Platform Independent Model Transformation Based on Triple

Andreas Billig[1], Susanne Busse[2], Andreas Leicher[2], and Jörn Guy Süß[2]

[1] Fraunhofer ISST, Berlin, Germany
Andreas.Billig@isst.fhg.de
[2] Technische Universität Berlin, Germany
{sbusse,aleicher,jgsuess}@cs.tu-berlin.de

Abstract. Reuse is an important topic in software engineering as it promises advantages like faster time-to-market and cost reduction. Reuse of models on an abstract level is more beneficial than on the code level, because these models can be mapped into several technologies and can be adapted according to different requirements. Unfortunately, development tools only provide fixed mappings between abstract models described in a language such as UML and source code for a particular technology. These mappings are based on one-to-one relationships between elements of both levels. As a consequence, it is rarely possible to customize mappings according to specific user requirements.
We aim to improve model reuse by providing a framework that generates customized mappings according to specified requirements. The framework is able to handle mappings aimed for several component technologies as it is based on an ADL. It is realized in Triple to represent components on different levels of abstraction and to perform the actual transformation. It uses feature models to describe mapping alternatives.

1 Introduction

In general, software development consists of several steps from which one is the design. The design is often expressed by a modeling language such as UML and sketches the specification of a system. It is common practice to translate the result of the design into source code. This is done either manually or automatically. Unfortunately, the link between design and source code often ends after the transformation step and the design is not kept up-to-date any longer. This has several disadvantages, particularly for reuse.

At present, reuse mainly manifests itself in artifacts: Either source code or commercials off-the-shelf (COTS) are reused. However, this kind of reuse is restricted to a small number of situations, where we can directly reuse an artifact. If we, for example, develop a system based on the .Net technology and we have an existing component that exactly fulfills a particular service but is written in CORBA, we can't directly reuse that component. Instead, we have to construct appropriate wrapper code to adapt the component to the actual environment. This often brings some disadvantages such as performance penalties etc. Unfortunately, we cannot simple extract the design from existing code and build an

H.-A. Jacobsen (Ed.): Middleware 2004, LNCS 3231, pp. 493–511, 2004.

adapted component from it, because code is the result of the amalgamation of both design and technology. It is difficult to recognize design in large projects, because it tends to get lost in the brittle structure of libraries and code. As a consequence, design reuse is hard to achieve, as it is not supported in current software development.

For that reason, Model-Driven Development [12] proposes to model systems on a higher level of abstraction – independent of a particular technology – and to automatically transform a design into code. This kind of development promises faster time-to-marked and cost reduction because specifications on a more abstract level simplify development. If we for example have the design specification of the CORBA component mentioned above, we can automatically generate a .Net version of this component based on the abstract design. More precisely, the exact extent of the generation depends on the degree of modeled information in the design model. If it only comprises interface definitions, we only can generate platform specific interfaces. However, we assume that a platform independent component specification additionally includes pre- and post conditions, behavior descriptions, and a state model of the component. Therefore, only a small amount of source code has to be generated manually.

The OMG however proposes Model-Driven Development by their Model Driven Architecture (MDA). The MDA [13] targets fully automated component generation. Therefore, it distinguishes two kinds of models: platform independent models (PIM) and platform specific models (PSM). We refer to a PSM if it is based on a particular form of technology such as Enterprise JavaBeans, JavaBeans, and Jini etc. A PSM is normally described in a modeling language such as UML and corresponds in a one-to-one fashion to an implementation of the system. For example, the OMG provides several UML profiles (PSM) that describe a platform such as Enterprise JavaBeans or CORBA in UML [15]. These profiles also define mappings in order to automatically generate source code. Contrary to these models, platform independent models (PIM) can be defined without reference to a platform, and therefore without describing a particular form of technology. Such kinds of models are usually specified using a modeling language without using platform specific elements, e.g. platform specific types. Figure 1 shows the core concept of Model-Driven Development. It distinguishes the different kinds of models as well as model transformations between them. In particular, it presents two mappings between a platform independent model and the Enterprise JavaBeans technology respectively the .Net Platform.

Unfortunately, the proposed advantages of Model-Driven Development can't yet realized with MDA, because automatic model transformations are required to gain an advantage towards traditional source code development. Currently, the MDA lacks a transformation language to perform necessary mappings [7]. Therefore, the OMG issued the Request for Proposal for Query/Views/Transformations (QVT) [14].

Another problem of today's software development concerns the lack of customization of particular model transformations. A transformation should be applied according to specific user requirements. For example, a platform indepen-

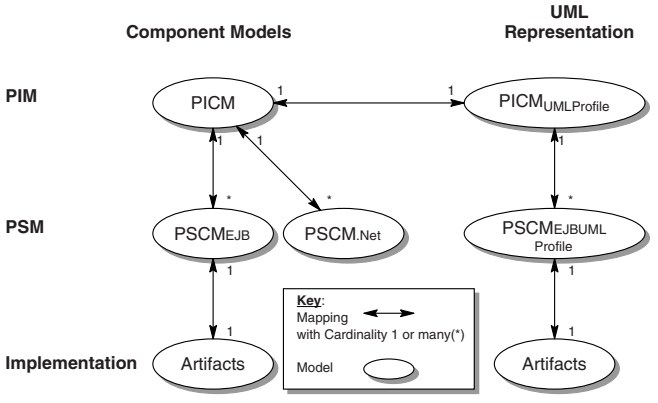

Fig. 1. Model-Based Transformation in MDA

dent component should be transformed into an EntityBean or a SessionBean according to particular requirements. Unfortunately, existing software development tools (such as Rose, ArgoUML etc.) do not support platform independent models. They often provide source code mappings only for one or a small number of technologies: These mappings are defined as one-to-one relationships between UML classes and source code classes. As a consequence, it is rarely possible to customize these mappings according to user requirements.

1.1 Objectives

We propose an ontology-based framework that provides customizable PIM-to-PSM mappings. The framework is based on a platform independent component specification that is constructed according to the component definition used in an Architecture Description Language (ADL). It is capable of handling mappings aimed for several platforms such as Enterprise JavaBeans, CORBA, .Net, COM etc. A user can add mappings for each platform of interest. Each mapping defines a relationship between the platform independent component model and a platform specific component model.

Furthermore, the framework allows defining several PIM-to-PSM mappings for each specific platform. Each mapping can be associated with a particular concept or feature that describes a situation when the mapping should be applied. Thus, a particular mapping is selected based on the requirements of a specific situation. For example, in this paper we start with one platform independent component specification and generate two different EJB models according to user requirements: One is optimized for data throughput between distributed components and one is optimized for local communication. A developer can specify certain properties such as quality of service attributes that should be taken into account in a particular situation. The framework chooses the appropriate mapping based on the specified properties and generates optimized platform specific EJB components.

The framework is based on Feature Models [5] as well as on Triple [16]. Feature models describe properties and alternatives for model transformations. They are used to specify mapping requirements. Triple is a deductive programming language, similar to F-Logic [10]. It is used to select the appropriate mapping and to perform the model transformation based on this mapping.

The remainder of the paper is organized as follows: Section 2 provides an overview of the framework with its component models and feature models. Section 3 explains model transformation including the realization with Triple.

1.2 Related Work

Several proposals for model transformation have been recently published in response to the OMG's RFP. These proposals can be classified regarding several categories such as how they define transformation rules or rule application strategies. Czarnecki and Helsen [6] provide a classification of model transformation approaches. According to this classification our approach, which is based on Triple is a declarative relational model-to-model approach. Other model transformation languages are based directly on UML. [18] for example defines an extension of the Object Constraint Language OCL using database manipulation operations of SQL. We use an existing language – Triple – to define mappings. Thus, the model transformation is "automatically" done by the inference engine. It allows declaring transformations in a very flexible and compact syntax, similar to F-Logic[1]. Additionally, the Triple concept of parameterized contexts allows a modularization of rule sets and enables the re-use of mappings by parameterized mapping specifications.

Additionally, our approach uses feature model instances, which describe mapping variants to parameterize mappings. Feature models are important in the context of product line engineering and domain analysis ([5, 4]). They are used to describe variants within a system family and to generate applications as instances of this system family from the application's specification.

Mostly, the generative approach is used on the implementation level. [2] defines the KobrA methodology for a component-based engineering with UML very similar to our approach. KobrA also contains the specification of variable parts of a system and feature models called decision models. But these concepts are only discussed in the context of product line engineering. We use them to support the general development process wherein alternative realizations must be chosen according to the requirements. Additionally, [2] discusses no explicitely specification of relationships between decisions and realizing system variants so that the transformation has to be done manually.

2 Overview of the Ontology-Based Framework

Our framework provides the frame for model transformation. It therefore has to model both platform independent components and platform specific com-

[1] As described in [7] a great advantage is the ability to express the model and instance level in an uniform way and to define multiple targets in a single rule.

ponents for each technology of interest. Furthermore, it includes properties as well as feature models to describe the customization of a model transformation. This section provides an overview of the architecture of our framework and the platform independent conceptual models. As an example for platform specific models we present the Enterprise JavaBeans model which is used in the given example.

2.1 General Architecture of the Framework

Figure 2 shows an overview of the framework's architecture. It is based on a knowledge base that provides reasoning and transformation capabilities. It mainly consists of models describing platform independent and platform specific components as well as of transformation rules. These rules transform a PIM model into a PSM model based on a feature model instance that describes user requirements. A property model allows marking model elements with feature values. The mappings are parameterized both on those marks and on the feature models. The figure does not show the behavioral model, as we do not map its specification at present.

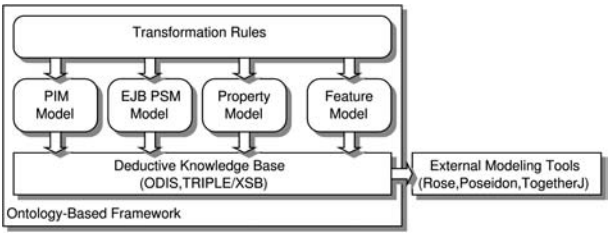

Fig. 2. The Architecture of our Framework

2.2 Platform Independent Models

The conceptual model handles component descriptions on an architectural level. It consists of three sub models representing each of the relevant areas of component descriptions as well as of a feature model to describe customized mappings. These models are shown in figure 3.

The structural model consists of elements that are found in most ADL: components, connectors, interfaces, as well as their relationships and subordinate elements such as operations. Moreover, it distinguishes instance and type elements, in order to describe both architectural styles and system configurations[2]. This model constitutes a type system, which can be specialized into technology specific types.

The behavioral model restricts components and connectors by means of pre- and post-conditions, as well as by protocols (order of method invocations). This

[2] Figure 3 shows only component types as we only use component types in our example.

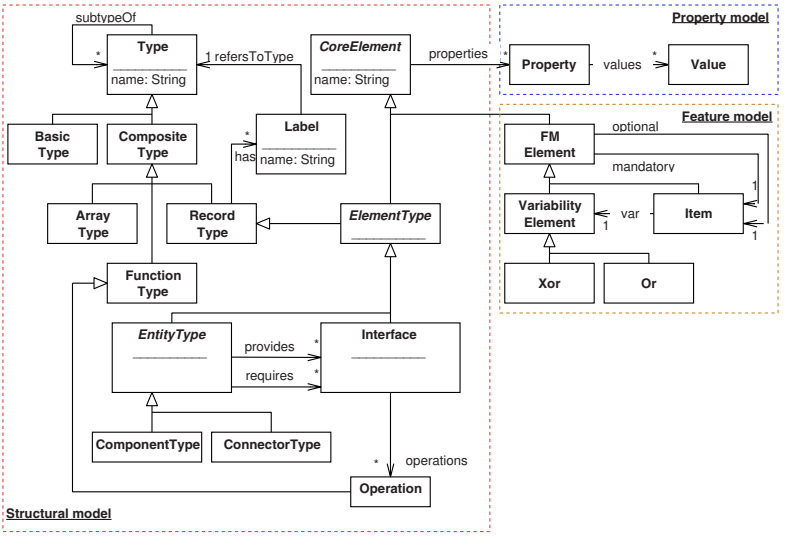

Fig. 3. Platform Independent Models of the Framework

allows verifying behavioral equivalence of components by using appropriate tools such as model checkers and theorem provers. However, this model is not shown in Figure 3, because we do not regard this information for transformation of EJB components.

The property model defines an ontology that describes architectural and technical properties of components. In particular, it provides a classification, which can be used to describe differences between components. Therefore, we integrated two well-known taxonomies from Allen/Shaw and Medvidovic/Mehta, which respectively describe architectural styles and communicational properties. In the context of model transformation, it is used to annotate components with their specific roles and to state user requirements on these elements. This is the basis for customized mappings, described in the next section.

Feature models describe alternatives to customize a mapping. They are graphically expressed in feature diagrams. Feature models play an important role in the area of domain analysis. Introduced in FODA [9] they serve as a description of the features of domain entities using and-or-trees enhanced with some useful elements to express variability. According to Deursen [19] they contain elements representing

- optional and mandatory features pointed to by a simple edge ending with an empty circle or a filled circle, respectively,
- alternative features pointed to by edges connected by an arc,
- non-exclusive features, also called *or-features*, pointed to by edges connected by a filled arc,
- and constraints over feature dependencies specified beside the feature diagram.

2.3 A Platform Specific Component Model for EJB

EJB components are described according to the EJB Profile for UML [8]. They mainly consist of a home interface, a remote interface and an implementation class. We use a slightly adapted form of the profile, because it doesn't support EJB2.x local interfaces. Figure 4 shows a simplified specification of the EJB model.

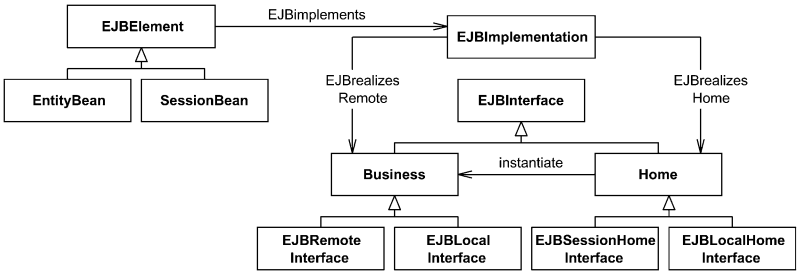

Fig. 4. The EJB Component Model

It is important to notice that the EJB specification and the Profile do not model connectors. Thus, we have to translate connectors from the platform independent level into EJB Components, if they include business logic. Otherwise, we do not need to model a connector at all, because, it is solely a relationship that we can configure in the EJB deployment descriptor.

2.4 Triple Realization

The framework is realized with Triple. Triple is based on F-Logic, which provides a logical foundation for object-oriented features. The design of Triple is influenced by the Resource Description Framework (RDF) [20], which is commonly known in the knowledge representation community. RDF is a general representation language for information structures. It provides basic constructors for the definition of concepts and their relations. Furthermore, it describes knowledge by Tuples (S, P, O): S is a subject, the entity to be described. P is a predicate that states the relation of interest, O stands for an Object, which is either a literal or another resource.

Despite the similarity to RDF, Triple describes tuples of the form (S, P, O, C). It introduces a 'context' as a new construct that allows specifying views of an object in different contexts. In our framework each core model is described within a separate context. This feature is extremely helpful because is divides up facts into chunks that can be used as separate units. Furthermore, it allows creating contexts on the fly by defining mapping rules.

Information specified in RDF can be validated against a schema definition, described by RDFS. The metamodels described above are schema definitions that correspond to RDFS. Instances (Facts) can be checked against these definitions.

An RDF statement can be formalized in Triple as follows

```
Subject[predicate -> object]@context.
```

Subject, predicate and object are normal parts of RDF whereas context refers to a particular tupel space as explained above. If the context is omitted the statement is valid in every possible context. The following example statement declares a *'context'* block:

```
@picm{
  CoreElement[
     subClass->ElementType[
                  subClass->EntityType[
                               subClass->ComponentType]]].
}
```

This indicates that all following statements are within the context *'picm'*. The statements describe a specialization relation between *'CoreElement'*, *'ElementType'*, *'EntityType'*, and *'ComponentType'*. Thus, Triple statements can be nested, which is extremely useful if we get to more complicated statements. Alternatively, we can state the example as

```
CoreElement[subClass->ElementType].
ElementType[subClass->EntityType].
EntityType[subClass->ComponentType].
```

We also can define instances for these schema definitions:

```
ComponentType[typeOf->Planner].
```

defines *'Planner'* as an instance of *'ComponentType'*.

As a final example, we define the platform specific component model for EJB expressed in Triple. Figure 5 shows the corresponding Triple statements.

The main focus of this paper concerns model transformation. Triple supports these transformations by mapping constructs. We show two examples: The first example describes a parameterized mapping by defining a parameterized context. It describes a kind of 'copy' operation that replicates all facts within the context A into the context B.

```
FORALL A,B @ picm(A,B) {
    FORALL S,P,O
        S[P->O]@A --> S[P->O]@B
}
```

The second example is a mapping rule within the actual context. It is not parameterized and generates for each *'Interface'* an equal named *'ComponentType'*.

```
FORALL Z Interface[typeOf->Z] --> ComponentType[typeOf->Z]
```

```
Class [
    typeOf -> { EJBElement, EJBImplementation, EJBEntityBean, SessionBean,
                Home, Business, EJBSessionHomeInterface, EJBLocalHomeInterface,
                EJBRemoteInterface, EJBLocalInterface } ].

EJBElement [
    subClass -> EntityBean;
    subClass -> SessionBean ].
EJBInterface [
    subClass -> Home [
       subClass -> EJBSessionHomeInterface;
       subClass -> EJBLocalHomeInterface ];
    subClass -> Business [
       subClass -> EJBRemoteInterface;
       subClass -> EJBLocalInterface ]
    ].

Property [ typeOf -> { EJBRealizesHome, EJBRealizesRemote, EJBImplements,
                       instantiate } ].

EJBRealizesHome   [ domain -> EJBImplementation; range -> Home ].
EJBRealizesRemote [ domain -> EJBImplementation; range -> Business ].
EJBimplements     [ domain -> EJBElement; range -> EJBImplementation ].
instantiate [ domain -> Home; range -> Business ].
```

Fig. 5. Platform Specific Component Model for EJB expressed in Triple

3 PIM-PSM Model Transformation Explained

In this section we present our approach on PIM-to-PSM transformations. Thereby we also regard variants of transformations using feature modelling to determine an appropriate mapping. After describing our approach to model transformation in general, we explain it using an example.

3.1 Customized Model Transformation

In difference to common practice we do not map a PIM directly into source code. Instead, it is mapped into a PSM expressed in UML. The appropriate mapping is chosen according to selected requirements. A PIM-PSM transformation is based on three elements:

- the platform independent model that should be mapped. In our context this will be a PIM represented as an instance of the metamodel described in the last section. Especially, the elements like components are annotated with further properties that can be used to customize a model transformation to the specific situation.
- a feature model instance describing requirements that should be considered in the transformation. It is used to choose an appropriate mapping. So, the feature modelling allows the customization of a model transformation.
- mappings that define rules for possible transformations. Mappings formally specify design knowledge that is used when realizing a system with a specific middleware technology. They enable the automatization of the transformation process.

A PIM-PSM transformation is done as following: At first, the developer designs a system modelling it independent from platform technologies. In a second step, he specifies his requirements on the PIM-PSM transformation by choosing features from the feature model describing possible variants of available transformation rules. The chosen features are called *feature model instances* (FI). In contrast to a feature model it is an instance model that does not contain any variants. In the last step, a development tool transforms the PIM according to the specified requirements.

Formally, the basis for the customized model transformation is a set of transformation rules. Each rule takes the PIM and the feature model instance (FI) as input arguments and specifies the PSM appropriate for the given situation. In our framework all participating models – PIM, feature model, and PSM – are represented as instances of corresponding metamodels that can be translated into the TRIPLE-based format. On this basis a mapping can be defined as a TRIPLE-mapping with parameterized contexts:

```
FORALL PIM, FI @ pim2psmMapping(PIM, FI) {

  // Transformation rule 1
  FORALL <...necessary variables...>
    <...constraint...> @ PIM,  <...constraint...> @ FI
    -->
      <...PSM elements...>

  // Transformation rule 2
  ...
}
```

3.2 Example

Our running example comes from federated information systems. We describe two possible PIM-to-PSM transformations from the platform independent model to EJB specific models regarding specific requirements on distribution and optimization.

3.2.1 Platform Independent Model (PIM). The PIM of our example consists of two components that are part of a mediator (see figure 6). A mediator is a kind of middleware that performs queries against heterogeneous distributed data sources ([21]). If a client queries a mediator calling the `execute` operation, the mediator first calculates which data sources are capable to answer the query or part of it (Planner component). Then, it queries these sources, integrates the answers and delivers the result back to the client.

The Planner calculates its plans based on specified interfaces of the data sources. These interface descriptions are called query capabilities (QC). A query capability, shown on the right side of the figure, consists of parameters that a data source can process as well as of result attributes returned by the data source. In figure 6, the QCManager component stores the query capabilities (QC)

of managed data sources. The Planner uses QC, obtained by the QCManager, to decide which data sources have to be queried. These query plans are provided to the execution component of a mediator.

The figure also shows some annotations that are used for the transformation later on. These annotations are properties describing a component's role regarding its interoperation with another component. For example, the Planner is a client using the interface of the QCManager.

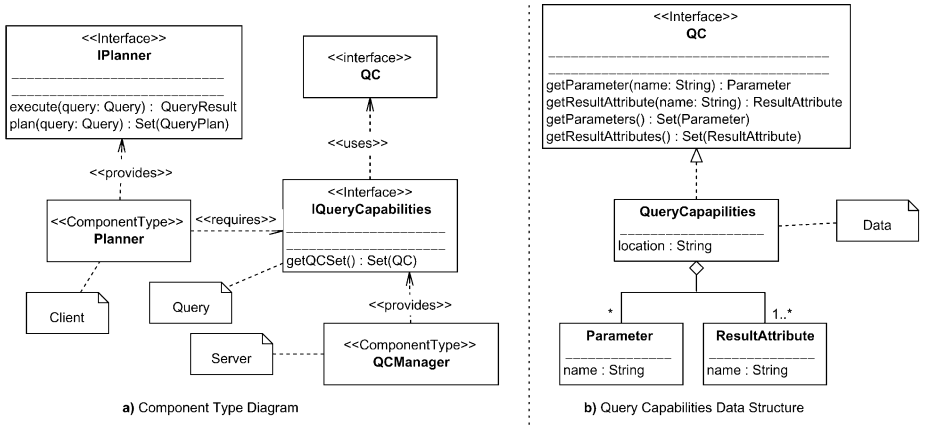

Fig. 6. Example – PIM Component Type View

Specification with TRIPLE. The formal specification of the PIM within our framework uses the TRIPLE representation of the UML metamodel and the PIM metamodel defined in the last chapter. It defines the elements of figure 6 as instances of the metamodels.

3.2.2 Variants of the PIM-PSM Transformation. The PIM of the QCManager is used as a starting point to exemplify two PSM transformations for the Enterprise JavaBeans platform. We present two customized transformations according to specific requirements (criteria) for our example: We regard physical distribution of mediator components and performance of communication measured in the number of procedure calls between components and the amount of transmitted data. In general, several factors influence performance. In distributed systems, performance can be improved by minimizing distributed transactions, remote procedure calls, the amount of transmitted data etc. Figure 8 shows the feature diagram related to our example.

The developer chooses the features from the feature model that should be considered in a specific transformation. We will examine a local PSM transformation optimized for the amount of transmitted data, as well as a distributed PSM transformation optimized for the amount of remote procedure calls. Both are realized with the EJB platform.

```
...
picm := "http://cis.cs.tu-berlin.de/modeltrafo#picm".
uml := ... .

//// Schema:

Class [ typeOf -> { Property, CoreElement, EntityType,
                    ElementType, ConnectorType, ComponentType } ].

CoreElement [
    subClass -> ElementType [
      subClass -> EntityType [
        subClass -> ConnectorType;
        subClass -> ComponentType ] ] ].

rdf:Property [ typeOf -> {properties, requires, provides,operations} ].

properties  [ domain -> CoreElement; domain -> Operation; range -> Property ].
requires    [ domain -> EntityType; range -> Interface ].
provides    [ domain -> EntityType; range -> Interface ].
operations  [ domain -> Interface;   range -> Operation ].
annotation  [ domain -> CoreElement; range -> Literal ].
...

//// Instances:

Interface     [ typeOf -> { IPlanner, IQueryCapabilities, QC } ].
ComponentType [ typeOf -> { Planner, QCManager } ].
uml:Class     [ typeOf -> { IQueryCapabilities, Parameter, ResultAttribute } ].

Planner [
   provides -> IPlanner; requires -> IQueryCapabilities; annotation -> "Client" ].
QCManager [
   provides -> IQueryCapabilities; annotation -> "Server" ].
IQueryCapabilities [
   operations -> op1 [
      name -> "getQCSet"; result -> Set(QC); annotation -> "Query" ] ].
QueryCapabilities [
   realizes -> QC; aggregates -> { Parameter, ResultAttribute };
   annotation -> "Data" ].
```

Fig. 7. Example – Triple Specification of the Platform Independent Model

Specification with TRIPLE. The features chosen by the developer determine which transformation is appropriate for the specific application. To manage transformation variants in our framework we use a TRIPLE representation again. Figure 9 shows an instance of the feature metamodel namely the feature model (FM) shown in figure 8, which is used for the mapping selection. A feature model instance (FI) is a feature model without any variants. It represents the features chosen by the developer and is the input for the PIM-PSM transformation. For example the FI at the end of figure 9 requests a transformation based on remote distribution and optimized procedure calls (distributed configuration).

3.2.3 PIM-PSM Transformation. Starting from the PIM of a mediator and the feature model for EJB architectures we show two possible transformations to a EJB-based PSM. In our example the transformation is based on patterns [11, 1] that were developed to optimize EJB communication and performance. We will discuss a local and a distributed configuration of the mediator components.

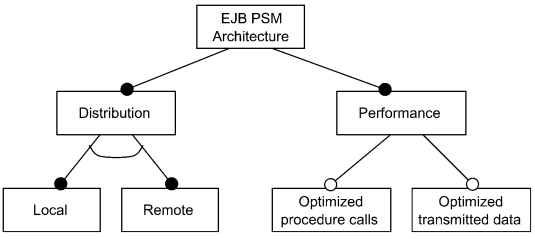

Fig. 8. Feature diagram for EJB-based architectures

```
 ...
 fm := "http://cis.cs.tu-berlin.de/modeltrafo#featureModel".

//// Instance of the feature metamodel as input for mapping selection:

 Item [ typeOf -> { EJB_PSM_Architecture, Distribution, Performance,
                    Local, Remote, Optimized_procedure_calls, Optimized_transmitted_data} ].
 Xor  [ typeOf -> xor1 ].

 EJB_PSM_Architecture [
    mandatory -> Distribution [
        var -> xor1 [
            mandatory -> {Local, Remote} ] ];
    mandatory -> Performance [
        optional -> {Optimized_procedure_call, Optimized_transmitted_data} ] ].

//// Instance of the feature model as a result of a mapping selection:

 EJB_PSM_Architecture [
    mandatory -> Distribution [
        mandatory -> Remote ];
    mandatory -> Performance [
        mandatory -> Optimized_procedure_call ] ].
```

Fig. 9. Example – Feature Models in Triple

Example: Local Configuration. This transformation is done according to the features that were chosen by the developer: both components are co-located and optimized for the amount of transmitted data. Using the patterns in [11, 1] the transformation results in the PSCM shown in figure 10. The Planner component as a client in this example is mapped into a Session Bean, because it is used as a business logic component, e.g. it provides computations. The QCManager, which includes the QC data structure (determined by the relationship to an element with a Data annotation), is mapped into three Entity Beans, as it presents persistent data. We don't need an extra QCManager Bean as this component would introduce another layer of indirection. Instead, we directly access the persistence layer. This leads to optimized data transfer, as we don't have to collect all data of the persistence layer and send it to the Planner component. Instead, data is returned in form of a set of references to locally available QC Entity Beans. The operation to get all QC is renamed to `findQCSet` as described in the profile. If the Planner needs parameters or return attributes, additional calls are performed to obtain the queried entities.

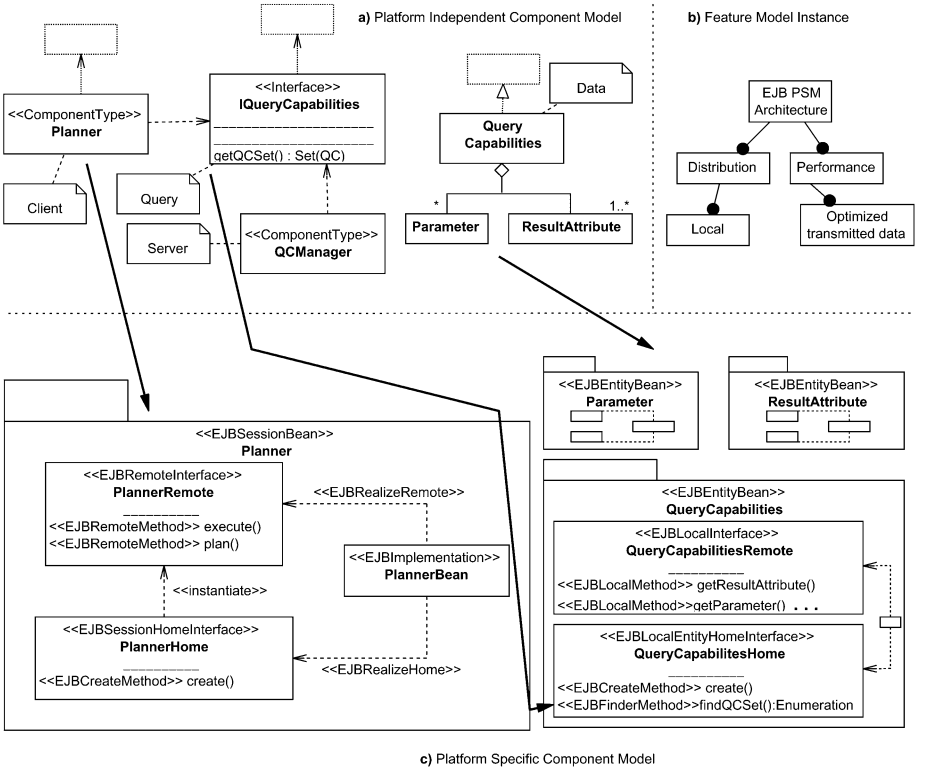

Fig. 10. Example – PIM-PSM Transformation for a Local Configuration

In order to save space, only the Planner component in figure 10 shows all parts of an EJB according to the profile. Otherwise, we only show interesting parts of a bean and represent other elements as small boxes.

Example: Distributed Configuration. The distributed transformation optimizes remote procedure calls between distributed Planner components and a single QCManager component. Again, the transformation is based on the chosen features from the feature model. Regarding the EJB platform, several patterns for performance optimization were developed. We will use the Data Transfer Object pattern (DTO)[11, 1] and the Data Transfer Object Factory pattern (DTOF)[11] for the PIM-to-PSM transformation in this example.

Figure 11 shows the mapping result. The QCManager component is mapped into a stateless session bean following the DTOF pattern that provides a facade to the persistence layer consisting of three Entity Beans. The QCManager locally assembles Data Transfer Objects for each query by calling the Entity Beans. In difference to the local mapping these objects are copies of persistent data. Thus, a query of the Planner component results in a single remote procedure call.

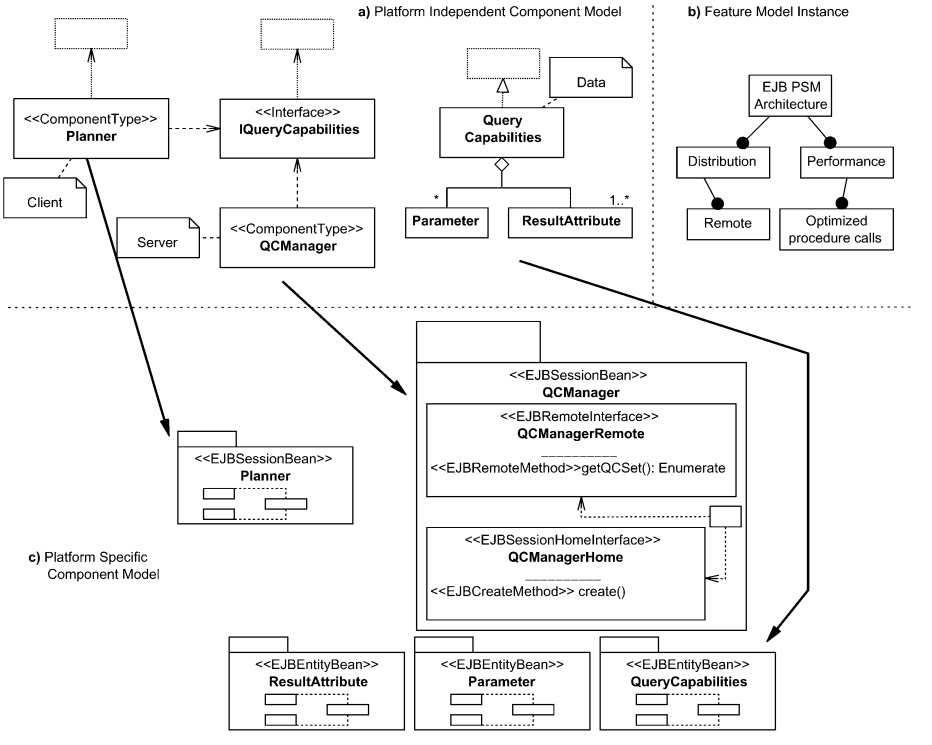

Fig. 11. Example – PIM-PSM Transformation for a Distributed Configuration

Specification with TRIPLE. The specification of the mappings for our transformation described before consists of two parts: the first one defines the general mapping from PIM elements to EJB, the second one defines the mapping depending on the possible features.

Figure 12 shows the general mapping definition from PIM models to session or entity beans. The arguments of the bean mapping are the resource variables `X`, `PIM`, and `Kind`. `PIM` is the context of the mapping source, `X` is the element from that source which shall be mapped, and `Kind` states whether a session or an entity bean shall be the result. The mapping definition consists of one rule which expresses the following: If a element with an appropriate `provides`/`realizes`-structure can be derived within the source context (left hand side of the rule) then a bean and the remote/home interfaces corresponding to the Enterprise EJB standard will be generated (right hand side of the rule). The instantiation of the `type`-variables in the target structure depends on `Kind`. The target operation `TF` is the result of a special predicate `convertst` called with the source operation `F`. This predicate adds the stereotypes defined by the EJB profile.

Figure 13 shows the specific PIM-to-PSM-mapping depending on possible feature model instances. Within the utility mapping `util` two specific FIs are considered: the distributed and the local configuration variant. The mapping

```
... // namespaces and abbreviations

FORALL X,PIM,Kind  @ beanMapping(X,PIM,Kind) { // general mapping to a bean

 FORALL B,R,H,Y,F,TF,P

    ( X [ P -> Y [ operations -> F ] ]@ PIM,
         convertst(F,TF),(P = provides OR P = realizes)),
    cond( Kind = "Entity",
          B = EJBEntityBean, R = EJBLocalInterface, H = EJBLocalHomeInterface ),
    cond( Kind = "Session",
          B = SessionBean, R = EJBRemoteInterface, H = EJBSessionHomeInterface ),
    -->
        X [ type -> B;
            implements -> X||'Bean' [
                type -> EJBImplementation;
                EJBRealizesRemote -> X||'Remote' [
                    type -> R;
                    operations -> TF ];
                EJBRealizesHome ->   X||'Home' [
                    type -> H;
                    instantiate -> X||'Remote' ] ]
}
```

Fig. 12. Example – General Bean Mapping

`pim2psmMapping` has two parameters: the context of the pim source `PIM` and the context of the feature instance `FI`. The body of the mapping contains the mapping rules according to the variants of transformation explained above:

- Any client element is mapped to a session bean.
- A server will become a session bean if the variant of remote distribution is chosen.
- All aggregated elements of a data element are mapped to entity beans.
- A data element x also become an entity bean. If the variant of local data is chosen then an EJB conform conversion of any `Query`-annotated operation which uses the interface of x is placed within the home interface of the generated bean.

4 Conclusions

In this paper, we presented a model transformation framework that is able to express and process customized PIM-to-PSM mappings. Contrary to existing tools, the framework handles several component technologies as it is based on the platform independent component model (PICM). PICM allows describing components on different levels of abstraction. Thus, it provides the foundation for PIM-to-PSM mappings, which is a feature that is not provided by existing modeling tools. As PICM is based on an architecture description language (ADL), it facilitates easy integration of new component-based technologies. However, instead of using an existing ADL, PICM is based on Triple/RDF. Its main purpose is to perform explicit reasoning on the selection of component mappings and to allow declarative rule specifications between model representations.

```
... // namespaces and abbreviations

FORALL FI  @ fm:util(FI) { // utility predicates for feature model instances

   fm:Distribution [ fm:mandatory -> fm:Remote ] @ FI,
   fm:Performance [ fm:mandatory -> fm:Optimized_procedure_call ] @ FI  -->  remoteCall.

   fm:Distribution [ fm:mandatory -> fm:Local ] @ FI,
   fm:Performance [ fm:mandatory -> fm:Optimized_transmitted_data ] @ FI  -->  localData.
}

FORALL PIM, FI  @ pim2psmMapping(PIM, FI) { // pim-to-psm mapping

   FORALL X,T
       X [ annotation -> "Client" ] @ PIM,  T @ beanMapping(X,PIM,"Session")
       --> T.

   FORALL X,T
       X [ annotation -> "Server" ] @ PIM,  T @ beanMapping(X,PIM,"Session"),
       remoteCall @ fm:util(FI)
       --> T.

   FORALL T,X,Y

       X [ annotation -> "Data"; aggregates -> Y ] @ PIM,  T @ beanMapping(Y,PIM,"Entity")
       --> T.

   FORALL X,Ifc,T,F,TF,U,H

       X [ annotation -> "Data"; realizes -> Ifc ] @ PIM,  T @ beanMapping(X,PIM,"Entity")
       -->
           T,
           ( localData @ fm:util(FI),
             F [ annotation -> "Query" ] @ PIM, use(F,Ifc), convert(F,TF),
             U [ EJBRealizesHome -> H ] @ beanMapping(X,PIM,"Entity")
             -->
                 H [ operations -> TF ] ).
}
```

Fig. 13. Example – Specific PIM-PSM Transformation Rules

Triple/RDF is a model representation and transformation language. It is suitable to represent and interrelate terminological structures such as feature models as well as conceptual models like UML class diagrams. It allows describing both the model and the instances in a uniform and simple syntax.

The second contribution of the paper is the combination of mappings with feature models. A feature model facilitates the selection of particular mappings depending on certain user requirements. This gives our framework the flexibility to choose appropriate model transformations in a particular situation. To the best of our knowledge there is no existing modeling tool for component-based systems that provide this flexibility.

Currently, the model transformation tasks are realized by services of the Ontology-Based Domain Repository ODIS [3]. Future developments will integrate the presented model transformation framework as a service in the Evolution and Validation Environment (EVE) [17]. EVE allows executing arbitrary services on UML models that were extracted from UML modeling tools. EVE is based on a MOF repository and uses XMI to get models from these tools. At present it supports ArgoUML and Rational Rose. As a consequence, we will

be able to directly transform UML models, which were created with a modeling tool, into representations in several component technologies either as UML PSM or as source code. This will be a further step in the direction of a model-driven software engineering.

References

1. Alur, D., Crupi, J., and Malks, D. *Core J2EE Patterns: Best Practices and Design Strategies.* Prentice Hall / Sun Microsystems Press, 2001.
2. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., and Zettel, J. *Component-based Product Line Engineering with UML.* Component Software Series. Addison-Weseley, 2002.
3. Billig, A. ODIS - Ein Domänenrepository auf der Basis von Semantic Web Technologien. In *Tagungsband der Berliner XML Tage* (2003), XML-Clearinghouse. english version: http://www.isst.fhg.de/~abillig/Odis/xsw2003.
4. Clements, P., and Northrop, L. *Software Product Lines: Practices and Patterns.* Kluwer, 2001.
5. Czarnecki, K., and Eisenecker, U. *Generative Programming - Methods, Tools, and Applications.* Addison-Wesley, 2000.
6. Czarnecki, K., and Helsen, S. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture* (Anaheim, October 2003).
7. Gerber, A., Lawley, M., Raymond, K., Steel, J., and Wood, A. Transformation: The missing link of MDA. *Lecture Notes in Computer Science 2505* (2002).
8. Greenfield, J. UML Profile For EJB. Tech. rep., Rational Software Corporation, May 2001. `http://www.jcp.org/jsr/detail/26.jsp`, Java Community Process (JCP).
9. Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, nov 1990.
10. Kifer, M., Lausen, G., and Wu, J. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM 42* (Juli 1995), 741–843.
11. Marinescu, F. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms.* John Wiley & Sons, Inc., 2002.
12. Mellor, S. J., Clark, A. N., and Futagami, T. Model-driven development. *IEEE Software 20*, 5 (2003), 14–18.
13. Miller, J., and Mukerji, J. Model Driven Architecture(MDA). Tech. Rep. ormsc/2001-07-01, Object Management Group(OMG), Architecture Board ORMSC, July 2001.
14. Object Management Group (OMG). *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*, April 2002. `http://www.omg.org/cgi-bin/apps/do_doc?ad/2002-04-10.pdf`.
15. OMG. *UML Profile for CORBA Specification V1.0*, 2000.
16. Sintek, M., and Decker, S. TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web. In *Proceedings of International Semantic Web Conference ISWC 2002* (2002), Lecture Notes in Computer Science, Bd. 2342, Springer.

17. SÜSS, J. G., LEICHER, A., WEBER, H., AND KUTSCHE, R.-D. Model-centric engineering with the evolution and validation en vironment. In *UML 2003 - The Unified Modeling Language: Modeling Lan guages and Applications, 6th International Conference, San Francisco, CA, USA* (2003), P. Stevens, J. Whittle, and G. Booch, Eds., vol. 2863 of *LNCS*, Springer, pp. 31 – 43.
18. SÜSS, J., LEICHER, A., AND BUSSE, S. OCLPrime - Environment and Language for Model Query, Views, and Transformations. In *OCL 2.0 - Industry standard or scientific playground?, Workshop on the 6th Int. Conf. UML 2003* (2003).
19. VAN DEURSEN, A., AND KLINT, P. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology* (2001).
20. W3C. Resource Description Framework (RDF) Model and Syntax Specification. URL: http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.
21. WIEDERHOLD, G. Mediators in the Architecture of Future Information Systems. In *Readings in Agents*, M. N. Huhns and M. P. Singh, Eds. Morgan Kaufmann, San Francisco, CA, USA, 1997, pp. 185 – 196.