Model Bus: Towards the Interoperability of Modelling Tools

Xavier Blanc, Marie-Pierre Gervais, and Prawee Sriplakich

Laboratoire d'Informatique de Paris 6 (LIP6), University Paris VI,
8, rue du Capitaine Scott, 75015 Paris, France
{Xavier.Blanc, Marie-Pierre.Gervais,
Prawee.Sriplakich}@lip6.fr

Abstract. MDA software development requires the interoperability of a wide range of modelling services (operations taking models as inputs and outputs), such as model edition, model transformation, and code generation. In particular, software development life cycle requires the interoperability of different modelling services. In particular, this interoperability concerns how to "connect" services (how to send an output model produced by one service as an input to another service). Today, the notion of modelling services is not yet well defined. Moreover, CASE tools, which implements different services, have heterogeneous interfaces. For this reason, the service connection is costly and cannot be automated. Currently, there are few works addressing this problem. Therefore, we propose an architecture and a prototype enabling the services of different tools to be connected.

1 Introduction

According to Model Driven Architecture (MDA), models are treated as first-class elements in software development [21]. MDA application requires a wide range of modelling services such as model edition [15], model storage [15], model manipulation [22][14], code generation [9] and model transformation [4][7][8]. We can mention also model execution and model validation as some work are now ongoing at the OMG (execution semantics defined in UML 2.0 [25], Object Constraint Language 2.0 [24]). For precision, we define the term ***modeling service*** as an operation having models as inputs and outputs. Hence, the users of modeling services are software developers that want to apply different modeling services to their models in order to, for example, analyze, design and implement software.

Several CASE tools, implemented by different vendors (or developer groups), offer various modelling services. For example, NetBeans Metadata Repository [18], ModFact [17], Eclipse Modeling Framework (EMF) [10], and Univers@lis [3] propose model storage and model manipulation. Rational Rose [30], Objecteering [19], EclipseUML [11], Poseidon [29] and ArgoUML [2] propose UML model edition and code generation. ArcStyler [1], MIA [16], and UMT-QVT [32] propose model transformation. Although these tools cover a lot of modelling services, some services, such as UML model execution [31], OCL constraint verification [13], deep model copy [28], are not commonly supported by commercial tools.

U. Aßmann, M. Aksit, and A. Rensink (Eds.): MDAFA 2003/2004, LNCS 3599, pp. 17 – 32, 2005.

According to the MDA vision, software development life cycle requires the interoperability of tools. In particular, the ***connection*** between the services of different tools must be enabled. This problem concerns how to send an output model produced by one service as an input to another service (which may be offered by a different tool). For example, connecting a model storage service to a model transformation service will enable the model transformation service to retrieve its input or to store its output in the model storage service.

Connecting modelling services is a difficult problem. We identified two main concerns regarding this problem: ***functional connection*** and ***concrete connection.*** Functional connection ensures that the service inputs and outputs have compatible types so that the services can exchange data. It particularly concerns the type compatibility of models. Concrete connection ensures that modelling service connections can be realized at run-time. In particular, the connected services must agree in a model representation form and in a mechanism for exchanging models.

Today service connection cannot be done in an automated way. As a result, users must spend a lot of technical efforts to realize the connection. Neither functional connection nor concrete connection can be automated. The functional connection (i.e. type compatibility checking) is not automated because today tools are only documented informally in natural languages (in manuals), so the information about input/output types may be insufficiently precise and can not be exploited.

Moreover, each tool has its own model representations for encoding its services' inputs and outputs. A model representation can be either a textual form (e.g. XML Metadata Interchange (XML) [27], Human-Usable Textual Notation (HUTN) [20]) or an object form (e.g. Java Metadata Interface (JMI) [14], EMF Repository [10]). Also, each tool provides different interfaces. Some tools provide graphical user interfaces [30][19], some are executed via command lines [17] and others propose APIs for calling services [10]. To connect services of different tools, a dedicated conversion is required for each pair of tools. This effort is costly and can only be done manually. For this reason, concrete connection is not automated.

Despite the needs for connecting modelling services, there are currently few works concerning this problem. The Eclipse platform has been developed for connecting tools. But Eclipse does not take into account the particularity of modelling domain. Although the EMF offers the integration of modelling tools into Eclipse, it does not address at all the functional connection problem and the way tool connections are realized is limited to the use of the EMF's Java API.

We propose here the Model Bus architecture for addressing the functional connection and the concrete connection problems. Model Bus is mainly based on middleware technologies such as CORBA and Web Services but it adds new features for dealing with modelling aspects. Model Bus enables the automation of modelling service connections. We have implemented a prototype of Model Bus on the Eclipse platform and we have connected several modelling services proposed by the ModFact tools.

This paper is organized as follows. Section 2 discusses the difficulties of modelling service connection. Section 3 presents Model Bus architecture and explains how Model Bus can automate modelling service connection. In section 4, we show how to use Model Bus in an example scenario. Section 5 validates our concepts by presenting our prototype. Section 6 compares our approach with others. The last section concludes our work and presents research perspectives.

2 Service Connection Problem

First of all, let us illustrate the notion of service connection through an example (c.f. figure below). In this example, a user (software developer) wants to perform a UML to Enterprise Java Bean (EJB) transformation. To do this, he does the following scenario: First he will find a UML model in a *UML Repository* service. This service requires a model name as an input and returns a UML model as an output. This output is connected to the input of a *Transformation* service for transforming the UML model to an EJB model. The output of the transformation service (i.e. EJB model) will be connected to the input of a *Code Generation* service for generating an EJB application (i.e. code).

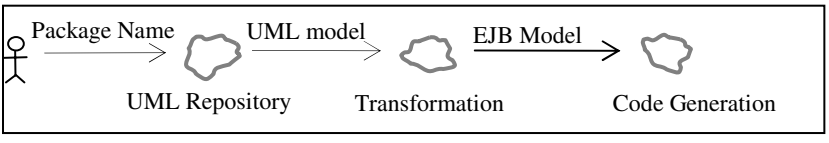

Fig. 1. A service connection example: A software developer wants to use three modelling services (UML Repository, Transformation, Code Generation) provided by three different tools conjointly.

This kind of scenario seems to be common in MDA software development. However, we will show you that there are significant difficulties in service connection.

2.1 Functional Connection: Checking Type Compatibility

To ensure that the service connection is possible, the type compatibility between an output of a service and an input of another service must be checked. The previous example requires the following checking: UML Repository's output and Transformation's input, Transformation's output and Code Generation's input.

The type compatibility is a well-known problem; however it has not been addressed in the modelling domain. Unlike classical data type, the model type compatibility is not a trivial problem because nowadays there is no well-known, precise definition of model types. Finding such a definition is also difficult because there are uncountable kinds of models (e.g. UML models, SPEM models, CWM models …). We will identify that model types have several characteristics. Then we will illustrate why these characteristics are important to the model type compatibility problem.

Model type characteristics and example of type checking rules

Metaclasses: It is a common practice to use a metamodel to define *model types* - input and output types of a service. In other words, the service's inputs or outputs can be anything conforming to a metamodel. However, currently there is no precise definition of metamodels: Is it a set of metaclasses (MOF classes) or a set of metapackage (MOF packages)? We propose that model types should be defined in terms of metaclasses rather than metapackages. This is because most services are

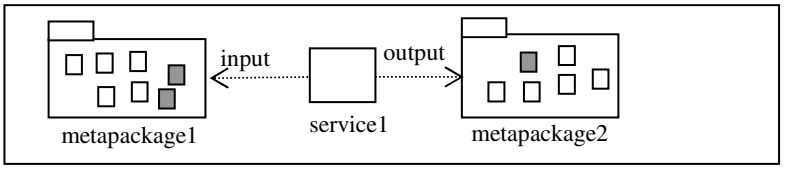

Fig. 2. Roles of metaclasses in the model type definition: The gay rectangles represent the metaclasses whose instances are inputs and outputs of a modelling service.

capable of processing instances of some metaclasses. A metapackage may contain metaclasses whose instances are not acceptable by services. The figure below shows how model types are defined. The grey rectangles in metapackage1 represent the metaclasses whose instances can be processed by the service and the metaclass in metapackage2 whose instances are produced by the service.

"Any" vs. specific model types: A model type is said to be specific if the corresponding models can contain only instances of some specific metaclasses. On the contrary, for the "any" model type, the corresponding models can contain instances of any metaclasses. The "any" model type is necessary because there are several services that operate on this type. For instance, the MOF QVT proposal [23] defines *generic* transformation that can be applied to any kinds of models. The input and output of this transformation is "any" model type.

Model granularity: A model can contain either a single instance of a metaclass (e.g. a UML package, a UML class) or a collection of instances (UML packages, UML classes). Therefore, the model type definition must specify the allowed number of instances, for example, "a single instance", "no more than two instances", or "any number of instances". Moreover, for collection-granularity model types, the order of instances in the collection may have meanings. Therefore, the type definition should specify whether the instances are required to be ordered.

The characteristics presented above are required for checking the type compatibility. We present some checking rules that use those characteristics. Then we will show that those rules cannot be verified in the UML-to-EJB example.

Metaclasses: An output model type (T1) is compatible to an input model type (T2) if all the T1's metaclasses are included in the set of T2's metaclasses.
As regards the example, UML Repository does not specify the return type. It may return either a UML package (instance of metaclass *Package*) or classes contained in the package (instances of metaclass *Class)*, or other things (UseCase, Sequence Diagramme etc). Consequently, we cannot check whether its return type is compatible to Transformation's input.

"Any" vs. specific model types: All specific model outputs are compatible to an "any" model input. On the other hand, an "any" model output does not always compatible to a specific model input depending on the actual type (at runtime) of that "any" model output. Therefore, the metaclass checking is necessary at runtime.

As regards the example, it is not specified whether Transformation service is generic or specific to particular kinds of models. As consequent, we cannot know whether the type checking must be performed statically or at runtime.

Model granularity: The instance number range of the output model type must be included in the range of input model type. For example, "only single instance" is included in "from zero to two instances".

As regards the example, Transformation service does not specify how many instances (of metaclasses) the result model will contain. If the result model contained multiple instances while Code Generation service can handle only one instance, the service connection would cause errors.

We conclude that the type compatibility verification requires precise service description, especially the input/output types of services. Moreover, if this description were specified in a well-defined format, the automation of the checking rules would be feasible. However, this is not the case in current practice because such description is usually written in natural languages (i.e. in tool manuals). For this reason, the functional connection is an unsolved problem.

2.2 Concrete Connection: Executing Connected Services

As previously explained, to execute connected services, the services must agree in a model representation form and in a mechanism for exchanging models. However, tools providing services are heterogeneous. Therefore, two tools can hardly exchange models. We identify two kinds of tool heterogeneity: model representation forms and interface styles (i.e. the way services receive inputs and return outputs).

Model representation forms: Tools have their own model representation forms. On one hand, some tools use models represented in textual formats. For example, Poseidon and ArgoUML store models in the XMI format. On the other hand, some other tools require models in object forms. For example, model edition services in EclipseUML operate on model objects in the EMF repository.

Interface styles: The way services receive inputs and return outputs vary from a tool to another. For example, Rational Rose offers to users a graphical user interface (GUI) for applying code generation services on a UML model. ModFact provides a command line interface for applying a DTD generation on a MOF model. EMF provides an API for using the model manipulation service on an EMF repository. Moreover, tools that support multi-users can provide remote access. For instance, ModFact repository allows the model manipulation service to be accessible through the CORBA RPC. We can also anticipate tools offering Web Service access to their services.

Both kinds of heterogeneity cause difficulties in concrete connection. If services use different model representation forms, an output of a service can not be understood by one another. Furthermore, some interface styles, such as command lines or GUIs, do not support automatic interaction. To connect the services offering such interfaces, users must manually transfer a model from one service to another. In this case, automating service connection is not possible.

Although this heterogeneity problem is a well-known problem and several solutions have already been proposed (e.g. CORBA, Web Service), none those solutions addresses the particularity of modelling domain. They do not define model representation forms and interface styles that are appropriate to modelling services.

3 Model Bus

3.1 Describing Functional Connection

Our design principle is to provide well-formed service description. In particular, service inputs and outputs must be precisely defined in order that the service connection can be checked. The next figure contrasts the current practice and our solution. In the current practice, as we have mentioned that today there is no well-known, precise definition of model types, the view of modelling services is unclear. Our approach proposes a uniform view where services are similar to software components having precise input and output definitions.

We propose a metamodel, called *Functional Description* (c.f. the next figure). This metamodel describes the signatures of modelling services in an abstract way. Modelling services are similar to classical operations that have input and output parameters. However they have a new important feature: their input and output types can be models.

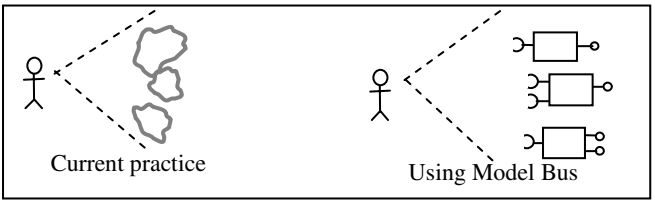

Fig. 3. Modelling services viewed as software components: Our goal is to provide a precise definition of modelling services. This definition must enable users to identify compatible services that can be connected.

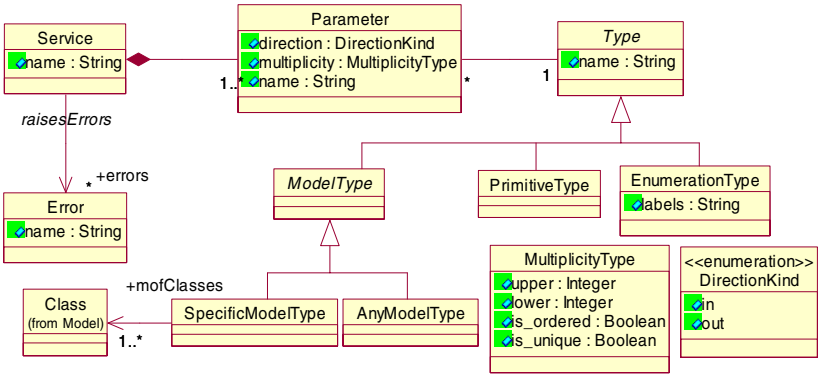

Fig. 4. Functional Description metamodel: We can describe each modelling service by creating an instance of this metamodel.

The Functional Description metamodel addresses the problem of the type compatibility verification by allowing services to be sufficiently described. The model characteristics presented in 2.1 can be precisely specified as follows.

Metaclasses: The metaclass *SpecificModelType* references MOF metaclasses whose instances can be contained in input and output parameters. For example, in the description of a service requiring a UML use case, the SpecificModelType will point to the metaclass *UseCase* in the UML metamodel. This approach is unambiguous since *SpecificModelType* allows users to obtain, for example, the complete definition of UML use cases (in a particular version of the UML metamodel).

"Any" vs. specific model types: The “any” and specific model types can be distinguished using metaclasses *SpecificModelType* and *AnyModelType*. SpecificModelType points to the metaclasses whose instances are expected while AnyModelType indicates that the parameter can contain instances of any metaclasses of any metamodels.

Model granularity: *MultiplicityType* allows model granularity to be specified using the *upper* and *lower* attributes. For example, [2..2] (i.e. lower=2, upper=2) and [1..*] (i.e. lower=1, upper= -1) denote that the model must contain respectively “exactly two” and “one or more” instances. Moreover, the *isOrdered* attribute specifies whether the order of instances (in a multi-instance model) must be respected.

The Functional Description is similar to the operation definition in MOF 1.4 [22]. However, it introduces two new features. Firstly, in MOF operations, a parameter type is limited to be a single metaclass. Therefore we cannot define, for example, a model including both UML classes and UML packages. In the Functional Description, *SpecificModelType* can define more flexible types because it can reference more than one metaclass. Secondly, in MOF operations, the "any" model type parameter doesn’t exist. Thus, the Functional Description can describe a wider range of services.

Our approach supports type checking automation. A service description repository can be built from our metamodel, based on technologies such as Java Metadata Interface (JMI) [14] or Eclipse Modelling Framework (EMF) [10]. This repository offers an API for manipulating service descriptions. This API allow us to write type compatibility checking rules in Java.

3.2 Describing Concrete Connection

In section 2.2, we have already explained that the tools heterogeneity causes difficulty for users. However, it is not practical in the real world to limit all tools to only one model representation form and one interface style. Moreover, each model representation form and interface style has its own advantages. For instance, object forms (e.g. JMI, EMF) provide model manipulation facilities while XMI format is better for model exchange. As for interface styles, it is simple and convenient to call local tools' services via an API while remote access mechanisms such as CORBA or Web Service are suitable for multi-user tools. This trade-off leads us to the following design principles:

EntryPoints: We provide a set of *EntryPoints* – concrete methods to call modelling services - allowing tool implementers to choose an EntryPoint suitable for their tools. EntryPoint definition will include model representation definition and interface style definition.

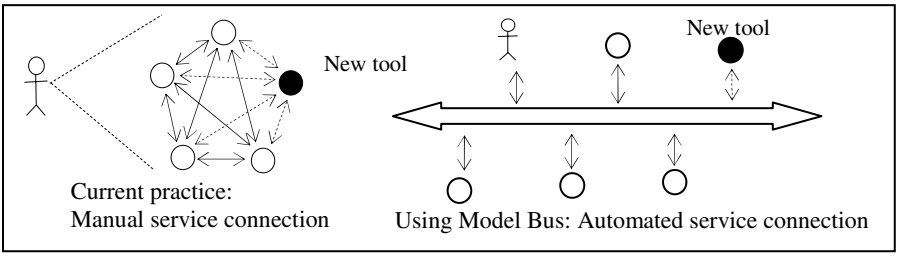

Fig. 5. How Model Bus enables concrete connection: Our goal is to generate code allowing services to be invoked. Thanks to this automated generation, the user does not need to be aware of tool heterogeneity.

Generation rules: For each EntryPoint, we also provide rules for generating 1) skeleton codes allowing services to be invoked and 2) service invocation codes for connecting an output of a service to an input of another service. Thanks to this automated generation, users who want to connect services do not need to be ware of service implementation.

The figure below illustrates how Model Bus solves the concrete connection problem. Without Model Bus, when a new tool is added, users will need to develop a dedicated method for connecting it with each existing tool. By using Model Bus, a new tool can automatically connect to others through the EntryPoints: the codes for connecting services will be generated using our generation rules.

EntryPoints: We propose a metamodel (c.f. next figure), for describing EntryPoints. The *EntryPoint* metaclass associates the concrete aspect with the abstract aspect of services. In other words, it specifies how the services defined abstractly in the Functional Description can be concretely invoked.

EntryPoint is specialized for representing each EntryPoint. We identify here three EntryPoints: *WsEntryPoint, CorbaEntryPoint, JmiEntryPoint*. Each EntryPoint is briefly defined in the table below according to model representation forms and interface styles.

Table 1. EntryPoint summary

EntryPoint	Model Representation Form	Interface style
WsEntryPoint	XMI	WSDL (Using SOAP message for invocation)
CorbaEntryPoint	CORBA objects (based on MOF-IDL)	CORBA (Using IIOP protocol for invocation)
JmiEntryPoint	Java objects (based on JMI)	Local Java API (Using Java method invocation)

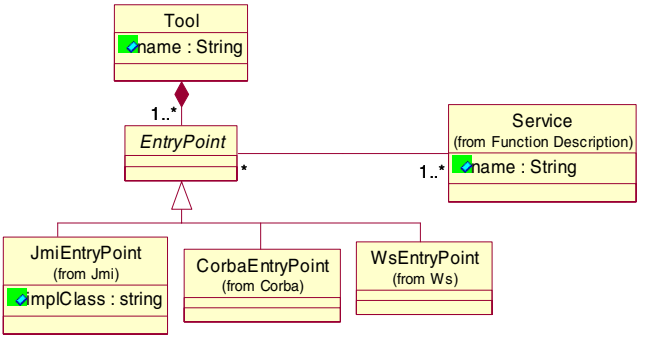

Fig. 6. EntryPoint metamodel: This metamodel describes how modelling services can be concretely invoked

All EntryPoints follows the similar principles: specifying the representation of service parameters (which are models) and specifying the service invocation mechanism via a specific interface style. In the rest of the article, we focus on the JmiEntryPoint. Our work concerning the WsEntryPoint is presented in [6].

Generation rules: For automating service access, we provide the generation rules which are used by both tool providers and users. First, they enable tool providers to generate skeleton codes allowing the services to be invoked. These skeleton codes will be used either for implementing the services or for delegating to existing implementation. Then, users can generate codes for invoking the services.
For JmiEntryPoint, a service description will be mapped to a Java interface. This interface will serve for both tool providers and users: It allows tool providers to provide the service implementation conforming to JmiEntryPoint. For users, it will be used in the generated codes that connect services (as we will later demonstrate in 4.2).

The rules for generating this Java interface are defined in terms of the correspondences between service description metaclasses and Java constructs as briefly shown the following table.

Table 2. Correspondences between service description elements and Java constructs

Service description elements		**Java constructs**
JmiEntryPoint		A singleton Java Interface <JmiEntryPoint.implClass>
Service		A Java method : Java.util.Map <Service.name>(java.util.Map inputMap)
Parameter	Input	A map entry (<Parameter.name>, value) in inputMap
	Output	A map entry (<Parameter.name>, value) in returned Map
Multiplicity	lower>=1	Corresponding map entry is required
Type	lower=0	Corresponding map entry is optional
	upper>1 or upper=*	Value must be instance of java.util.Collection
Type	PrimitiveType	Basic Java types (e.g. java.lang.String, java.lang.Boolean)
	EnumerationType	javax.jmi.reflect.RefEnum
	ModelType	javax.jmi.reflect.RefObject

A JmiEntryPoint is mapped to a Java interface. Each service referred by the *JmiEntryPoint* will be mapped to a Java method "java.util.Map <Service.name> (java.util.Map inputMap)". *inputMap* allows the service's input parameters to be passed as name-value pairs in the map data structure (java.util.Map). Likewise, the returned map will contain the name-value pairs of all output parameters.

The rest of the metaclasses (Parameter, Multiplicity, Type) serve as constraints on parameter values: *PrimitiveType* is mapped directly to Basic Java types (e.g. java. lang.String, java.lang.Boolean). For *ModelType*, the parameter values must be objects representing metaclass instances in JMI repositories (i.e. java.jmi.reflect. RefObject). For the optional parameter (i.e. MultiplicityType.lower>0), the map entry representing the parameter's value can be absent. For the parameter containing multiple objects (i.e. MultiplicityType.upper>1), the class java.util.Collection is used for holding the objects.

4 Model Bus Example

We take the same example UML-to-EJB for illustrating how Model Bus can solve the service connection difficulties.

4.1 Solving Functional Connection

For solving functional connection problem, we define each tool (UmlRepository, UmlToEjb, CodeGeneration) using the Functional Description metamodel. The result is shown in the following table.

The first tool, ***UmlRepository***, offers two services: *findClass* and *findPackage*. The former returns a UML class from a given name while the latter returns a UML package. The second tool, ***UmlToEjb***, offers the *transform* service that transforms UML packages (instances of metaclass *Model_Management::Package* in the UML

Table 3. Example of Functional Descriptions

Tool	Service	Parameter	Direction /Multipicity	Type
Uml Repository	findClass	className	In [1..1]	PrimitiveType (String)
		class	Out [1..1]	SpecificModelType (Foundation::Core::Class)
	findPackage	packageName	In [1..1]	PrimitiveType (String)
		package	Out [1..1]	SpecificModelType (Model_Management::Package)
UmlToEjb	transform	sourceModel	In [1..*]	SpecificModelType (Model_Management::Package)
		targetModel	Out [1..*]	SpecificModelType (ejb::EjbComponent)
Code Generation	generateSingle Component	ebjComponent	In [1..1]	SpecificModelType (ejb::EjbComponent)
	generate Components	ebjComponents	In [1..*]	SpecificModelType (ejb::EjbComponent)

metamodel) into instances of *EbjComponent* (defined in the EJB metamodel). The last tool, ***CodeGeneration***, offers two services: *generateSingleComponent* and *generateComponents*. The former requires a single *EbjComponent* instance while the latter requires a collection of *EbjComponent* instances.

For connecting the services, users must choose one service for each tool. Since the UmlRepository tool and CodeGeneration tool propose more than one service, appropriate choices must be made. The next figure shows the choices that the user makes (i.e. *findPackage, transform, generateComponents*).

To verify that the choices are correct, the user can use the following rules to check automatically the type compatibility of the inputs and outputs of the connected services.

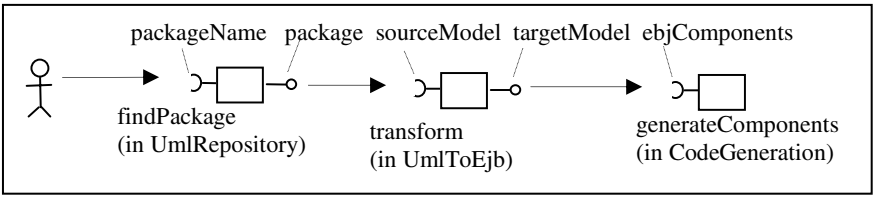

Fig. 7. Example of modelling service connections: When the modelling services are precisely described, we can identify whether the inputs and outputs of them are compatible and can be connected.

The *findPackage* & *transform* services: The output parameter *package* is connected to the input parameter *sourceModel*. The model types of both parameters correspond to the same metaclass (Model_Management ::Package) and hence are compatible. Their granularities are also compatible ([1..1] → [1..*]). Therefore, the service connection is correct.

The *transform* & *generateComponents* services: The output parameter *targetModel* is connected to the input parameter *ebjComponents*. The model types of both parameters correspond to the same metaclass (ejb::EbjComponent). Their granularities are also compatible ([1..*] → [1..*]). Therefore, the service connection is correct.

If the user made bad choices, the similar analysis as above could detect bad service connections. For example, the connection of the *findClass* service to the *transform* service would be incorrect because the model types of their parameters are incompatible (metaclass Foundation::Core::Class vs metaclass Model_Management ::Package). The connection of the *transform* service to the *generateSingleComponent* service would also be incorrect because the granularities of their parameters are incompatible ([1..*] → [1..1]).

4.2 Solving Concrete Connection

As described, EntryPoint is used for specifying how to invoke services. We will illustrate how to connect services via *JmiEntryPoint*. By using the generation rules, Java interfaces can be generated from the service descriptions as shown below:

```
public interface UmlRepository {
  public Map findPackage(Map inputMap);
  public Map findClass(Map inputMap);              }
public interface UmlToEjb {
  public Map transform(Map inputMap);              }
public interface CodeGeneration {
  public Map generateSingleComponent(Map inputMap);
  public Map generateComponents(Map inputMap);     }
```

To execute all the service connections, only a simple code is needed for connecting them. For brevity, only the connection of *transform* service and *generateComponents* service is shown below. The two services are connected by linking the *targetModel* output to the *ebjComponents* input. To connect them, first the service producing the output (i.e. *transform*) is invoked (line a). Then, the output is extracted from the map data structure (line b). Next the output is linked to the input by putting it in the map (line d). Finally, the service consuming the input (i.e. *generateComponents*) is invoked (line e).

```
a. Map transformOutput = UmlToEjb.transform(transformInput);
b. Collection targetModel = (Collection)
   transformOutput.get("targetModel");
c. Map generateComponentsInput = new Hashtable();
d. generateComponentsInput.put("ebjComponents", targetModel);
e. Map generateComponentsOutput =
   CodeGeneration.generateComponents(CodeGenerationInput);
```

The codes for linking other parameter pairs follow the same pattern. For this reason, by specifying a parameter pair to be linked, we can automatically generate the code.

5 Proof of Concepts: Model Bus Integrated Environment (MBIE)

We have implemented a Model Bus prototype on the Eclipse platform. This prototype is called Model Bus Integrated Environment (MBIE). MBIE provides two facilities. Firstly, it allows users to browse all service descriptions. In particular, users can examine the signature of each modelling service. Secondly, MBIE automatically generates a GUI from service descriptions. Users can then use this GUI for invoking any service. This implementation proves that 1) service descriptions can be automatically processed and 2) The invocation of any service can be automated in the sense that users need not writing codes.

The following figure illustrates the MBIE architecture. MBIE is connected to the bus like other tools. Instead of accessing the bus directly, users can alternatively use the GUI facilities provided by MBIE to interact with tools. MBIE contains two components: Functional Management and EntryPoint Management. The Functional Management allows users to browse service descriptions. The EntryPoint Management allows users to invoke the chosen service via an automatically generated GUI.

Functional Management provides a GUI, called *Functional View* (c.f. the next figure), which lets users explore tools' Functional Descriptions (i.e. modelling service signatures) and then select a service to be invoked.

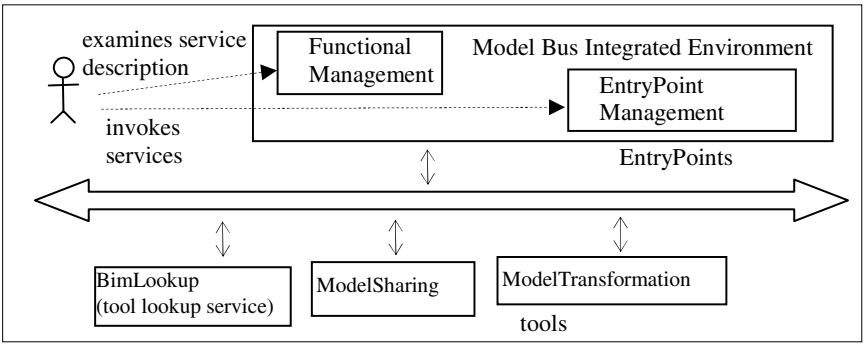

Fig. 8. MBIE Architecture: MBIE is an environment that allows users to use modelling services of any tools. It has two parts. Functional Management allows users to examine available services and to determine functional connection. EntryPoint Management allows users to invoke services transparently form service implementation.

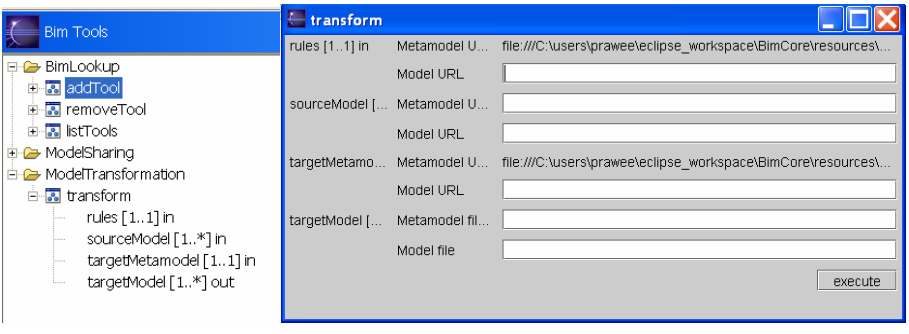

Fig. 9. GUI of MBIE: Functional Management (left) and EntryPoint Management (right)

As shown in the figure 9, three tools are available: *BimLookup,* which provides lookup services for service descriptions*, ModelSharing*, which offers a model storage service, and *ModelTransformation,* which proposes a transformation service based on Transformation Rule Language [8]. This Functional View also shows that the *ModelTransformation* tool offers the *transform* service having four parameters (rules, sourceModel, targetMetamodel and targetModel).

EntryPoint Management allows users to invoke a modelling service through the *Service Call Dialog,* which is automatically generated from the signature of the service. Firstly, this GUI takes inputs from users. Then the service is invoked using the appropriate EntryPoint. Finally, the results are returned to users.

Figure 9 shows a Service Call Dialog for invoking the *transform* service. This dialog allows users to supply three inputs parameters (rules, sourceModel, and targetMetamodel) and to receive the result (targetModel).

6 Related Works

The works related to Model Bus concern frameworks where tools can be integrated. Our previous work, Integrated Transformation Environment (ITE) [5], allows users to use many transformation engines in the same environment. Compared to Model Bus, the ITE approach is more restrictive. Firstly, ITE limits integrated tools to be model transformation tools having one input model and one output model. Model Bus can describe more flexible functionalities (i.e. any number of inputs and outputs). Secondly, ITE uses metamodels for defining model types. Model Bus proposes a more precise definition of model types using metaclasses.

The providers of some repository implementations such as Netbeans Metadata Repository [18], Eclipse Modeling Framework [10], and Univers@lis [3] propose frameworks where all tools share the same central repository. This approach allows tools to be tightly integrated: all models are stored in the same repository and hence can be shared among all tools. For example, model visualization, transformation and code generation tools are integrated in the same Univers@lis repository. However this approach has two disadvantages. Firstly, it does not address how functional connection can be checked. On the other hand, Model Bus offers a metamodel for describing modelling service signatures and also rules for checking the model type compatibility. Secondly, the central repository approach is not suitable for distributed environments: the remote access to the central repository is costly and can expose security risks. To overcome this problem, Model Bus includes the Web Service EntryPoint for supporting distributed tools.

Middleware architectures such as Web Service [33] and CORBA are similar to Model Bus in the sense that they allow services (or services) to be described (e.g. CORBA - IDL, Web Service - WSDL) and they define interfaces for invoking services (CORBA - IIOP, Web Service - SOAP Bindings). However, those architectures do not support services that have models as inputs and outputs. Model Bus is dedicated to the modeling domain. It defines model types and model representation forms to be used in modelling services.

The workflow process definition language (WPDL) [33] allows process connections to be specified. Some work for applying WPDL for connecting modeling tools [12] has been made. However this work did not address the functional and concrete connection problems. For this moment, Model Bus does not have a metamodel for expressing how services are connected. We think that a subset of WPDL can be reused for expressing service connection in Model Bus.

7 Conclusion and Perspectives

Model Bus allows modelling services to be connected. To connect services, the functional connection and the concrete connection problems must be solved. To solve the functional connection problem, we proposed the Functional Description metamodel for describing modelling service signatures. In particular a precise model type definition was described. As a result, type compatibility of the connected parameters can be automatically checked. To solve the concrete connection problem, we defined a set of EntryPoints allowing services to be invoked. We have shown how

the service descriptions can be used to automatically generate a Java interface for tool providers to implement the services and for users to invoke the services. We have also demonstrated how to generate codes for automating service connections.

The Model Bus prototype is implemented in Eclipse Platform. It offers users the high-level facilities for browsing services and invoking any services. This prototype proves that modelling service description can be described and Model Bus automates the service invocation.

For future work, we plan to advance this research particularly in two aspects. At this time, modelling services are described in terms of model element types and model granularities. However, some services require model types to be more specific, for example, a service that requires a UML class having at least one attribute, a service that requires a UML class with stereotype ‹‹Table››. Therefore, we plan to augment model type semantics with Object Constraint Language (OCL). We think that this improvement will ensure better the correctness of service connections.

For the second aspect, we want to propose a method for rigorously expressing how services are connected. For example, "output A of service S1 is connected to input B of service S2". In particular, we need a metamodel for describing the structure of this information. This metamodel will allow us to specify software development scenarios involving many modelling services. We also look forwards to automating the execution of those scenarios.

References

1. ArcStyler, http://www.io-software.com
2. ArgoUML, http://www.argouml.tigris.org
3. M. Belaunde: A Pragmatic Approach for Building a User-friendly and Flexible UML Model Repository, 2nd International Conference on The Unified Modelling Language (UML'99), 1999.
4. J. Bézivin et al.: First experiments with the ATL model transformation language: Transforming XSLT into XQuery, 2nd OOPSLA Workshop on Generative Techniques in the context of MDA, 2003.
5. X. Blanc et al.: Towards an Integrated Transformation Environment (ITE) for Model Driven Development (MDD), to be published in the Invited Session Model Driven Development, The 8th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI 2004), July 2004.
6. X. Blanc, M-P. Gervais, P. Sriplakich: Modeling Services and Web Services: Application of ModelBus, to appear in the 2005 International Conference on Software Engineering Research and Practice (SERP'05), 2005.
7. K. Czarnecki, S. Helsen: Classification of Model Transformation Approaches, 2nd OOPSLA Workshop on Generative Techniques in the context of MDA, 2003.
8. T. Gardner et al.: A review of OMG MOF 2.0 Query /Views /Transformations Submissions and Recommendations towards the final Standard, http://www.omg.org/docs/ad/03-08-02.pdf
9. D. Hearnden, K. Raymond, J. Steel: Anti-Yacc: MOF-to-Text, EDOC 2002.
10. Eclipse Modeling Framework, http://www.eclipse.org/emf
11. Eclipse UML, http://www.omondo.com

12. G. van Emde Boas: From the Workfloor: Developing Workflow for the Generative Model Transformer, 2nd OOPSLA Workshop on Generative Techniques in the context of MDA, 2003.
13. Hamie: Towards Verifying Java Realizations of OCL-Constrained Design Models Using JML, 6th IASTED International Conference on Software Engineering and Applications, 2002.
14. Java Community Process: Java Metadata Interface (JMI) Specification, http://www.jcp.org, 2002.
15. Ledeczi et al.: The Generic Modeling Environment, Workshop on Intelligent Signal Processing, 2001.
16. MIA, http://www.model-in-action.fr
17. ModFact, http://modfact.lip6.fr
18. NetBeans Metadata Repository, http://mdr.netbeans.org
19. Objecteering, http://www.objecteering.com
20. OMG: Human-Usable Textual Notation (HUTN) Specification, document no: ptc/04-01-10, 2003.
21. OMG: MDA Guide Version 1.0.1, document no: omg/2003-06-01, 2003.
22. OMG: Meta Object Facility (MOF) Specification version 1.4, document no: formal/2002-04-03, 2002.
23. OMG: Request for Proposal MOF2.0 Query /Views /Transformations, document no: ad/2002-04-10, 2002.
24. OMG: Request for Proposal UML 2.0 OCL, document no: ad/2000-09-03, 2001.
25. OMG: UML 2.0 Superstructure Specification, document no: ptc/03-08-02, 2004.
26. OMG: Unified Modeling Language Specification version 1.4, document no: formal/01-09-67, 2001.
27. OMG: XML Metadata Interchange (XMI) Specification version 2.0, document no: formal/03-05-02, 2003.
28. Porres: M. Alanen, A Generic Deep Copy Algorithm for MOF-Based Models, Model Driven Architecture:Foundations and Applications, 2003.
29. Poseidon, http:// www.gentleware.com
30. Rational Rose, http://www.rational.com
31. D. Riehle & al.: The Architecture of a UML Virtual Machine, OOPSLA 2001.
32. UMT-QVT: http://umt-qvt.sourceforge.net
33. W3C: Web Services Architecture, http://www.w3.org/TR/ws-arch, 2004.
34. Workflow Management Coalition: Workflow Process Definition Language, document no: WFMC-TC-1025, version 1.0, 2002.