteristic under certain circumstances, are proposed. Rules for composing mapping operations into EDOC – EJB transformations are suggested and illustrated by examples of EDOC – EJB transformations.

Keywords: model transformation, Model-Driven Development, EDOC, EJB, efficiency

1 Introduction

Transformations play a key role in Model-Driven Development (MDD) [15]. We consider transformations from the Enterprise Distributed Object Computing (EDOC) to Enterprise JavaBean (EJB), which translate a model expressed in the EDOC language to a model in the EJB language.

Transformations have to be correct, which informally means that they have to generate a target model without “losing” information included in a source model. Such transformation is proposed in [12]. However, it is not sufficient to produce correct models, because transformations should also meet additional requirements. We consider a good efficiency of generated model to be an additional requirement of EDOC – EJB transformations.

The aim of the paper is to propose an alternative transformation that under special circumstances may give better results. We present the transformation and compare it to the standard transformation [12]. The scope of the paper is though limited to structural aspect transformations of a subset of the EDOC, the Component Collaboration Architecture (CCA). Such transformations to EJB and other platforms are considered in [1], [2], and [13].

The paper consists of following sections: in Section 2, an introduction to CCA and EJB is given. In the main Section 3, the alternative transformation is described and compared to the standard transformation. In Section 4, example transformations' executions are illustrated. The paper is concluded in Section 5.

2 Domain and codomain of transformation

A transformation can be considered as a function, which domain is a set of all CCA models and codomain is a set of all EJB models.

The CCA is a language for describing software architectures [9]. The OMG has prepared the CCA UML Profile [11], and the CCA model can be represented using UML. Only subset of the CCA concepts, required to understand the paper, is herein presented, in particular: a *Process Component*, a *Port* and its specialisation a *Flow Port*. A *Process Component* represents some process in a modelled system. A *Process Component* is modelled by a *Classifier* with stereotype *«ProcessComponent»*. A *Process Component* can be composed of other *Process Components*. A *Port* is *Process Component*'s connection point. Each *Port* has properties, for instance, considered *Direction* – direction of sending the first message in an interaction. The simplest subclass of a *Port* is a *Flow Port* represented by a *«FlowPort»* stereotype. It is capable for generating or consuming single typed data.

The EJB is a middleware platform [14]. The Java Community Process (JCP) has prepared the standard representation of EJB-based software artefacts in the form of the EJB UML Profile. A subset of the EJB concepts, required to understand the paper, includes a *Session Bean*, an *Implementation Class*, a *Session Home Interface*, a *Remote Interface*, and a *Remote Method*. A *Session Bean* is a server-side component. It models a business process; it can perform certain actions [14]. It is represented by a *Subsystem* with stereotype *«EJBSessionBean»*, which consists of an *Implementation Class*, a *Session Home Interface*, and a *Remote Interface*. An *Implementation Class* contains implementation of a *Session Bean*. It is expressed by a *Class* with stereotype *«EJBImplementation»*. A *Session Home Interface* is an interface to an object responsible for creating and destroying of *Session Beans*. It is modelled by a *Class* with stereotype *«EJBSessionHomeInterface»*. A *Remote Interface* contains business methods available for clients or other enterprise beans using remote method calls. It is modelled by a *Class* with stereotype *«EJBRemoteInterface»*. Methods available within a *Remote Interface, Remote Methods,* are modelled using an *Operation* with stereotype *«EJBRemoteMethod»* [6].

3 Transformations

Operational transformations are considered herein [5]. In the operational transformation approach, a transformation is a function taking as an argument a source model, and returning a target model. The function can be decomposed into mappings, which are responsible for transforming an element or a group of elements of a source model into an element or a group of elements of a target model. The mappings are required to be correct in the same senses as the whole transformation. An operational transformation is realized by a sequence of the mappings, each time returning an intermediate model until a target model is generated.

3.1 Transforming CCA to EJB

A way for transforming a CCA model to an EJB model by mappings between CCA and EJB is described in paper [12]. In Table 1, mappings limited to these examined herein, are presented.

Model element	
CCA	***EJB***
Process Component	*Session Bean*
Flow Port	In *Implementation Class* - an *Java Method* without return value, in *Remote Interface* – an *Remote Method* without return value

Table 1*. Mappings between CCA and EJB model elements [12]*

Transformations built according to the mapping in the paper [12], in particular Table 1, generate correct EJB models and are able to transform a specification of any system expressed in the CCA. However, the proposal given in the paper [12], does not avoid pitfalls related to remote methods call or a *Session Bean* component's life cycle, which can cause a low overall performance of a system [7], [8], [14].

An alternative proposal is to change a *Session Bean* to a *Java Class*. A *Process Component* may be represented by a *Java Class,* if it is a part of a compound *Process Component* and interacts with, transformed to *Java Classes, Process Components* of the compound one or interact with the compound one. This change entails changes in the mappings of *Ports. Ports* of a *Process Component* transformed to *Java Class* do not have to be represented as remote method. The rationale behind this way of *Process Component* mapping is that during a *Session Bean* component's life cycle usually more resources are used than during a pure Java object's life cycle. Thus, it is reasonable to substitute it, where possible, in particular because of performance issues [3], [7].

Model element	
CCA	***EJB***
Process Component	*Java Class*
Flow Port	In *Java Class* - an *Java Method* without return value

Table 2. *Enhancements of the mappings from Table 1*

The enhancement of the mappings from Table 1, as alternative mappings of *Process Component* and *Ports* is presented in Table 2. The proposed enhancement does not ensure that a transformation defined according to it will generate a model of a better efficiency characteristic. Nevertheless, if a system specification in the CCA is made in accordance with good design practices, e.g., low coupling and high cohesion of *Process Components*, alternative mappings might be used in more situations and better performance of an application may be expected.

3.2 Building transformations

We consider CCA – EJB transformations as an unidirectional transformation, defined on set of mappings.

Let *SOURCE* be a set of elements of a source model and *TRANSF* be a set of a source model elements that have been already mapped during a transformation, $TRANSF:=\emptyset$. Let *INTER* be a set of intermediate model's elements, $INTER:=\emptyset$. Let *TARGET* be a set of elements of a target model.

During a transformation, mappings are executed on elements of a *SOURCE* and *INTER.* After each execution of a mapping, an *INTER* is updated by a result of the mapping. If the transformation is finished an *INTER* become a *TARGET.*

Let *MAP* be a set of mappings. A mapping is a function, two kinds of mappings are possible, a *new* mapping $new\_map{:}2^{SOURCE} \rightarrow 2^{INTER}$, an *update* mapping $update\_map{:}2^{SOURCE} \times 2^{INTER} \rightarrow 2^{INTER}$. A *new* type mapping takes one argument, a group of a source model's elements, which are going to be mapped, and returns a group of an intermediate model's elements that are created. An *update* type mapping takes two arguments, a group of a source model's elements, which are going to be mapped, and a group of an intermediate model's elements, which properties are going to be updated. An *update* type mapping returns a group of elements of an intermediate model, which contains updated and/or created elements of an intermediate model.

Let $pre{:}MAP \times 2^{SOURCE} \times 2^{SOURCE} \times 2^{INTER} \rightarrow Boolean$ be a Boolean function, which is a precondition for execution of a mapping from *MAP.* The function arguments are the mapping, a group of elements from 2^{SOURCE}, a context of the group of elements from *SOURCE*, and elements of a target model from *INTER.*

Let $to\_update{:}MAP \times 2^{SOURCE} \times 2^{SOURCE} \times 2^{INTER} \rightarrow 2^{INTER}$ be a function, which returns a group of elements that are going to be updated by a mapping from *MAP.* The function arguments are the mapping, a group of elements from 2^{SOURCE}, a context of the group of elements from *SOURCE*, and elements of a target model from *INTER.*

Mappings are executed according to the schema:

1. Select arbitrary $e \in 2^{SOURCE}$, and $m \in MAP$, such that $pre(m,e,SOURCE,INTER)$ returns *true*.
2. If a map *m* is a *new* type of mapping then
   $m(e)=e'$; $INTER:=INTER \cup e'$.
   If a map *m* is an *update* type of mapping then
   $to\_update(m,e,SOURCE,INTER)=e'$; $m(e,e')=e''$; $INTER:=INTER - e' \cup e''$.
   Lastly, $TRANSF := TRANSF \cup e$.
3. If $TRANSF <> SOURCE$ then go to step 1 otherwise, $TARGET:=INTER$, and stop execution of the schema.

It may happen that several alternative transformations are possible. In this situation, we should consider models generated by these transformations and choose a transformation, which generates a model of the best quality characteristics.

3.3 Mapping Operations

Mapping operations are realisation of mappings between models, and in this work we focus on mappings between the CCA and the EJB elements.

A mapping operation contains a pre-condition and a mapping body, see Listing 1. A pre-condition is a condition that is a prerequisite for a mapping execution, and is expressed in *when* section. A mapping body specifies an algorithm of creation and adding new elements to a target model, or modification of a target model's elements.

Mapping operations of CCA elements to EJB elements, limited to proposed alternative mappings, are presented herein. Four mapping operations are presented, see Listing 1, and each is a realisation of the mappings in Table 1, and Table 2. They are expressed using MOF QVT Operational Mapping language [10].

The pre-condition is a composition of three types of constraints, see Listing 1. A first type of the constraints checks a mapping applicability for an element or a group of elements. Next type of constraint is a constraint on context of mapped element within an input model. Using this type of constraints, we specify context of an element, prerequisite to execute a mapping operation. Third constraint type checks whether an element or group of elements were mapped by a particular mapping.

The mappings presented in the Listing 1, reflect the transformation model proposed in Section 3.2. *PCtoEJBSessBean* and *PCtoJavaClass* are *new* type mappings. The *pre* function is realized in *when* sections of the mappings. *FlowPortToRemMethEJBSessBean* and *FlowPortToOperJavaClass* are *update* type mappings. A *pre* function is realized in *when* clauses of the mappings, and a *to_update* function is realized in *init* sections of these mappings.

Mappings from Table 1	**Mappings from Table 2**
Mapping: *Process Component to Session Bean*	**Mapping**: *Process Component to Java Class*

Mapping: *Process Component to Session Bean*

```
mapping Class::PCtoEJBSessBean()
when {
 // pre-condition:
 is_pc(self) // type of element
}
{
 // mapping algorithm
 population{
  object ejbBean:Subsystem{
   name:=self.name;
   stereotype += getStereotype('EJBSessionBean');
   ownedElement += object Class {
    stereotype +=
      getStereotype('EJBImplementation');
   }
   // similar algorithm for EJBRemoteInterface
   // and EJBSessionHomeInterface
 }
 }
 // outputModel is a global variable
 // representing an EJB model
 outputModel.ownedElement += ejbBean;
}
```

Mapping: *Process Component to Java Class*

```
mapping Class::PCtoJavaClass()
when {
 // pre-condition:
  // type of element
 is_pc(self) and
  // input model context
 can_be_mapped_to_JC(pc)
}
{
 // mapping algorithm
 population{
  object jc:Class{
   name:=self.name;
  }
 }
 // adding to an output model
 outputModel.ownedElement += jc;
}
```

Mapping: *Flow Port to Remote Method of Session Bean*

```
mapping Class::FlowPortToRemMethEJBSessBean()
when {
 // pre-condition:
  // type of element
 is_flow_port(self)
 and is_responds_port(self) and
  // input model context
 is_associated_with_pc(self) and
  // mapping context
 was_mapped_by(PCtoEJBSessBean,
  get_associated_PC(self))
}
{
 init{ejbSessBean:=get_mapping_result
   (get_associated_PC(self));
    ejbImpl:=ejbSessBean.getEJBImplementation();
     // similar for ejbRemote
```

Mapping: *Flow Port to Operation in Java Class*

```
mapping Class::FlowPortToOperJavaClass()
when {
 // pre-condition:
  // type of element
 is_flow_port(self)
 and is_responds_port(self) and
  // input model context
 is_associated_with_pc(self) and
 // mapping context
 was_mapped_by(PCtoJavaClass,
 get_associated_PC(self))
}
{
 init{javaClass:=get_mapping_result
   (get_associated_PC(self));
    }
 // mapping algorithm
```

```
   }
// mapping algorithm
population{
 object implOper:Operation{
  name:=self.name+'Responds';
  parameter=object Parameter{
   kind=ParameterDirectionKind.in;
  }
 }
 // similar algorithm for remotOper
}
// updating elements of an output model
ejbImpl.feature += implOper;
ejbRemote.feature += remotOper;
}
```

```
population{
 object operation:Operation{
  name:=self.name+'Responds';
  visibility:=VisibilityKind.public;
  parameter=object Parameter{
   kind=ParameterDirectionKind.in;
  }
 }
}
// updating elements of an output model
javaClass.feature += operation;
}
```

***Listing 1**. CCA - EJB Mappings*

4 Example transformations

We present two possible transformations of a CCA model, which contains a compound *Process Component*: *Seller*. The *Process Component* is composed of two *Process Components*: *QuoteCalculator* and *SellerOrders*, see Figure 1.

First, we consider a transformation made according to mappings proposed in Table 1. It is a composition of mapping operations in a following order: mapping operation *PCtoEJBSessBean* executed for elements: *Seller, QuoteCalculator* and *SellerOrders*, and mapping operation *FlowPortToRemMethEJBSessBean* for elements: *Quote, Order, QuoteCalculator* and *SellerOrders*. A result of the transformation is shown in the Figure 2. As we could predict, three *Session Beans* are created corresponding to *Process Components* in the source model.

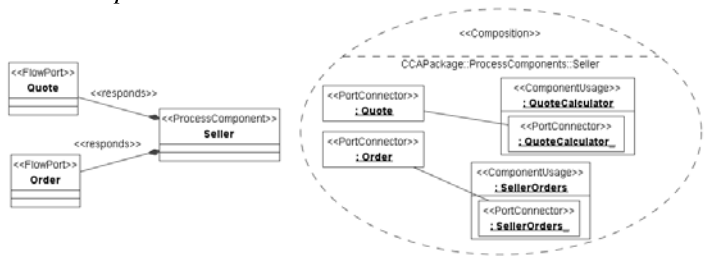

Figure 1. *Input model - the compound Process Component Seller [12]*

Now, let us consider a transformation based on mappings in the both Table 1 and Table 2. The transformation is a composition of mapping operations in a following order: *PCtoEJBSessBean* for *Seller*; *PCtoJavaClass* for *QuoteCalculator*, *SellerOrders*; *FlowPortToRemMethEJBSessBean* for *Quote* and *Order*; *FlowPortToOperJavaClass* for *QuoteCalculator* and *SellerOrders*. A result of the transformations is shown in the Figure 3. One *Session Bean* corresponding to *Seller* is created, and two *Java Classes* corresponding to *QuoteCalculator* and *SellerOrders* in the source model are created.

In the first output model (Figure 2), three *Session Beans* are created, instead of one *Session Bean* in the second output model (Figure 3). Hence, in the first case, a lot of communication has to be done using remote method invocation, because each *Process*

Components are represented by different *Session Bean*. In the second case, *Process Components* are transformed to a *Session Bean* and *Java Classes* and such should decrease a number of remote method's invocations. It seems that more execution time of an EJB server is spent on communication in the first case, than in the second case [4].

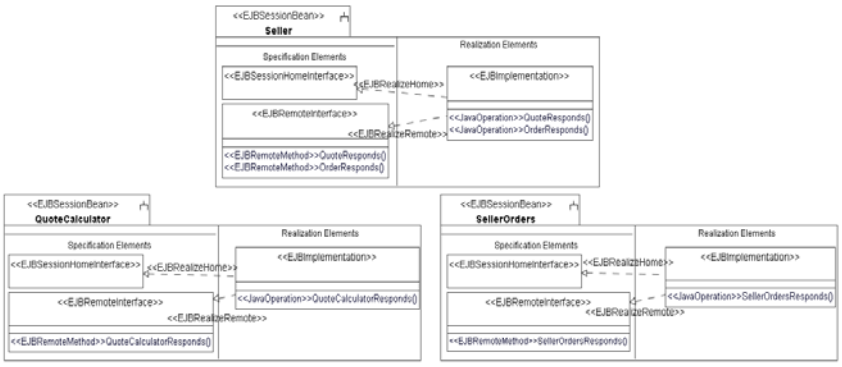
<<EJBSessionBean>>
Seller
Specification Elements
Realization Elements
<<EJBSessionHomeInterface>>
<<EJBRealizeHome>>
<<EJBImplementation>>
<<EJBRemoteInterface>>
<<EJBRealizeRemote>>
<<JavaOperation>>QuoteResponds()
<<JavaOperation>>OrderResponds()
<<EJBRemoteMethod>>QuoteResponds()
<<EJBRemoteMethod>>OrderResponds()
<<EJBSessionBean>>
QuoteCalculator
<<JavaOperation>>QuoteCalculatorResponds()
<<EJBRemoteMethod>>QuoteCalculatorResponds()
<<EJBSessionBean>>
SellerOrders
<<JavaOperation>>SellerOrdersResponds()
<<EJBRemoteMethod>>SellerOrdersResponds()

Figure 2. *First possible output model - only Session Beans*

However, it might be a different situation if an application, generated according to the first model (Figure 2), is deployed on a cluster of EJB servers. The cluster might be able to copy and put each *Session Bean* component individually on a different server participating in the cluster. Such being the case, the *Session Beans* of the first model can be individually distributed among the servers. Hence, an efficiency of the model in the Figure 2 might be better than an efficiency of the second model (Figure 3) [14].

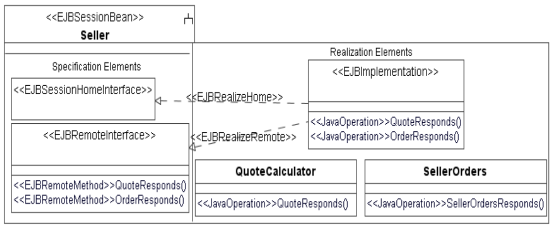
<<EJBSessionBean>>
Seller
Specification Elements
Realization Elements
<<EJBSessionHomeInterface>>
<<EJBRealizeHome>>
<<EJBImplementation>>
<<EJBRemoteInterface>>
<<EJBRealizeRemote>>
<<JavaOperation>>QuoteResponds()
<<JavaOperation>>OrderResponds()
<<EJBRemoteMethod>>QuoteResponds()
<<EJBRemoteMethod>>OrderResponds()
QuoteCalculator
<<JavaOperation>>QuoteResponds()
SellerOrders
<<JavaOperation>>SellerOrdersResponds()

Figure 3. *Second possible output model - Session Bean and Java Classes*

Thereby, the choice of the transformation is an individual case depending on requirements of a developed system. Referring to the example, if we deploy an application on a single EJB server, then we will probably choose the transformation to the model in the Figure 3. However, if we deploy the application on a cluster of EJB servers, then it will be better to consider the transformation to the model in the Figure 2.

5 Conclusions

The main goal of this paper has been to examine transformations from EDOC to EJB, taking into consideration an efficiency aspect of generated models. The transformations have been composed of mappings. The approach for building transformation has been proposed.

We have considered a set of mappings presented by the OMG, and we have proposed alternative mappings in order to achieve EJB models of a better efficiency characteristic. We have shown the proposal of implementation of these mappings using mapping operations introduced in the MOF QVT [10]. We have given examples of EDOC-EJB transformations and have discussed the transformations results.

The suggested herein transformation approach gives an ability to build many transformations on a set of the mappings. Results of these transformations can be examined from the perspective of an efficiency and other quality characteristics. However, these transformations are not able to generate a complete EJB application. Important problem are behavioural aspect mappings and generating an EJB model, which contains both structure, and behaviour of a system.

References

1. Belaunde, M., and Peltier, M., "From EDOC Components to CCM Components: A Precise Mapping Specification", FASE 2002, 2002, p. 1-16.
2. Born, M., Blazarenas, A., Funabashi, M., Hirai, C., Kath, O., Ritter, T., and Soden, M., "An Open Modeling Infrastructure integrating EDOC and CCM", IEEE, 2001.
3. Broemmer, D., *J2EE - Best Practices, Java Design Patterns, Automation and Performance*, John Wiley and Sons, 2003.
4. Cecchet, E., Marguerite, J., and Zwaenepoel, W., "Performance and Scalability of EJB Applications", Proc. 17th Ann. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, 2002, pp. 246-261.
5. Czarnecki, K., and Helsen, S. "Feature-based survey of model transformation approaches", IBM Systems Journal, Vol 45, No 3, 2006.
6. Java Community Process, *Java UML/EJB Mapping Specification*, http://www.jcp.org/jsr/detail/26.jsp, 2001.
7. Kalidindi, R., and Datla, R., *Best Practices to improve performance in EJB*, http://www.precisejava.com/javaperf/j2ee/EJB.htm, Nov. 2001.
8. Marinescu, F., *EJB Design Patterns - Advanced Patterns, Processes, and Idioms*, John Wiley and Sons, 2002.
9. Object Management Group, *Enterprise Collaboration Architecture (ECA) Specification*, formal/2004-02-01, 2004.
10. Object Management Group, *MOF QVT Final Adopted Specification*, ptc/05-11-01, 2005.
11. Object Management Group, *UML Profile for Enterprise Collaboration Architecture Specification*, formal/2004-02-05, 2004.
12. Object Management Group, *UML Profile for Enterprise Distributed Object Computing, Part II Supporting Annexes*, ad/2001-08-20, 2001.
13. Patrascoiu, O., "Mapping EDOC to Web Services using YATL", Proceedings of the 8th International IEEE Enterprise Distributed Object Computing Conference (EDOC 2004), September 2004, pp. 286-297.
14. Roman, E., Sriganesh, R. P., and Brose, G., *Mastering Enterprise JavaBeans*, John Wiley and Sons, 2005.
15. Sendall, S. and Kozaczynski, W., "Model transformation - the heart and soul of model-driven software development", IEEE Software, Special Issue on Model Driven Software Development, 20(5):42–45, 2003.