# Applying Model Fragment Copy-Restore to Build an Open and Distributed MDA Environment*

Prawee Sriplakich, Xavier Blanc, and Marie-Pierre Gervais

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

**Abstract.** ModelBus is a middleware system that offers the interoperability between CASE tools for supporting software development according to MDA. This interoperability allows tools to share services and models, by using an RPC mechanism. ModelBus adopts the call-by-copy-restore semantic, as it is very close to local call semantic and is flexible as regards tools' heterogeneous model representations. In this work, we extend this semantic to enable only specific model fragments to be passed as parameters, instead of complete models. The advantages are 1) improving the performance because passing only model fragments requires less data processing and 2) enhancing access control to models because the service's modification can be restricted to the specific model fragment that is specified as parameters. The implementation of this work is available as the Eclipse project Model Driven Development integration (MDDi).

## 1 Introduction

The Model Driven Architecture (MDA) [16] is a software development approach which focuses on models. In MDA, all software development artifacts are represented by models. Those models can be manipulated by a variety of CASE tools which offer automated operations on the models, such as model visualization, model edition, model transformation and model well-formed-ness checking.

In our previous research, we have proposed a middleware system supporting the interoperability between heterogeneous and distributed CASE tools to support MDA. This MDA environment, called ModelBus [2] [3] [15] [24], enables distributed and heterogeneous CASE tools to share their functionality and models. ModelBus achieves this interoperability by using the RPC paradigm, which enables tools to invoke each other's services and exchange models by parameter passing. Thus, in ModelBus, RPC parameters are models.

ModelBus supports the call-by-copy-restore semantic[1], which is very close to the semantic of the local procedure call. Our choice is motivated by two reasons. First,

* The work presented in this paper is supported by the project MODELWARE, co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2002-2006).

[1] ModelBus offers the call-by-copy semantic for IN and OUT parameters and the call-by-copy-restore semantic for INOUT parameters [15]; however, in this paper, we focus on call-by-copy-restore.

O. Nierstrasz et al. (Eds.): MoDELS 2006, LNCS 4199, pp. 631 – 645, 2006.
© Springer-Verlag Berlin Heidelberg 2006

several model manipulations such as in-place model transformation [22] and model refactoring [26] require the ability to modify models. To allow such model manipulations to be shared as services, ModelBus should not limit to read-only parameter passing but also enable tools to modify each other's models. Second, unlike the call-by-reference tool integration approach [9], our call-by-copy-restore approach avoids the complexity and cost of representing parameter values as distributed objects (e.g. CORBA, RMI).

The copy-restore mechanism of most RPC systems, such as NRMI [25], which is Java RMI-based, and Microsoft RPC, which implements the DCE RPC specification [20], transmits a *deep copy* of parameter values: the objects that a programmer specifies as parameters and all objects reachable from them are copied. In our context, parameters are models, which are graph data structures containing model elements and links between them. Hence, applying this deep-copy mechanism to a model will result in transmitting the entire model graph, which is inappropriate for the following reasons.

- *Performance*. A model can include more elements than required by the service. For instance, a UML [19] model can contain use case elements; class diagram elements, and sequence chart elements (with links between them). If the service does not use all these elements, transmitting the entire model graph will unnecessarily waste computing resources.

- *Access control.* The deep-copy mechanism offers too much access to the service: It enables the service to modify the entire model, i.e., all elements reachable from parameters values. Consequently, the caller can not protect parts of models from modification by the service.

Those reasons motivate us to propose a new parameter passing semantic that transmits and restores only a specific *model fragment* (i.e. a subgraph of a model). Compared to existing graph fragment transmission solutions, our approach offers the following novel features:

- *Flexible specification of model fragments*. The approaches based on the notion of object views (reduced objects) [5] [13] or on the Demeter graph traversal language [14] offer a way to specify graph fragments to be transmitted. However, their fragment specification is statically fixed in the service definition. Therefore, at runtime, it is not possible to change the fragment specification for each service call. On the other hand, our approach offers the flexibility to specify arbitrary fragments and to change them in each service call.

- *Access control in parameter passing*. Caching systems (e.g., CORBA caching [4], RMI caching [5]) enables graph elements (i.e. objects) to be transmitted only when requested (to avoid complete graph transmission). However, to our knowledge, few works offer means for limiting the model elements that a service is allowed to modify. I.e., if a service requests all elements in the graph, then it can modify the entire graph. On the other hand, our approach offers a mechanism to protect parts of models from modification.

- *Preserving tools' existing data structures*. To integrate existing CASE tools with caching systems, tool programmers would need to change the existing implementation of tools' data structures to the one supported by the caching systems (e.g. object stubs). Our approach is different as it requires no change to existing data structure implementation. For this reason, it has little impact on existing tools' implementation and facilitates their ad hoc integration.

This work has been implemented in ModelBus, which will be soon available as an Eclipse open source project Model Driven Development integration (MDDi, http://www.eclipse.org/mddi). It is built on top of the Web Services platform, which is widely used for integrating heterogeneous applications. While the Web Services protocol (SOAP/HTTP) only defines the RPC message format, ModelBus extends it by providing a parameter passing mechanism for transmitting model fragments and restoring the update made by the service to the original model.

This paper is organized as follows. Section 2 presents our research background on tool integration and explains why the call-by-copy-restore approach is chosen. Section 3 states the objectives and requirements of this work. Section 4 describes our solution and its rationales. Section 5 describes the implementation and performance result achieved by ModelBus. Related works are discussed in section 6, before conclusion.

## 2 Background: CASE Tool Integration with Call-by-Copy-Restore RPC

ModelBus deals with model exchange between tools via RPC. In this environment, we assume that models being manipulated by tools (both caller and service) are stored in the tools' memory, similarly to the way software generally manipulate data. When one tool invokes another tool's service, the callee tool needs means for accessing (reading/writing) models that are service parameters located in the caller tool's memory. To support this model access, RPC middleware needs to solve two complications:

- *Remote communication*. An open MDA environment should support the integration of tools executing in different machines, therefore middleware needs to handle data transfer between the caller and the service.
- *Heterogeneous model representations*. As each CASE tool can be implemented with different programming languages, their memory representation of models can be different (e.g. Java objects, C data structures). If the caller and callee tools use different model representations, the middleware needs to translate models from one representation to another.

We focus on RPC approaches that offer close semantic to local call as this can hind the complication of tool distribution. In our previous work [24], we have studied two main approaches: call-by-reference and call-by-copy-restore. The call-by-reference approach requires that models be represented as distributed objects so that the callee tool can read and modify the remote models by using callback mechanism. On the other hand, in the call-by-copy-restore approach, models are copied from the caller tool to the callee tool at the beginning of service invocation. At the end, the model is copied back to replace the original model at the caller tool.

For purpose of tool integration, we have chosen the call-by-copy-restore approach rather than call-by-reference. First, in call-by-reference, callback makes model access very costly. The study by Kono & al. [10] shows that, when more than 5% of objects are accessed, call-by-copy-restore has significantly better performance than call-by-reference.

Second, call-by-reference requires that models be represented as distributed objects. Existing tools that have not been planned for integration usually implement model representation with simple, local data structures. Consequently, to apply call-by-reference, their model representations would need to be changed to distributed objects. On the other hand, for call-by-copy-restore, the marshaling /unmarshaling mechanism of middleware can be extended to cope with any model representations. Hence, tool programmers do not need changing the existing model representations of tools for integrating them.

We identify two copy-restore RPC approaches. In the first approach, parameter restoration is done only at the end of service call. This is the case for NRMI and DCE RPC systems. In the second approach, systems offer a stronger guarantee: the parameter value copy at the service side is kept consistent with the original copy at call time (even after service call). This is the case for caching systems. In this work, we focus on the first approach (restoring at the end of service call). This is because we aim to preserve existing data representation of tools. The caching approach requires a mechanism for intercepting when data is modified so that it can restore the data. If this approach were used, we would face the difficulties in changing or adapting the existing data representation of tools to support this interception.

## 3 Model Fragment Copy-Restore: Objectives and Requirements

**Improving performance.** Despite the advantages of distributed tool integration, the RPC causes additional latency compared to local call. In fact, marshaling and unmarshaling complex, large data structures has been recognized as costing major latency in RPC (25%-50%) [21]. Hence, the larger models, the more latency for marshaling, transmitting and unmarshaling them. Moreover, if the callee tool does not entirely use the models, transmitting the entire model can waste the memory for storing unused fragments.

By passing only model fragments as parameters in service call, the amount of data to be processed is reduced. Therefore, it can significantly improve the performance especially if the model fragments are relatively small.

**Enhancing access control.** The access control problem has not been addressed yet in the RPC domain. Existing call-by-copy-restore middleware, such as NRMI and DEC RPC, enables a programmer to pass program pointers as service parameters. It considers that the service should have access to all objects reachable from those pointers. Therefore, the entire graph is transmitted to the service side and is entirely restored at the end of service call.

In MDA, a model can be built up from a large number of model elements, each of which describes a different software module or aspect. As those elements have relations with each other, they are parts of the same graph. According to the existing call-by-copy-restore semantic, passing a single element as a parameter enables a service to reach and modify the entire model. This approach can be dangerous because the service can modify model parts beyond the caller's intent.

This problem motivates us to integrate access control with parameter passing. The idea is to associate each parameter value with a model fragment (i.e. a subgraph) to restrict the service to access only elements in the fragment.

**Providing consistent restoration.** In the call-by-copy-restore mechanism, the modification that a service makes to the data's copy needs to be restored back to the caller side. Existing call-by-copy-restore systems (e.g. NRMI, DCE RPC) have already proposed a mechanism for complete graph restoration, which we will refer to as the "*basic*" mechanism. This mechanism consists in overwriting each graph element's content with an updated value. In the case of models, a model element's content is a set of properties, each of which contains either primitive data or references (pointers) to other model elements. Hence, to restore a model, this basic mechanism would overwrite all the property values of each model element.

In our context, the data that is transmitted to the service corresponds to a model fragment, which also has links with the rest of the model. The links between the model fragment and the rest of the model consist of *outbound links*, which are the references owned by the fragment's elements to elements outside the fragment and *inbound links*, from outside to the fragment. E.g., the complete model in fig. 1 contains the elements `{A, B, C…I}`; the specified fragment is `{F, G, H, I}`; and the outbound and inbound links are `{F→A, G→C}` and `{E→H, D→I}` respectively. These outbound and inbound links exist at the caller side but not at the service side (since one of their ends does not exist). The basic restoration mechanism (i.e. for restoring a complete graph) is not aware of this fact. Therefore, it needs to be extended or modified to deal properly with those links as follows.

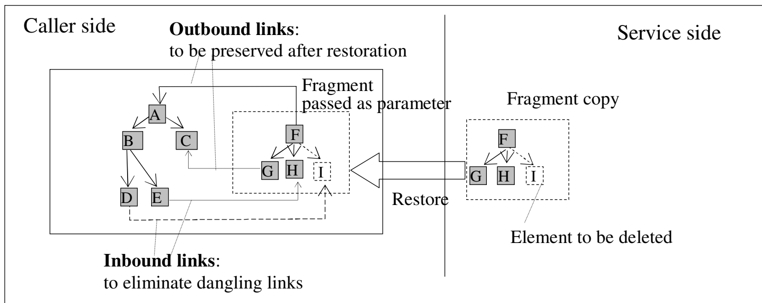


**Fig. 1.** Consistent restoration of a model fragment

- *Preserving outbound links*. According to the basic mechanism, overwriting the properties of the caller side's elements with the properties of the service side's elements would make the outbound links lost, e.g., in fig. 1, `{F→A, G→C}` would be lost after restoration. For this reason, the restoration mechanism for a model fragment needs to recognize outbound links and preserve them.

- *Supporting consistent element deletion*. Service logics may require the deletion of elements passed as parameters. We observe that few call-by-copy-restore systems enable the service to explicitly delete elements: most systems only enable a service to do so implicitly by making elements unreferenced (to be garbage-collected). Those systems make element deletion difficult because 1) the service needs to search for references to be eliminated, and 2) if the model is not entirely transmitted and there exist inbound links to some elements, then the deletion of those elements will be

impossible, e.g., in fig. 1, the inbound link `D→I` prevents the deletion of `I`, despite the service's intent. This motivates us towards the explicit deletion approach, in which the service can specify elements to be deleted. In this new approach, the restoration mechanism needs extension to support the elimination of *dangling inbound links*, which reference deleted elements.

# 4 Design in ModelBus

Similarly to other call-by-copy-restore systems, ModelBus offers the transparent management of partial parameter passing through *client* and *server stub* components. The stubs offer programming interfaces enabling a tool programmer to write service call code and service implementation code. A new aspect is that these interfaces are extended in order that the programmer can specify a model fragment to be passed as parameters and the stub implementation takes into account model fragment specification when marshaling and restoring parameters.

## 4.1 Enabling Model Fragment Specification Through Stub Interface

A stub interface generated by most RPC systems (e.g. RMI, CORBA) enables a programmer to specify complex-structured parameter values (i.e. graphs) with program pointers. When these pointers are passed to the service, the middleware will create a copy of the pointed data structure at the service side and create new pointers pointing to the copied data.

ModelBus offers a new way of generating stub interfaces to add an extra parameter, called *scope*, which enable a programmer to specify a model fragment to be transmitted. A scope is a subset of model elements selected from a complete model. Independent from programming languages and regardless the model representation used, a scope is a set of references to the objects representing model elements. This scope parameter can be mapped to any programming languages using their native types that can represent a set of object references, e.g., in Java, it can be mapped to `java.lang.Collection`. To define a model fragment using this scope parameter, a programmer instantiates a set and adds model element references to this set.

This approach is flexible, as it enables the specification of any arbitrary model fragments; however, having to add each model element individually to the collection may be cumbersome. For this reason, we also provide a helper operation enabling a programmer to easily add a group of hierarchical elements. The helper operation `addWithChildren(Collection scope, Element e)` recursively adds the element `e` and also its child elements to the scope. It exploits the aggregation relations defined in metamodels for identifying models' hierarchical structure. An example use of this operation is to add a UML package and all its content (the classes in this package, the classes' features …) to a scope.

The operation in both client stub and server stub's interfaces has the `scope` parameter. In the client stub interface, the `scope` parameter enables a client program to specify the model fragment to which the service has access. In the server stub interface, this parameter enables the service program to specify the model fragment that is the result of service execution. It contains the model elements to be transmitted back to the client for restoration, which include both the elements previously received from

the client (which can be modified by the service) and new elements produced by the service. Moreover, the service program can explicitly delete existing elements by excluding them from the scope collection.

**Stub generation.** The stub interfaces can be generated from the service description. ModelBus provides a service description language dedicated to the modeling domain. In this language, service description is defined independently from service implementation and the model representation used by the service. It uses Meta Object Facility (MOF) for defining the structures of models that are services parameters. More precisely, service parameters are typed by metaclasses (MOF classes). At the implementation level, the metaclasses are mapped to concrete data representations that the caller and callee tools use for manipulating models (e.g. Java classes, C structure types). The stub generation can be extended to support any model representations used by tools (e.g. Java Metadata Interface (JMI) [8] and Eclipse Modeling Framework (EMF) [6]). ModelBus enables a programmer to choose a model representation used by his tool for generating the corresponding stub interfaces.

**Example.** We illustrate an example service and its stub interfaces. The `moveClass` service enables a developer to modify his UML model by moving a UML class from one package to another. To use this service, he needs to specify two parameters: a class to be moved and the target package to which this class will be moved. Therefore, the abstract definition of this service is `moveClass(inout c:Class, inout targetPackage:Package)`, where `Class` and `Package` are metaclasses of the UML metamodel.

If we used existing middleware to generate stub interfaces for Java, we would obtain the method `void moveClass(uml.Class c, uml.Package targetPackage)`, supposing that Java classes `uml.Class` and `uml.Package` concretely represent Class and Package model element types. This method offers no means for the client program to specify the model fragment to be passed as parameters; therefore, the middleware will entirely marshal the UML model. On the other hand, with ModelBus, the generated stub will offer the method with the scope parameter: `void moveClass(Collection scope, uml.Class c, uml.Package targetPackage)`. This method enables the client program to specify a specific model fragment relevant to the service. For example, if a developer wants to move a class C1 from a package P1 to P2, then this service needs to modify only C1, P1, and P2, i.e. it removes the containment link between C1 and P1 and it creates a new containment link between C1 and P2. As other model elements do not concern the service, the programmer can optimize the service call by specifying the scope to be only these three elements.

**Rationale.** This new stub interface is motivated by the following reasons.

- *Flexible specification of model fragments*. Representing a model fragment as a collection offers the full flexibility to programmers. It enables the caller to define fragments arbitrarily and to change the fragment definition in each service call, i.e. the members of the scope collection can be selected dynamically. Therefore, this approach can accommodate the different needs of tools.

- *Small change to original service signatures*. We only add one extra parameter to stub interfaces, while the other parameters remained unchanged. Therefore, the effort

of adapting our solution to existing RPC application only consists in adding the code for specifying the scope's value, while the existing code remain unchanged.

### 4.2 Model Fragment Marshaling

The stubs offer a marshaling mechanism enabling the transmission of the model fragment specified by the scope parameter from the caller to the service and also from the service back to the caller. This mechanism is different from one used by existing RPC systems as it deals with an incomplete graph transmission. Only the elements that are included in the scope are serialized and the elements outside the scope are not serialized, even if they are linked with elements in the scope.

Marshaling a model element consists in writing its properties' values. These values are either primitive data or references to other elements (e.g., a UML package element contains the property `name`, which is primitive data and the property `ownedMember`, which contains references to other model elements owned by this package). Contrary to the complete model marshaling mechanism, which serializes all the property values, the model fragment marshaling mechanism must avoid marshaling the references to elements outside the scope (i.e. outbound links, see fig. 1.), because those references will become dangling when transmitted to the other side. The code at line 7 serializes only intra-fragment links. This mechanism is written in pseudo code as follows.

```
1. serializeModelFragment(Collection scope, OutputStream out) {
2. for each Element e in  scope {
3.  for each Property p in getProperties(e) {
4.   Object v = getPropertyValue(e, p);
5.    if(isPrimitiveData(v))  out.writePrimitive(v);
6.    else for each Reference r in v
7.     if(scope.contains(r)) out.writeLink(r)           } } }
```

In our approach, first the model fragment specified by the scope is marshaled, and then the service parameters are marshaled as pointers to the previously marshaled elements. At the receiver side, first unmarshaling the scope produces model elements in memory, and then unmarshaling the service parameters produces the pointers to those model elements. This approach avoids duplicate model transmission when multiple parameter values reference the nodes of the same graph. In this case, only one graph copy is created at the service side and the transmitted parameter values will point to the nodes in this copy, which results in the identical structure to the one at the caller side.

**Example.** We continue with the `moveClass` example from 4.1. By using the `serializeModelFragment` mechanism, the specified elements (`C1, P1, P2`) can be transmitted without other surrounding elements. As shown in fig. 2 (a, b), even though packages `P1` and `P2` contain classes `C2` and `C3`, those classes will not be transmitted. This example also shows the pointer transmission for service parameters (`c`, `targetPackage`), which enables the callee tool obtains identical pointers to the ones at the caller side.

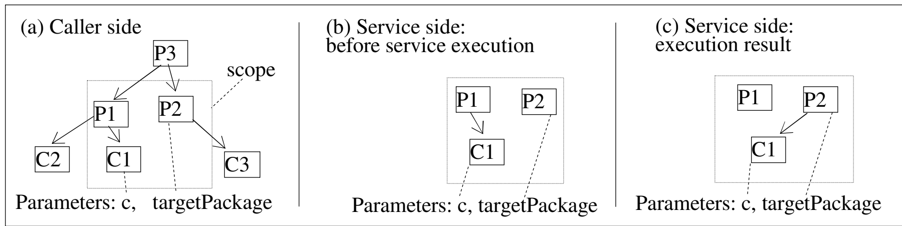


**Fig. 2.** Model fragment serialization

**Rationale.** This approach obviously improves performance: the amount of data to be serialized/ deserialized and transmitted is reduced proportionally to the scope's size. Moreover, we choose to transmit all the elements in the scope at a time, instead of transmitting elements on demand to reduce the complication of callback. As the scope is specified at application level, where the knowledge of service logics is available, we assume that the high portion of the elements in the scope will be used by the service. In this case, this approach is more optimal than on-demand transmission.

As regards access control, our approach protects the service from modifying elements outside the scope, since those elements are not transmitted to the service.

### 4.3 Model Fragment Restoration

As described in 4.1., the service program can access to the `scope` parameter, to specify the model fragment that are the result. This scope initially contains model elements transmitted from the caller. The service can modify the content of those elements, i.e. modify their property values. As an element can contain not only primitive data but also references to other elements, the service can also add/remove links between elements.

Moreover, the service can add/ remove model elements to/from the scope collection. Adding elements to the scope enables the service to transmit back the new elements that do not exist at the caller side. Removing elements from the scope will result in deleting those elements at the caller side.

At the end of service invocation, the server stub transmits the scope back to the client and the client stub overwrites the *original fragment* with the *received fragment*. The restoration consists in 1) adding new elements to the caller side's scope, 2) updating the existing model elements' content, and 3) deleting the model elements correspondingly to the deletion at the service side. In this work, we offer the following extensions to the "basic" restoration mechanism (cf. section 3).

- *Preserving outbound links*. In the basic mechanism, the content of each original element is replaced by the content of received element. This mechanism preserves the inbound links (because the original elements preserve their identity; hence, the links to them remain valid). However, this mechanism makes outbound links lost; therefore, we propose the `updateLink` operation for updating intra-fragment links while preserving outbound links, cf. following code. This operation is applied to two corresponding elements: one in the original fragment and the other in the received fragment. It updates a property whose value is a set of model element references. It has

two parameters: `originalProp` is the original element's property value to be updated and `newProp` is the received element's property value. The algorithm begins by removing all the intra-fragment links in `originalProp` while preserving outbound links (lines 2-3). Then, the links in `newProp` are copied to `originalProp` (lines 4-5).

```
1. updateLink(ReferenceSet orginalProp, ReferenceSet newProp) {
2.  for each Reference r in originalProp
3.   if( scope.contains(r) ) originalProp.remove(r);
4.  for each Reference r in newProp
5.   originalProp.add( getCorrespondingElementOf(r) );        }
```

- *Supporting consistent element deletion.* Our approach enables the service to delete elements simply by excluding them from the scope. To apply the deletion, the caller stub searches and deletes dangling inbound links. To optimize search performance, the search space is reduced by exploiting metamodel information. In fact, potential elements that can contain dangling inbound links are the elements that have properties typed by the metaclasses of the deleted elements; therefore, we can filter out non-potential elements without examining their contents. Moreover, for the potential elements found, only their specific properties are examined, instead of examining their whole content.

**Example.** Fig. 2(c) shows the model fragment at the service side to be propagated back. According to the UML class diagram structure, a package element has the property `ownedMember`, which refers to the package's elements, i.e. its value is a set of element references. To restore this property is not to simply overwrite the property value of the client side's element with the one of the service side's element; otherwise, the outbound links (`P1→C2, P2→C3`) would be lost. We have proposed the `updateLink` operation to restore the property value correctly.

We illustrate an element deletion example with fig. 1. In this example, the service explicitly removes element `I` by excluding it from the scope; hence, the transmitted-back fragment will not contain `I`. This enables the caller stub to detect element deletion so that it can search and eliminate dangling inbound links.

**Rationale.** Our restoration mechanism satisfies the objectives of enhancing access control and preserving the entire model's consistency. It protects elements outside the scope from modification and it properly manages the inbound and outbound links for integrating the update to the entire model.

## 5 Implementation and Performance Results

**Implementation.** This work has been implemented in ModelBus, a middleware system for CASE tool integration. ModelBus' tool integration method has already been described in both research papers [2] [3] and in a ModelWare project deliverable [15]. This method is similar to the one of existing RPC middleware. First, ModelBus provides the service description language, which enables heterogeneous tools' services to be uniformly defined. Our service description language is different from others in that it uses MOF metamodels for defining service parameters; hence the model structures of services' input/output are clearly identified in a standard way. Second, ModelBus

provides the stub generation for generating client and server stubs, which implement our model fragment copy-restore mechanism. Currently, ModelBus only offers Java stub generation; however, the proposed copy-restore mechanism is language independent.

The stubs communicate with the SOAP/HTTP protocol. This choice is motivated by two reasons. First, it is programming-language independent. Second, it is compatible with the XML Metadata Interchange (XMI) standard [18]: models encoded with XMI can be easily put inside SOAP messages.

**Empirical performance results.** We report the performance of ModelBus in two aspects. First, we show that our approach enhances the scalability in service invocation performance: Even when the size of the complete model increases, the user can obtain the constant performance of service invocation by limiting the size of fragments to be passed as service parameters.

We set up the experiment as follows. We generate UML models with different sizes (from 2,000 to 100,000 model elements). Each model contains UML classes organized in an arbitrary package hierarchy, similar to usual UML models in software development. Our benchmark program invokes a service (that has one parameter) with different model fragment sizes extracted from those generated models (10, 50, 100, 500, 1000 elements). The cost measured by the benchmark tool is the total cost of all activities in service invocation, except the execution of the service logic (i.e. serialization/deserialization, data transmission through LAN, and restoration).

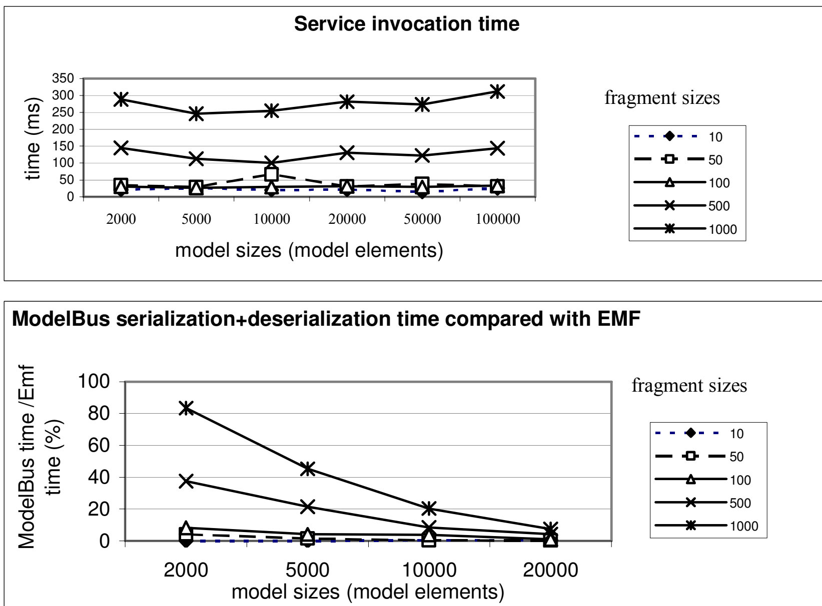


**Fig. 3.** ModelBus' Performance in service invocation and model serialization/ deserialization

As illustrated with the result in fig. 3(top), our approach enables the user to work with very large models. For example, by fixing a constant fragment size of 500 model elements, the service invocation costs around 125 ms, regardless the size of the complete model. Please note that the illustrated performance is relative to the performance of machine, network, model encoding method and RPC protocols. In this work, we encode model with the standard XMI format and invoke service with SOAP/HTTP. This choice offers interoperability at the cost of XML processing.

As for the second aspect, we compare the performance of ModelBus with Eclipse Modeling Framework (EMF), a toolkit that is optimized for performance [7]. In this case, we compare only the performance of model serialization/ deserialization (as EMF does not offer an RPC mechanism). We observe that when the models become large, the EMF performance decreases rapidly (40 ms for 2,000 elements vs. 1.5 s for 50,000 elements). Moreover, when models are very large (100,000 elements in a machine with 1 GB of memory), EMF generates an out-of-memory error. In our approach, the user can avoid this problem by limiting the size of model fragment. Fig. 3(bottom) shows the percentage of the serialization and deserialization time of ModelBus compared to EMF. It shows that this percentage is close to zero when the model is larger than 20,000 elements, i.e., EMF becomes significantly slow.

## 6 Related Works

**Object views.** Eberhard [5] Lipkind [13] propose the way to transmit graph fragments. In their approach, graph fragments are defined with object views. An object view is derived from a class (i.e. data type) but contains only a subset of the class' properties. Since properties can represent links, the object view can define a subgraph including only elements to which the object view's properties link. E.g., given an object view `v1` that excludes the property `prop1`, its corresponding fragment will exclude elements to which `prop1` links. Compare to this approach, ModelBus offers a more flexible way of specifying model fragments as follows.

- *Dynamic model fragment specification*. Object views are specified statically at the service signature level, i.e., as the types of service parameters. Therefore, it is not possible to change, for each service call, the structure of the fragment to be transmitted. For example, if a service parameter is typed by object view `v1` (previously defined), then elements to which `prop1` links will never be transmitted. On the other hand, in our approach, a subgraph is represented by a scope (a collection), which can be specified differently in each service call.

- *Arbitrary model fragment specification*. With object view, a programmer can choose either to transmit all elements to which a property links, or not to transmit them at all. For example, given that a package has the property `ownedMember`, the programmer can either include or exclude all elements owned by this package. Our approach gives the freedom to programmers to define an arbitrary fragment, e.g. a package with a subset of its owned elements.

**Adaptive Parameter Passing.** Lopes [14] proposes a parameter passing mechanism that avoids the transmission of entire graphs. The expression of subgraphs to be transmitted is based on the Demeter graph traversal language [12]. It expresses a

traversal from a specified element to visit elements reachable from it. I.e., this traversal contains a set of selected paths from this element to some other elements. This approach considers that all elements in those paths will be included in the subgraph. We illustrate an example of expressing a UML model fragment. The expression "***from*** *Package* ***through*** *ownedMember* ***to*** *Class*" expresses all paths from a `Package` element to `Class` elements that include at least once the edge `ownedMember`.

This approach has a similar limitation to the object view approach. Demeter expressions are statically defined at the service signature level (as the types of service parameters). For example, let a model consists of a UML package containing N classes. Applying the previous expression example to this package always yield the same subgraph. The caller can not specify a different subgraph for each service call. Moreover, the caller can not specify an arbitrary fragment, such as, a subgraph containing this package and a subset of its own classes. The subgraph will always contain all the owned classes.

## 7 Conclusion and Future Works

In this work, we propose a new parameter passing semantic for transmitting only fragments of models. This parameter passing offers the advantages of improving performance and enhancing access control to models. Our approach enables a programmer to define a scope of service parameters, so that the middleware can transmit and restore the model fragment specified by this scope.

Even though we focus on models in this paper, our mechanism is also applicable in general-domain applications. In fact, metamodels are similar to class diagrams, which define abstractly data in any application domain, and models can be manipulated by any programming languages; therefore, our approach can be used for integrating heterogeneous applications in any domain, provided that they share the same abstract data structures.

Our parameter passing approach has been implemented in ModelBus, which is available as an Eclipse open source project MDDi. The development of ModelBus is supported by the IST project ModelWare, which aims at promoting the successful application of the MDA approach. Currently, we are applying the ModelBus concepts for integrating industrial and research tools provided by the project partners, such as Objecteering (http://www.objecteering.com), Open Source Library for OCL (OSLO, http://oslo-project.berlios.de), and ATL model transformation engine [1].

For future works, we aim to extend our approach to overcome the following limitations. First, in this approach, the caller must have the knowledge of what model elements the service needs and must specify them in the scope parameter. For future works, we aim to relieve this complication from the caller by proposing an alterative approach that exploits the knowledge of the service about what model elements it needs. Our goal is to provide the service signature that can define the model elements that the service needs. This signature can be exploited by the caller stub to identify the model fragment to be passed to the service. Consequently, the caller can call the service without having to specify the scope itself.

Second, in this work, we do not take into account the concurrency of model modifications. We assume here that the caller tool is blocked during the service invocation

to avoid that the caller and callee concurrently update the model, or that the caller concurrently apply another service that will update the same model. Our recent work to support concurrent model update [23] addresses the problem of how concurrent modifications made by different tools on the same model can be unified. For future work, we aim to combine the aspect of model fragment with the aspect of model update concurrency. More precisely, we aim to enable each tool to make a different model fragment corresponding to what it needs. The fragment of one tool can overlap with the ones of others, and those tools are allowed to concurrently modify their fragment. We would like to study how to unify the concurrent modifications made to those overlapping fragments.

## References


1. Bézivin, J., Hammoudi, S., Lopes, D., Jouault, F., Applying MDA Approach for Web Service Platform, *Proc. of the 8th Int'l IEEE Enterprise Distributed Object Computing Conf. (EDOC)*, 2004.
2. Blanc, X., Gervais, M-P., Sriplakich, P., Model Bus: Towards the Interoperability of Modeling Tools, *Proc. of the European MDA Workshop: Foundations and Applications (MDAFA 2004)*, LNCS 3599, Springer, 2005.
3. Blanc, X., Gervais, M.-P., Sriplakich, P., Modeling Services and Web Services: Application of ModelBus, *Proc. of the Int'l Conf. on Software Engineering Research and Practice (SERP)*, 2005.
4. Chockler, V.G., Dolev, D., Friedman, R., Vitenberg, R., Implement a Caching Service for Distributed CORBA objects, *Proc. of the IFIP/ACM Int'l Conf. on Distributed Systems Platforms (Middleware)*, 2000.
5. Eberhard, J., Tripathi, A., Efficient Object Caching for Distributed Java RMI Applications, *Proc. of the IFIP/ACM Int'l Conf. on Distributed Systems Platforms (Middleware)*, 2001.
6. Eclipse, *Eclipse Modeling Framework (EMF)*, http://www.eclipse.org/emf
7. Eclipse, EMF Performance: EMF 2.0.1 vs. EMF 2.1.0 RC1, http://www.eclipse.org/emf
8. Java Community Process, *Java Metadata Interface (JMI) Specification version 1.0*, http://www.jcp.org, 2002.
9. Kath, O. et al., An Open Modeling Infrastructure integrating EDOC and CCM, *Proc. of the 7th IEEE Int'l Enterprise Distributed Object Computing Conf. (EDOC)*, 2003.
10. Kono, K., Kato, K., Masuda, T., Smart Remote Procedure Calls: Transparent Treatment of Remote Pointers, In *Proc. of the 14th Int'l Conf. on Distributed Computing Systems (ICDCS)*, 1994.
11. Krishnaswamy, V., Walther, D., Bhola, S., Efficient Implementation of Java Remote Method Invocation (RMI), *Proc. of the 4th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS)*, 1998.
12. Lieberherr K. J., Silva-Lepe, I., Xiao, C., Adaptive object-oriented programming using graph-based customization, *Comm. of ACM*, 37(5), May 1994.
13. Lipkind, I., Pechtchanski, I., and Karamcheti, V., Object views: Language support for intelligent object caching in parallel and distributed computations, *Proc. of the 14th ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1999.
14. Lopes, C. V., Adaptive Parameter Passing, *Proc. of the 2nd JSSST Int'l Symposium on Object Technologies for Advanced Software (ISOTAS)*, LNCS 1049, Springer, 1996.
15. *ModelBus: Functional & Technical architecture document (Vol II)*, ModelWare project deliverable D3.1, http://www.modelware-ist.org, May 2005.
16. OMG, *MDA Guide Version 1.0.1*, document no: omg/2003-06-01, 2003.

17. OMG, *Meta Object Facility version 2.0*, document no: formal/06-01-01, 2006.
18. OMG, XML Metadata Interchange (XMI) Specification version 2.0, document no: formal/03-05-02, 2003.
19. OMG, *UML 2.0 Superstructure Specification*, document no: formal/05-07-04, 2005.
20. The Open Group, *DCE 1.1 RPC Specification*, http://www.opengroup.org, 1997.
21. Philippsen, M., Haumacher, B., More Efficient Object Serialization, *Proc. of the ACM 1999 Java Grande Conf.*, June 1999.
22. Porres, I., Model Refactorings as Rule-Based Update Transformations, *Proc. of the 6th Int'l Conf. on the Unified Modeling Language*, 2003.
23. Sriplakich, P., Blanc, X., Gervais, M-P., Supporting Collaborative Development in an Open MDA Environment, *Proc. of the 22nd IEEE Int'l Con. on Software Maintenance (ICSM)*, 2006.
24. Sriplakich, P., Blanc, X., Gervais, M-P., Supporting transparent model update in distributed CASE tool integration, *Proc. of the 21st ACM Symposium on Applied Computing*, 2006.
25. Tilevich, E., Y. Smaragdakis, NRMI: Natural and Efficient Middleware, *Proc. of the 23rd Int'l Conf. on Distributed Computing Systems (ICDCS)*, 2003.
26. Tokuda, L., and Batory, D., Evolving Object-Oriented Designs with Refactorings, *Proc. of the 14th IEEE Int'l Conf. on Automated Software Engineering (ASE)*, 1999.