Model Bus:
Towards the interoperability of modelling tools

Xavier Blanc, Marie-Pierre Gervais, 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. To develop software with MDA, users need to apply a wide range of model operations (i.e. operations taking models as inputs and outputs). All the operations that users require for realizing the MDA software production are not likely to be supported in a unique tool. Consequently, users need to connect different tools for applying all the required operations. More precisely, they need the means for linking the output of an operation to the input of another operation even if both operations are not supported by the same tool. Due to the heterogeneity of tools in both functionalities and the way users interact with them, connecting tools is difficult. Nevertheless, today there are few works addressing this problem. Therefore, we propose an architecture and a prototype enabling the operations of different tools to be connected.

1 Introduction

According to Model Driven Architecture (MDA), models are treated as first-class elements in software development [20]. MDA application requires a wide range of model operations such as model edition [14], model storage [14], model manipulation [21][13], code generation [8] and model transformation [4][6][7]. 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 [24], Object Constraint Language 2.0 [23]). We think that this list of model operations is not exhaustive and that new ones will be identified in the future.

A lot of tools are now available commercially or as open source and provide various model operations. For example, NetBeans Metadata Repository [17], ModFact [16], Eclipse Modeling Framework (EMF) [9], and Univers@lis [3] propose model storage and model manipulation. Rational Rose [29], Objecteering [18], EclipseUML [10], Poseidon [28] and ArgoUML [2] propose UML model edition and code generation. ArcStyler [1], MIA [15], and UMT-QVT [31] propose model transformation. Although these tools cover a lot of model operations, some operations, such as UML model execution [30], OCL constraint verification [12], deep model copy operation [27], are not commonly supported by commercial tools.

Currently, there is no single tool that proposes all model operations. There are too many operations to be supported by a unique tool as we have already shown a non-exhaustive list of operations. However, the MDA software production requires a wide range of model operations to be used in different software development activities (e.g. analysis phase, design phase, test phase, implementation phase). Therefore, the fact that model operations are not supported by the same tool must not prevent models to be processed by all the operations required in the MDA software production. In particular,

when users need to use two tools conjointly, they must be able to send an output model produced by an operation of tool T1 as an input to an operation of tool T2. We use the term "*operation connection*" to denote the action of linking an operation's input to another operation's output.

Connecting model operations is a difficult problem. We identified that this problem includes two sub-problems: *functional connectivity* and *protocol connectivity*. Functional connectivity ensures that the inputs and outputs of model operations have compatible types and can then be connected. This sub-problem particularly concerns the type compatibility of models. Protocol connectivity ensures that model operation connections can be realized. In particular, the connected operations must agree in a model representation form and in a mechanism for transmitting models. Please note that we use the term "protocol" in the sense that it can manage model transmission between distributed operations (i.e. two connected operations are executed on different machines).

Moreover, it should be noted that connection of model operations must be automated in order to let users concentrate on the profits of the connection and not on how to technically realize the connection. Right now, because tools are only documented with manuals (in natural language), we argue that functional connectivity cannot be automated as there is no possibility for processing the model operation descriptions in order to check the type compatibility.

On the other hand, each tool has its own model representation forms for encoding its operations' inputs and outputs. The model representation can take either textual forms (e.g. XML Metadata Interchange (XMI) [26], Human-Usable Textual Notation (HUTN) [19]) or object forms in model repositories (e.g. Java Metadata Interface (JMI) [13], EMF Repository [9]). Moreover, each tool provides a different way for users to interact to its operations (i.e. some tools provide graphical user interfaces [29][18], some are executed via command lines [16] and others propose APIs for calling model operations [9]). To connect operations of different tools, a dedicated method must be used for each pair of tools. This effort is costly and can only be done manually. For this reason, protocol connectivity is not automated.

Despite the needs for connecting model operations, 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 the model world. Although the EMF offers the integration of modeling tools into Eclipse, it does not address at all the functional connectivity 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 connectivity and the protocol connectivity problems. Model Bus is mainly based on middleware technologies such as CORBA and Web Services but it adds new features for dealing with modeling aspects. Model Bus enables the automation of model operation connections. We have implemented a prototype of Model Bus inside the Eclipse platform and we have connected several model operations proposed by the ModFact tools.

This paper is organized as follows. Section 2 discusses the difficulties of model operation connection. Theses difficulties will be illustrated through an example in section 3. Section 4 presents Model Bus architecture and explains how Model Bus can automate model operation connection. In section 5, Model Bus is used to solve the difficulties illustrated in the previous example. Section 6 validates our concepts by presenting our prototype. Section 7 compares our approach with others. The last section concludes our work and presents research perspectives.

2 Model operation connection problem

The difficulties of model operation connections are related to two sub-problems: functional connectivity and protocol connectivity.

2.1 Functional connectivity

Functional connectivity ensures that the inputs and outputs of model operations have compatible types and can then be connected (i.e. an operation can consume the output of another operation). Though the type compatibility is a well-known problem, it has not been addressed in the model world. Unlike classical data type, the model type compatibility is not a trivial problem because today, there is no well-known, precise definition of model types. Finding such a definition is also complex because there are uncountable kinds of models (e.g. UML models, SPEM models, CWM models …). We identify the characteristics that must be taken into accounts for defining model types. Then we relate them to the model type compatibility problem.

Metaclasses: It is a common practice to use a metamodel to define the type of model. In other words, a model is anything conforming to the metamodel. We argue this approach is not sufficiently precise. Firstly, there is no agreement on what is precisely a metamodel: Is it a single metapackage (i.e. MOF package) or a collection of metapackages? Secondly, a metamodel contains several metaclasses. As a result, this model type definition allows models to be instances of any metaclasses. However, an operation may be capable of processing the instances of only some metaclasses (e.g. operation capable of processing a UML class but not UML use case).

Therefore, metaclasses are required to be identified for defining a model type. The figure below shows how an input type and output type of a model operation is defined. The dark circles represent two metaclasses whose instances can be processed by the operation and a metaclass whose instances are produced by the operation.

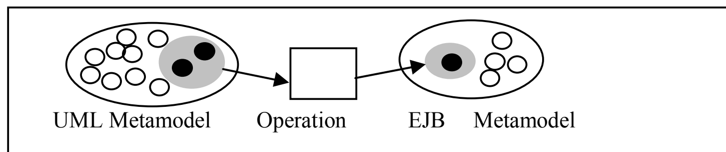

Fig. 1. Roles of metaclasses in the model type definition

"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 we have found several operations that require the "any" model type (e.g. generic transformation engine [22], or reflective MOF repository [16] [17]).

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 number of instances that is allowed, for example, a single instance, no more than two instances, or any number of instances. 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 problem. We present some checking rules that use those characteristics.

Metaclasses: 1) An output model type (T1) is conformed to an input model type (T2) if the metaclasses corresponding to T1 are contained in the set of metaclasses corresponding to T2.

“Any” vs. specific model types: 1) All specific model outputs are compatible to an "any" model input. 2) An "any" model output may not be conformed to a specific model input depending on the runtime type of that "any" model output. Therefore, the metaclass checking is necessary at runtime.

Model granularity: 1) The instance number range of the output model type must be included in the one of input model type. For example, "a single instance" is included in "no more than two instances". 2) Connecting an unordered-collection output to an ordered-collection input may cause non-deterministic results; therefore users should receive warnings.

It can be observed that if those characteristics were precisely specified in a well-defined format, the automation of the checking rules would be feasible. However, in current practice, the input and output types of model operations are neither precise nor well-formed. It is because they are described in natural languages (i.e. in tool manuals). That is why the functional connectivity problem is difficult to be solved.

2.2 Protocol connectivity

Protocol connectivity concerns how model operation connections can be *concretely* realized. We identify that the difficulties of protocol connectivity are caused by two heterogeneity aspects of tools: 1) the model representation forms the tools choose for encoding their operations' inputs and outputs and 2) how users interact with tools for applying their operations (interaction style).

Model representation forms: Tools currently available have their own model representation forms. On one hand, some tools use models represented in textual formats. For example, Rational Rose has its proprietary format (MDL). UMT-QVT can read and write models in the HUTN format. Poseidon and ArgoUML use the XMI format. On the other hand, some other tools require models in object representations. For example, the ModFact transformation engine requires object-form models in the JMI repository and the Java code generation tool, which is a part of EMF, requires object-form models in the EMF repository.

Interaction styles: The way users interact with tools can vary from a tool to another. For example, Rational Rose offers to users a graphical user interface (GUI) for applying code generation operation 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 operation on an EMF repository. Moreover, tools that support multi-users can provide remote access. For instance, ModFact repository allows the model manipulation operation to be accessible through the CORBA RPC. We can also anticipate tools offering Web Service access to their operations.

In the first aspect, the heterogeneity does not allow the operations of different tools to share input and output models. For example, an operation of a tool using object-form models (in a JMI repository) cannot process models produced by an operation of another tool using models written in the textual HUTN format.

In the second aspect, the heterogeneity of interaction styles (e.g. GUI, command line, API or RPC) force users to switch from one interaction style to another for connecting operations. Moreover, some interaction styles, such as GUI and command line are only

intended for manual use. Other interaction styles such as API, Web Service, allow the operations to be used in programs. However, since they are too heterogeneous, the programs must be manually written for connecting operations. As a result, automating the operation connections is difficult.

This heterogeneity problem is a well-known problem and several solutions have already been proposed (e.g. CORBA, Web Service). However, those solutions do not address the interoperability of the modeling tools where models are first-class data to be exchanged among model operations.

3 Operation connection example

The difficulties of tool connections will be illustrated through an example: a UML to Enterprise Java Bean (EJB) transformation. This example uses the following scenario: First a user will find a UML model in a *UML Repository* tool. This tool gets a model name as an input and returns a UML model as an output. This output is connected to the input of a *Transformation* tool for transforming the UML model to an EJB model. The output of the transformation tool (i.e. EJB model) will be connected to the input of a *Code Generation* tool for generating an EJB application (i.e. code). The figure below illustrates the operation connections in this example (i.e. from UML Repository to Transformation and from Transformation to Code Generation).

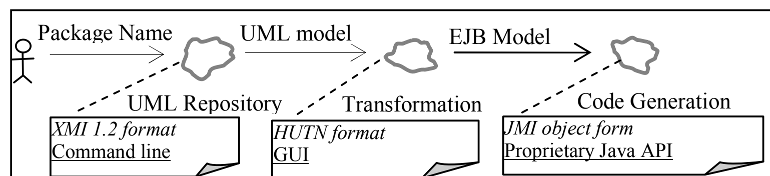

Fig. 2. Example: a UML to EJB transformation

We will detail hereafter the functional connectivity and protocol connectivity problems related to this scenario.

3.1 Functional connectivity difficulties

We will show that the type characteristics identified in II.1 are necessary for checking the type compatibility.

Metaclasses: We assume that the UML Repository can support two operations: *findPackage* and *findClass* (c.f. the figure below). The *findClass* and *findPackage* operations require a class name and a package name as input (i.e. model name) respectively. As a result, the user must choose, among *findPackage* and *findClass*, the operation compatible to the transformation operation. However, this choice cannot be made if the tool manual informally states that the transformation operation takes a “UML model” and returns an “EJB model” because the user does not know whether the required UML model must be a UML package or a UML class. Therefore, the input and output of the transformation operation must be described in terms of metaclasses (i.e. *Package* and *Class* metaclasses in UML metamodel). If the instance types are specified as shown in the following figure, the user will see that the *findPackage* operation is compatible to the transformation operation and should be chosen.

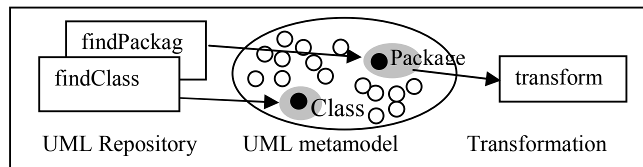

Fig. 3. Checking type compatibility using metaclass matching

"Any" vs. specific model types: The *generic* transformation operation that takes and produces any kind of models (i.e. "any" model type) has already been identified in [22]. If the transformation tool in this example proposed this operation, the user would need to verify at run-time the actual type of its output model (as it can be any kind of models), before connecting it to the code generation operation, which requires a specific model type input.

Model granularity: The transformation operation in this example does not specify how many instances (of metaclasses) the target model will contain. If the target model contains multiple instances while the code generation operation can handle only one instance, the operation connection will cause errors.

3.2 Protocol connectivity difficulties

Figure 2 also illustrates the model representation form (*italic text*) and the interaction style (underlined text) of each tool. The tools in this example use different model representation forms. Therefore, we can remark problems in the following connections: the connection of the UML Repository's output (XMI 1.2) to the Transformation's input (HUTN) and the connection of the Transformation's output (HUTN) to the Code Generation's input (JMI objects).

Furthermore, some interaction styles, such as command lines or GUIs, require manual interventions from the user. In other words, the user has to get back the output of an operation and manually forward it to another operation for connecting them. For this reason, operation connections cannot be automated. It is also not convenient for the user to switch from one interaction style to another when the interaction styles are heterogeneous. These disadvantages are illustrated as follows:

Connecting UML Repository to Transformation: First, the user has to use a command line for getting a UML model from the UML Repository. Then, he/she must manually transfer the UML model to the Transformation operation through GUI.

Connecting Transformation to Code Generation: First, the user will use the GUI of the Transform operation for getting back the transformation result (i.e. the EJB model). Then he/she must write a Java program for invoking code generation operation through the Java API.

4 Model Bus

4.1 Solving functional connectivity

The design principle of our approach is to offer a uniform way for describing the model operations of tools. In particular, the input and output types of the operations must be precisely defined in order that the operation connections 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 global view of model operations is unclear and does not enable the type compatibility checking in connected operations. Our approach proposes a uniform view where operations are similar to software components having precise input and output definitions.

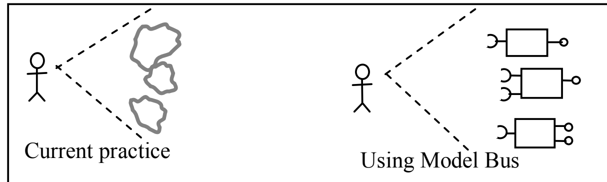

Fig. 4. Model operations viewed as software components

We propose a metamodel, called *Functional Description* (c.f the next figure). This metamodel describes the signatures of model operations in an abstract way. Like classical operations, model operations can have inputs and outputs of basic types (e.g. String, Integer, Boolean, Enumeration). However they have a new important feature: their input and output types can be models.

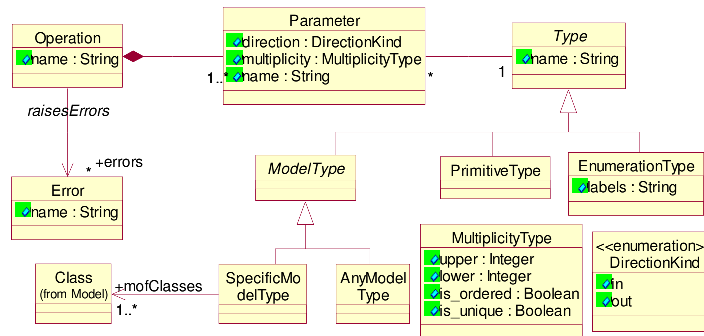

Fig. 5. Functional Description metamodel

The Functional Description metamodel addresses the problem of the type compatibility verification by allowing operations to be sufficiently described. The model characteristics presented in II.1.1 can be precisely specified as follows.

Metaclasses: The metaclass *SpecificModelType* references MOF metaclasses whose instances can be contained in input and output models. For example, in the description of an operation requiring a UML use case, the SpecificModelType will point to the metaclass *UseCase* in the UML metamodel. As a result, if users are not sure what exactly UML use cases are, or which version of UML the operation requires, they can obtain the complete definition of UML use cases (from a specific version of the UML metamodel) referenced by this description.

"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. The different values of *upper* and *lower* can express

various granularity semantics. 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) has particular meanings.

The Functional Description is similar to MOF 1.4 operation definition [21]. 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 type which can be either a UML class or a UML package. 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, we introduced the *AnyModelType* metaclass in the Functional Description.

For automation aspect, we built a repository based on Java Metadata Interface (JMI) 13 for storing all operation descriptions defined by the Functional Description metamodel. This repository offers an API for manipulating operation descriptions. Using this API, a program can navigate in operation descriptions for checking type compatibility of connected operations. Therefore, our solution supports the automation of type compatibility verification.

4.2 Solving protocol connectivity

In section 2.2, we have already explained that the tools heterogeneity causes difficulty for users. However, it is not a good idea to limit all tools to only one protocol (i.e. one model representation form and one interaction style). Each protocol has its own advantages. For instance, it is simple and convenient to execute local tools' operations via an API call. For multi-user tools, remote access protocols such as CORBA or web service are suitable. This trade-off leads us to the following design principles:

Model Bus protocols: There should be a set of well-known and well-defined protocols allowing tool implementers to choose a protocol suitable for their tools. We will call those protocols *Model Bus protocols*. Each protocol definition will include model representation definition and interaction style definition.

Generation rules: For each Model Bus protocol, we also provide rules for generating 1) skeleton codes allowing operations to be invoked and 2) operation invocation codes for connecting outputs of an operation to inputs of other operations. Therefore, our solution can automate operation connections.

The figure below illustrates how Model Bus solves the protocol connectivity 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 connect to others through the Model Bus protocols without any manual efforts: the codes for connecting operations will be automatically generated using our generation rules.

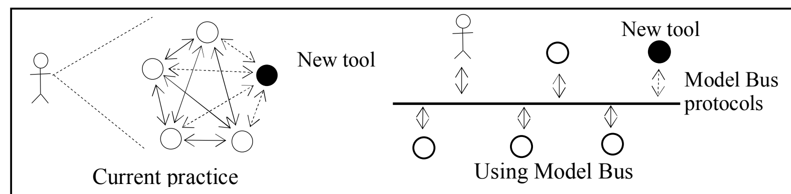

Fig. 6. How Model Bus enables protocol connectivity

Model Bus protocols: We propose a metamodel, called Protocol Description (c.f. next figure), for describing Model Bus protocols. The *EntryPoint* metaclass is used for

associating the protocol aspect with the functional aspect of operations. More precisely, this metaclass is associated with the *Operation* metaclass defined in the Functional Description metamodel. This association specifies how the operations defined abstractly in the Functional Description can be concretely invoked.

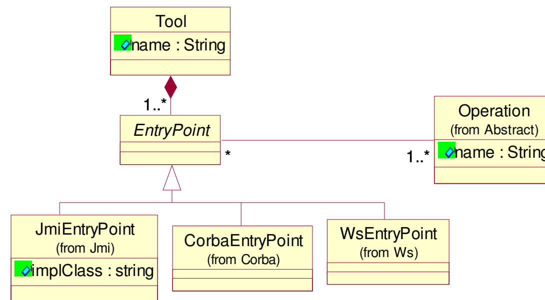

Fig. 7. Protocol Description metamodel

EntryPoint can be specialized for representing each Model Bus protocol. We identify here three Model Bus protocols: *WsEntryPoint, CorbaEntryPoint, JmiEntryPoint*. Each protocol is briefly defined in the next table according to model representation form and interaction style.

At this time, we have only implemented JmiEntryPoint. Therefore, in the rest of article, we will focus on this EntryPoint. However, the implementation of other protocols will follow the same principle.

Table 1. Model Bus protocols

EntryPoint	Model Representation Form	Interaction Style
WsEntryPoint	XMI	SOAP messages (Web Service)
CorbaEntryPoint	CORBA objects (based on MOF-IDL)	CORBA
JmiEntryPoint	Java objects (based on JMI)	Local Java API

Generation rules: For automating tool access, the tool description (i.e. Functional and Protocol Descriptions) is used by both tool providers and users: Firstly, tool providers use it to generate skeleton codes allowing operations to be invoked. These skeleton codes will be used either for implementing the operations or for delegating to existing implantation. Secondly, users will use it to generate codes for invoking the operations.

For JmiEntryPoint, a tool 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 operation implementation conforming to JmiEntryPoint protocol. For users, it will be used in the generated codes that connect operations (as we will later demonstrate in 5.2).

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

A JmiEntryPoint is mapped to a Java interface. Each operation referred by the *JmiEntryPoint* will be mapped to a Java method "java.util.Map <Operation.name> (java.util.Map inputMap)". *inputMap* allows the operation'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.

Table 2. Correspondences between tool description metaclasses and Java constructs

Tool description metaclasses		**Implementation**
JmiEntryPoint		A singleton Java Interface <JmiEntryPoint.implClass>
Operation		A Java method : Java.util.Map <Operation.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 Type	lower>=1	Corresponding map entry is required
	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

5 Model Bus example

We take the same example in section 3 for illustrating how Model Bus can solve the operation connection difficulties.

5.1 Solving functional connectivity

For solving functional connectivity problem, we define each tool using the Functional Description metamodel. The result is shown in the following table.

Table 3. Example of Functionality Descriptions

Tool	**Operation**	**Parameter**	**Direction /Multiplicity**	**"Any" or specific model type /Metaclass identification**
Uml Repository	findClass	className	In [1..1]	PrimitiveType (String)
		class	Out [1..1]	SpecificModelType (Model_Management::Package)
	findPackage	packageName	In [1..1]	PrimitiveType (String)
		package	Out [1..1]	SpecificModelType (Foundation::Core::Class)
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)

The first tool, ***UmlRepository***, offers two operations: *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* operation that transforms UML packages (instances of metaclass *Model_Management::Package* in the UML metamodel) into instances of *EbjComponent* (defined in the EJB metamodel). The last tool, ***CodeGeneration***, offers two operations: *generateSingleComponent* and *generateComponents*. The former requires a single *EbjComponent* instance while the latter requires a collection of *EbjComponent* instances.

For connecting the operations, users must choose one operation for each tool. Since the UmlRepository tool and CodeGeneration tool propose more than one operation, 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 operations.

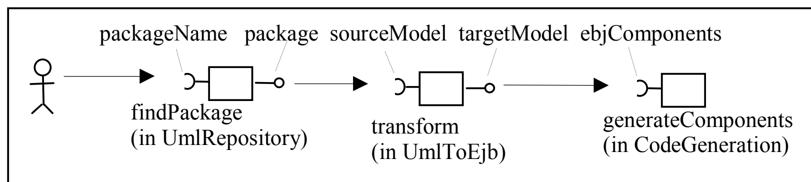

Fig. 8. Example of model operation connections

The *findPackage* & *transform* operations: The output parameter of the former operation (*package*) is connected to the input parameter of the latter operation (*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 operation connection is correct.

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

If the user made bad choices, the similar analysis as above could detect bad operation connections. For example, the connection of the *findClass* operation to the *transform* operation 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* operation to the *generateSingleComponent* operation would also be incorrect because the granularities of their parameters are incompatible ([1..*] → [1..1]).

5.2 Solving protocol connectivity

In this step, the Protocol Description metamodel is used for specifying how the user can interact with tools. Suppose that all tools provide *JmiEntryPoint* protocol (they may also provide other alternative Model Bus protocols). Hence, in the tool descriptions, all the instances of *Operation* (from the Functional Description metamodel) will be associated to

a *JmiEntryPoint* instance. Java interfaces can therefore be generated from the tool 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 operation connections, only a simple code is needed for connecting them. For brevity, only the connection of *transform* operation and *generateComponents* operation is shown below. The two operations are connected by linking the *targetModel* output to the *ebjComponents* input. To connect them, first the operation 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 entry (line d). Finally, the operation 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.

6 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 services. Firstly, it allows users to browse all tool descriptions. In particular, users can examine the signature of each model operation. Secondly, MBIE automatically generates a GUI from tool descriptions. Users can then use this GUI for invoking an operation of any tool. This implementation proves that 1) tool descriptions can be automatically processed and 2) The invocation of any operation 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 Protocol Management. The Functional Management allows users to browse tool descriptions. The Protocol Management allows users to invoke the chosen operation 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. model operation signatures) and then select an operation to be invoked. As shown in the figure, three tools are available: *BimLookup,* which provides lookup operations for tool descriptions*,*

ModelSharing, which offers a model storage operation, and *ModelTransformation,* which proposes a transformation operation based on Transformation Rule Language [7]. This Functional View also shows that the *ModelTransformation* tool offers the *transform* operation having four parameters (rules, sourceModel, targetMetamodel and targetModel).

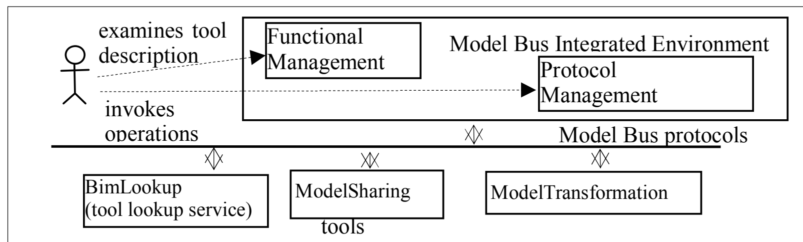

Fig. 9. MBIE Architecture

Protocol Management allows users to invoke a model operation through the *Operation Call Dialog,* which is automatically generated from the signature of the operation. Firstly, this GUI takes inputs from users. Then the operation is invoked using the appropriate protocol, specified by the Protocol Description. The invocation mechanism is transparent to users. Finally, the results are returned to users.

The figure below shows an Operation Call Dialog for invoking the *transform* operation. This dialog allows users to supply three inputs parameters (rules, sourceModel, and targetMetamodel) and to receive the result (targetModel).

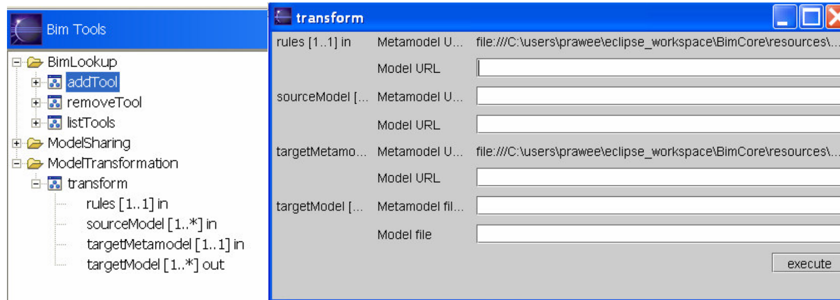

Fig. 10. Functional View (left) and Operation Call Dialog (right)

7 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. Since this approach is not sufficient, Model Bus proposes a more precise definition of model types using metaclasses.

The providers of some repository implementations such as Netbeans Metadata Repository [17], Eclipse Modeling Framework [9], 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 solve the functional connectivity problem. More precisely, it does not address how operation connection correctness can be checked. On the other hand, Model Bus offers a metamodel for describing model operation 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 protocol for supporting distributed tools.

Middleware architectures such as Web Service [32] and CORBA are similar to Model Bus in the sense that they allow operations (or services) to be described (e.g. CORBA's IDL, Web Service's WSDL) and they define protocols for invoking operations (CORBA's IIOP, Web Service's SOAP Bindings). However, those architectures do not support operations that have model inputs and output. Model Bus is dedicated to modeling aspects. It defines model types to be used in model operations and the protocols for invoking those operations.

The workflow process definition language (WPDL) [33] allows process connections to be specified. Some work for applying WPDL for connecting modeling tools [11] has been made. However this work did not address the functional and protocol connectivity problems. For this moment, Model Bus does not have a metamodel for expressing how operations are connected. We think that a subset of WPDL can be reused for expressing this aspect in the model world.

8 Conclusion and perspectives

Model Bus allows model operations to be connected. To connect operations, the functional connectivity and the protocol connectivity problems must be solved. To solve the functional connectivity problem, we proposed the Functional Description metamodel for describing model operation 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 protocol connectivity problem, we defined a set of Model Bus protocols allowing operations to be invoked. We have shown how the tool descriptions (i.e. Functional Description and Protocol Description) can be used to automatically generate a Java interface for tool providers to implement the operations and for users to invoke the operations. We have also demonstrated how to generate codes for automating operation connections.

The Model Bus prototype is implemented in Eclipse Platform. It offers users the high-level facilities for browsing tools and invoking an operation of any tools. This prototype proves that tool description can be automatically processed and Model Bus automates the operation invocation.

For future work, we plan to advance this research particularly in two aspects. At this time, model operations are described in terms of model element types and model granularities. However, some operations require model types to be more specific, for example, an operation that requires a UML class having at least one attribute, an operation

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 operation connections.

For the second aspect, we want to propose a method for rigorously expressing how operations are connected. For example, "output A of operation O1 is connected to input B of operation O2". 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 model operations. 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 & 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 & 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. K. Czarnecki, S. Helsen: Classification of Model Transformation Approaches, 2nd OOPSLA Workshop on Generative Techniques in the context of MDA, 2003.
7. T. Gardner and 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
8. D. Hearnden, K. Raymond, J. Steel: Anti-Yacc: MOF-to-Text, EDOC 2002.
9. Eclipse Modeling Framework, http://www.eclipse.org/emf
10. Eclipse UML, http://www.omondo.com
11. 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.
12. A. Hamie: Towards Verifying Java Realizations of OCL-Constrained Design Models Using JML, 6th IASTED International Conference on Software Engineering and Applications, 2002.
13. Java Community Process: Java Metadata Interface (JMI) Specification, http://www.jcp.org, 2002.
14. A. Ledeczi & al.: The Generic Modeling Environment, Workshop on Intelligent Signal Processing, 2001.
15. MIA, http://www.model-in-action.fr
16. ModFact, http://modfact.lip6.fr
17. NetBeans Metadata Repository, http://mdr.netbeans.org
18. Objecteering, http://www.objecteering.com
19. OMG: Human-Usable Textual Notation (HUTN) Specification, document no: ptc/04-01-10, 2003.
20. OMG: MDA Guide Version 1.0.1, document no: omg/2003-06-01, 2003.

21. OMG: Meta Object Facility (MOF) Specification version 1.4, document no: formal/2002-04-03, 2002.
22. OMG: Request for Proposal MOF2.0 Query /Views /Transformations, document no: ad/2002-04-10, 2002.
23. OMG: Request for Proposal UML 2.0 OCL, document no: ad/2000-09-03, 2001.
24. OMG: UML 2.0 Superstructure Specification, document no: ptc/03-08-02, 2004.
25. OMG: Unified Modeling Language Specification version 1.4, document no: formal/01-09-67, 2001.
26. OMG: XML Metadata Interchange (XMI) Specification version 2.0, document no: formal/03-05-02, 2003.
27. I. Porres: M. Alanen, A Generic Deep Copy Algorithm for MOF-Based Models, Model Driven Architecture:Foundations and Applications, 2003.
28. Poseidon, http:// www.gentleware.com
29. Rational Rose, http://www.rational.com
30. D. Riehle & al.: The Architecture of a UML Virtual Machine, OOPSLA 2001.
31. UMT-QVT: http://umt-qvt.sourceforge.net
32. W3C: Web Services Architecture, http://www.w3.org/TR/ws-arch, 2004.
33. Workflow Management Coalition: Workflow Process Definition Language, document no: WFMC-TC-1025, version 1.0, 2002.