Supporting Transparent Model Update in Distributed CASE Tool Integration[1]

Prawee Sriplakich, Xavier Blanc, 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

Model Driven Architecture (MDA) is a software development approach that focuses on models. In order to support MDA, a lot of CASE tools have emerged; each of them provides a different set of modeling services (operations for automating model manipulation). We have proposed an open environment called ModelBus, which enables the integration of heterogeneous and distributed CASE tools. ModelBus enables tools to invoke the modeling services provided by other tools. In this paper, we focus on supporting a particular kind of modeling services: services that update models (i.e. they have inout parameters). Our contribution is to enable a tool to update models owned by another tool. We propose a parameter passing mechanism that hides the complexity of model update from tools. First, it enables a tool to update models transparently to heterogeneous model representations. Second, it enables a tool to update models located in the memory of another remote tool transparently, as if the models were local. Third, it ensures the integrity between the updated models and the tool that owns the models.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments – *Integrated Environments*; D.2.12 [**Software Engineering**]: Interoperability.

Keywords

Middleware, CASE Tool, Interoperability, Integration, Graph, Data Structure, RPC, Call-by-copy-restore

1. INTRODUCTION

Model Driven Architecture (MDA) [25] is a software development approach that focuses on models. It aims at automating steps in software development, in order to reduce development cost and to increase software quality. This automation is realized by a set of modeling services – operations for automating model manipulation such as model storage [6], model edition [9], model transformation [11] [7], model verification [28] and model execution [19].

Therefore, an environment supporting MDA software development must offer modeling services to software developers. It must also ensure the interoperability between them [30]. Such environments have already been implemented in a form of CASE (Computer-Aided Software Engineering) tools, for instance, Rational Software Architect [16] and Objecteering [29]. However, existing environments only provide a limited set of modeling services. Consequently, the users of one environment (i.e. software developers) can meet the interoperability problem in using the modeling services offered by other environments. An open environment for integrating CASE tools is a solution to this problem. The goal is to enable users to take advantages of all modeling services provided by different CASE tools. Those CASE tools can be heterogeneous in terms of implementation technologies and can be executed in different locations/ operating systems.

Our previous works [4] [5] has contributions in realizing such an open environment, called ModelBus. ModelBus uses Remote Procedure Call (RPC) techniques to realize the communication between heterogeneous, distributed CASE tools. It enhances basic RPC by providing supports for model representation and model transmission. ModelBus enables a tool to invoke a modeling service that is provided by another tool. For example, it enables a UML editor to invoke the model transformation service provided by a model transformation engine [7]. ModelBus is available in the Eclipse open source project Model Driven Development integration (MDDi, http://www.eclipse.org/mddi/). It also has an important role in the MODELWARE project (http://www.modelware-ist.org) for integrating academic and industrial CASE tools.

Parameter passing mechanism is an important part of ModelBus: It enables the exchange of models between the *caller tool* (which invokes a modeling service) and the *callee tool* (which provides a modeling service). This mechanism must support three kinds of modeling service parameters: input, output and inout. An input is a model the caller sends to the callee. An output is a model returned from the callee as a result of the service invocation. An inout is a model that the caller sends to the callee and that is expected to be updated by the callee. This definition is well-

SAC'06, April, 23-27, 2006, Dijon, France.

[1] 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).

known. The new aspect is that we apply the notion of services to software development tools.

Prior to this work, ModelBus only supported input and output parameter passing. To complete this functionally, we propose a solution for inout parameter passing in this paper. Our contribution is to enable a CASE tool to update models owned by another CASE tool. This need is made evident by the emergence of several modeling services, such as in-place model transformation [11], model refactoring [24], and application of design patterns [13], which need the ability to update models passed as parameters.

The paper is organized as follows. Section 2 briefly presents the background about tools and modeling services to facilitate the further explanation. Section 3 presents our design objectives for enabling model update in tool integration. Then we illustrate those objectives through a concrete example. Section 4 presents the parameter passing approach in general and discusses the problems for applying those approaches to realize model update. Section 5 explains our solution and explains why it meets the objectives. Section 6 shows our experience in using the proposed solution. Section 7 discusses about related works. The last section concludes this work and discuses about future works.

2. BACKGROUND: MODEL, MODELING SERVICE, AND MODEL UPDATE

A model is a set of model elements. For instance, in a UML class diagram, each UML class, attributes, associations are model elements. Each model element is an instance of a metaclass (defined in the Meta Object Facility standard [26]). It can have properties conforming to the metaclass (e.g. a UML attribute has properties name, visibility, multiplicity etc.). The model elements can be linked together according to the associations between metaclasses.

A **modeling service** is an operation that has models as parameters. A parameter of modeling services can be input, output or inout. In this paper, we focus on a particular kind of modeling services: modeling services that have inout parameters (e.g. in-place model transformation [11], model refactoring [23], and application of design patterns [13]).

We present here an example of a modeling service: pullupMembers(inout ClassDiagramModel). This service has one inout parameter which is a UML class diagram. It serves for improving a structure of a class diagram. It updates the class diagram. The update is performed as follows: It examines the class hierarchy in the class diagram. If it detects two conditions: 1) there are two classes C1, C2 sharing the same superclass C3 and 2) C1 and C2 have equivalent attributes (same name and same type). If such conditions are detected, it removes such attributes from C1 and C2 and creates equivalent attributes at C3 instead. This service is inspired by the work in [31]. Figure 1 illustrates an example of the application of this modeling service. The left side is the model before service application and the right side is the result. Please note that the purpose of this example is to illustrate that model update can be realized with the parameter passing of modeling services. This paper does not focus on the logics inside the modeling services (e.g. techniques of model refactoring).

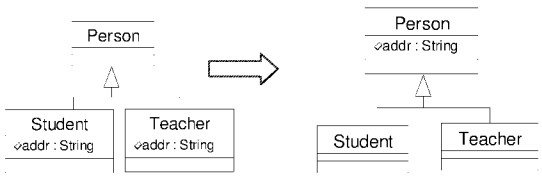


Figure 1. pullupMembers modeling service

Model update is the action of modifying the content of a model. Model update can be realized with any techniques such as model manipulation with API [26], in-place model transformation [11] and graph transformation [2]. By using whatever model update techniques, the result is equivalent to applying the following elementary actions to the model:

- *Model element creation/removal*: Creating/ removing a model element in the model.
- *Property update*: Setting a property value of a model element
- *Link update*: Creating/ removing a link between two model elements.

These actions can be identified from the example in figure 1 as follows. *Model element creation/removal*: creating a new attribute for Person, removing attributes addr of Student and Teacher. *Property update*: setting the name of the new attribute of Person (addr). *Link update*: linking the new attribute with Person (meaning that the attribute is owned by the class).

3. MODEL UPDATE IN TOOL INTEGRATION

3.1 Objectives

Our main goal is to enable a tool to 'update' a model 'owned by' another tool. 'Owned by' means that models are stored in the tool's memory so that the tool can manipulate them. For example, a UML editor stores a model in the memory in order to visualize it. Therefore, our main goal aims to enable a tool to change the memory content of another tool where the model is stored.

We aim to integrate distributed, heterogeneous tools. This means that tools can execute in heterogeneous machines. Each tool has its own memory space and it is free to choose any memory representation for storing models (e.g. Java objects, C++ objects). In order to update a model of another tool, the updater tool needs 1) remote communication enabling it to access the remote model and 2) the adaptation of memory representation enabling it to understand the model structure in the memory of the remote tool. For the facility of tool integration, these functionalities should be provided at middleware level. In other words, middleware should enable the tool to update the remote model as if the model were local.

Figure 2 illustrates an invocation of a modeling service having an inout parameter. In order to perform the service, the callee tool needs to update the model (passed as parameter), which is located at the caller tool's memory. However, the model in the caller tool is represented as Java objects while the callee tool program view models as C structured data. Therefore, the middleware is required for providing the remote communication and adapting different memory representations used in the two tools.

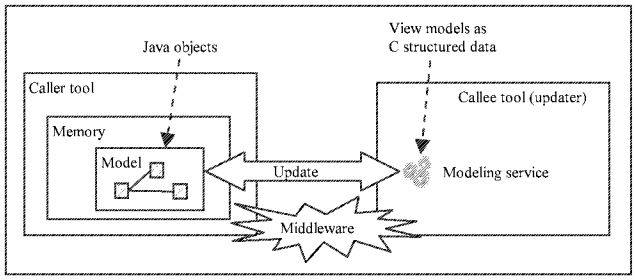


Figure 2. Model update in tool integration

We have the following objectives in the designs of this middleware:

Supporting heterogeneous memory representation. As heterogeneous tools are implemented with different technologies, the memory representations they manipulate can be different. In order to support this heterogeneity, the middleware must enable tools to update models regardless the memory representation of the models.

Preserving pointers to model elements. In order to manipulate models, a tool program needs to have pointers to the model elements in the memory. These pointers enable the tool to, for example, read the properties of each model element.

After the model update performed by the callee tool, it is desirable that the pointers to model elements in the caller tool program remain valid (except pointers to removed elements). Otherwise, the caller tool will lose access the model elements that were previously pointed by those pointers.

Notifying of model element creation/removal. During model update, the callee tool can create or remove model elements. This action causes changes in the memory of the caller tool (i.e. allocating an address for a newly created element, or freeing the address of a removed element). To enable the caller tool adapt to such changes, middleware must notify it. This notification allows the caller tool to adapt itself as follows.

- Creation case: the caller tool needs to know the memory address where the new element is created in order to gain access to this new element.
- Removal case: the caller tool needs to eliminate all the pointers to the memory address of the removed element in order to avoid accessing it (as this element does not exist anymore).

3.2 Illustration in an Example

We present an example of tool integration that requires model update. Two CASE tool are integrated, a UML editor and a model transformation (MT) engine. The UML editor allows the users to create UML diagrams. The MT engine offers a set of model transformation services including pullupMembers. The objective of this integration is to enable the user to apply pullupMembers to the model elaborated inside the UML editor.

Based on our objectives, the integration is expected to produce the following results.

Supporting heterogeneous memory representation. The fact that two tools uses different memory representations (the UML editor uses Eclipse Modeling Framework: EMF [12], while the MT engine uses Java Metadata Interface: JMI [17]) will not prevents the MT engine to update the model owned by the UML editor.

Preserving pointers to model elements. Suppose that the user elaborates the class diagram in figure 1 (left) on the UML editor. Therefore, the UML editor's memory contains a set of Java objects corresponding to the model elements. The UML editor has pointers to those model elements as the model is "in use", c.f. figure 3 (left).

Then the user applies pullupMembers. As a result, the model is updated. The pointer preservation keeps the updated model elements accessible by the UML editor. In the updated model showed in figure 3 (right), the pointers to UML classes Person, Student, Teacher, and UML data type String still remain valid.

Notifying of model element creation/removal. As the result of the pullupMembers application, the attributes named addr of the class Student and Teacher (denoted by addr1, addr2) are removed. The equivalent attribute (denoted by addr3) is created for the class Person. The notification of these changes enables the UML editor to gain access to addr3, and avoid accessing addr1, addr2. In figure 3 (right), thanks to the notification, the UML editor creates a pointer to addr3, removes the pointers to add1, addr2 and frees the memory associated to add1, addr2.

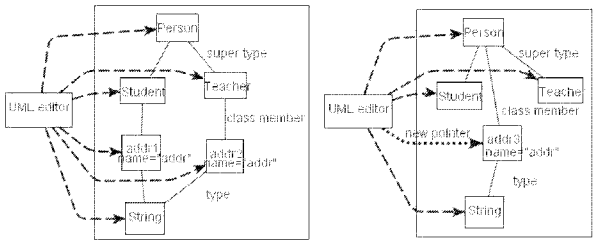


Figure 3. Original model and updated model

4. PARAMETER PASSING AND MODELING SERVICES

Two basic approaches for parameter passing are pass-by-reference and pass-by-copy [23]. We discuss them in the context of **inout** parameter passing.

Pass-by-reference. In this approach, distributed objects (i.e. stubs and servants) are used to represent parameter values. The servants hold the real data (parameter values) while the stubs serve for delegating data access to the servants. More precisely, the callee accesses the content of parameter values by invoking local operations on the stubs. Behind the scene, the stubs perform the remote communication (*callback*) to the servants in order to access the remote memory content.

Pass-by-copy. In this approach, a copy of a parameter value is made at the callee side, so the callee can update the parameter value in this copy. At the end of the invocation, the updated copy is sent back to the caller as a new value. Therefore, after the invocation, the caller has two versions of the parameter value, the original one and the updated one.

The previously presented approaches can be applied to models as follows:

Pass-by-reference. In this approach, models are represented as a graph of distributed objects, which offer remote access to tools. Model repositories [3] [10] have been implemented for enabling tools to represent models as distributed CORBA objects. By using such repositories, a caller tool can transmit, as modeling service parameters, CORBA object references to the callee tool. At the callee side, those references are used for constructing CORBA stubs which enable the callee tool to update models by invoking their callback operations.

This approach supports pointer preservation. The pointers (i.e. object references) to CORBA objects remain valid all the object life time (until the objects are destroyed).

However, it has a restriction on the memory representations: model elements must be represented as distributed objects. As most existing tools have not been planned for this kind of integration, they usually represent models as local data structures. Consequently, to apply this approach, the model representations need to migrate from local data structures to distributed objects. This causes inefficiency and requires a lot of tool integration efforts.

- Inefficiency. The distributed objects are much more complex than regular local objects as they need to handle the callback mechanism. This approach forces tools to abandon the simple light-weight data structures and to use more costly representations.
- Tool integration efforts. In order to migrate from local data structures to distributed objects, we need to adapt the tool programs with the interfaces provided by the distributed object stubs. This adaptation causes the following efforts. First, the data structures of models are complex and they vary with each kind of models (e.g. UML models or domain-specific models); therefore, implementing stubs for representing each kind of models causes a considerable effort. Second, as the manipulation of local data structures is based on local memory access (e.g. creating/removing model elements using local memory allocation/deallocation; accessing non-encapsulated model element content), it is not compatible with distributed object paradigm, where all access must be done through encapsulating interfaces. Therefore, migrating from local data structures to stubs requires a considerable modification in tool programs: all access to local data structures must be changed to stub operation call.

Consequently, we can conclude that this approach does not satisfy our objective as it does not support tool integration without changing the existing model representations of tools.

Moreover, this approach does not deal with the way the caller gets notified of element creation/removal. Consequently, the caller may fail to adapt its program to model element creation/removal which is performed by the remote callee.

Pass-by-copy. This approach has no restriction in memory representation. Each tool has total freedom in choosing memory representation of models. During service invocation, the marshaling and unmarshaling techniques [8] [15] are applied for converting models form one memory representation to another.

Most commonly used RPC middleware, such as Java API for XML-Based RPC (JAX-RPC) [18], CORBA (passing value types), offers a similar solution for inout parameter passing by using the holder (an object that points to the parameter value). Before the invocation, the caller tool make the holder point the original value. After the invocation, the holder is modified by middleware so that it points to the updated value.

However, the holder approach implies that the updated value does not replace the original value in the same memory address (a new memory address is allocated for storing the updated value), as illustrated in figure 4. Consequently, all the pointers to model elements in the caller tool become obsolete after the invocation (i.e. they point to the old version of model elements). Therefore, we argue that this approach does not support pointer preservation.

Moreover, this approach does not support the notification of model element creation/removal. The caller only observes a new version of the model. It does not know which elements are newly created or removed.

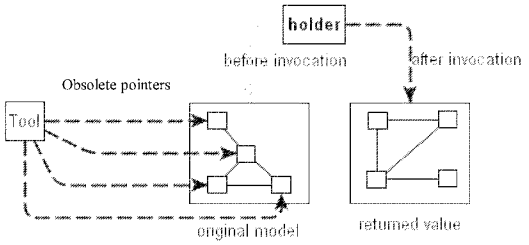


Figure 4. Inout parameter passing by using holder

5. OUR APPROACH: PASS-BY-COPY, WITH UPDATE PROPAGATION

5.1 Overview

As we have identified that the pass-by-copy approach supports the heterogeneous memory representations but it does not provide in-place update mechanism for preserving pointer validity, we extend this approach by adding the missing update mechanism. Our parameter passing mechanism follows the following steps. 1) The original model is transmitted to the callee side as a copy. 2) After the callee performed the update on the copy, the copy is transmitted back to the caller side. This copy is stored in a new memory address. 3) The update automatically propagates from the copy to the original model. This last step is the added-value of our approach. The tree steps are illustrated in figure 5 (top).

For making this mechanism totally transparent to tools, we propose a middleware layer that manages it. This layer consists of components called stubs. As illustrated in figure 5 (bottom), the caller tool invokes the modeling service by invoking the API of the *caller stub*. The caller stub marshals and transmits the model (parameter) to the *callee stub*. The callee stub performs unmarshaling. Then it asks the callee tool to perform the modeling service on this model copy. Next, it marshals and transmits the updated copy back to the caller stub. The caller stub then unmarshals the copy and propagates the update to the original model.

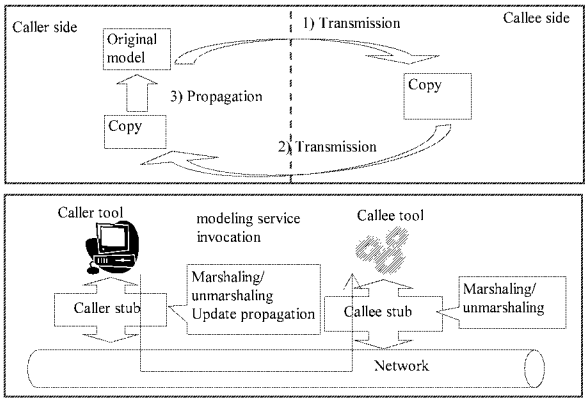


Figure 5. Pass-by-copy with update propagation

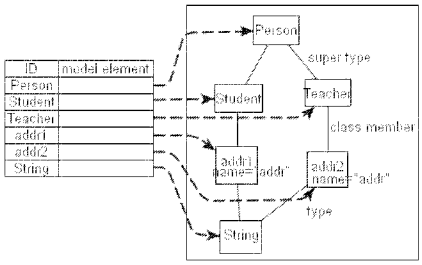
5.2 Model Element Correspondence

The main problem to solve is how the caller stub performs update propagation. We have identified that the information about model element correspondence is necessary for solving this problem. If a callee tool updates the element E1 in its local copy, in order to propagate the update to the original model, it is necessary to identify which element in the original model corresponds to E1, i.e., the element from which E1 was copied.

However, the classical marshaling/unmarshaling techniques [8] [15] do not preserve this correspondence information. Consequently, when the result model is copied back to the caller side, the caller stub will face difficulties in matching the model elements in the transmitted-back model and the ones in the original model in order to perform update propagation. For this reason, we propose a new marshaling/unmarshaling mechanism that preserves the correspondence information in order to enable update propagation.

This mechanism is manipulated by the caller and callee stubs. The stubs maintain the correspondence information by attaching an ID to each model element. Two model elements (in two different copies) are correspondent if they have the same ID. Before transmitting the original model to callee side, the caller stub assign a unique ID to each model element of the original model. The IDs are transmitted with the model to the callee stub. After service execution, the IDs are transmitted back with the result model to the caller stub. The caller stub then compares the IDs of the model elements in the received copy with the IDs of the model elements in the original model to find model element correspondence.

The manipulation of IDs is totally transparent to caller and callee tools, i.e., the tools are not aware that the IDs are assigned to model elements. The IDs are assigned to model elements stored in the memory using the ID table, which is managed by the stub, c.f. figure 6 (top). The ID table only points to the model elements without modifying them, so it is transparent to tools. When the model is marshaled, the stub serializes the IDs together with model elements. When the model is unmarshaled, the ID table is reconstructed.




```
<uml:Class xmi:id="Person" name="Person">
<uml:Class xmi:id="Student" name="Student" supertype="Person">
 <ownedAttribute xmi:type="uml:Attribute" xmi:id="addr1" name="addr" type="String" />
</uml:Class>
 .....
```

Figure 6. Maintaining model element correspondence using IDs

We have chosen XML Metadata Interchange (XMI) format [27], which is the OMG standard, as the model transmission format. In this format, the ID can be attached to a model element with the XML attribute xmi:id, which is assigned to each XML element representing a model element. We illustrate a part of a serialized model in figure 6 (bottom).

It is worth nothing that the use of IDs is already proposed in XMI standard. However, our novel contribution deals with how to preserve the same IDs when models are transmitted multiple times (from the caller to the callee and backwards) in order to keep correspondence information.

5.3 Realizing the Objectives

Supporting heterogeneous memory representation. Our mechanism includes the marshaling and unmarshaling technique. This enables models to be heterogeneously represented in different tools. During transmission, models are converted from one representation to another in a tool-transparent way.

Preserving pointers to model elements. Based on the model element correspondence information as described, we propose here an update algorithm that the caller stub performs for propagating update. This algorithm modifies the content of the existing model elements while they remain in the same memory addresses. Consequently, it keeps the model element pointers of the caller tool program valid.

The algorithm consists of three rules according to the update actions, i.e., model element creation/removal, property update, and link update, as expressed in table 1. Rules R2 and R3 intend to update the properties and links of existing model elements without re-allocating new memory addresses. The stub only allocates new memory addresses for the model elements that did not exist before (by using rule R1). The *created element list* and the *removed element list* are created for the notification purpose, as will be explained next.

Table 1. Update propagation algorithm

Let OEi be a model element in the original model; Ei be a model element in the copied model; and OEi corresponds Ei, where i, j are 1, 2, 3…

R1: Creating/removing model elements.

- For all Ei in the copy: If no element in the original model corresponds to Ei (i.e. OEi does not exist), then the stub creates OEi in a new memory address. It also adds this address to the *created element list*.
- For all OEi in the original model: If no element in the copy corresponds to OEi (i.e. Ei does not exist), then the stub adds OEi's address to the *removed element list*.

R2: Property update.

- For all Ei in the copy: The stub updates all the properties of OEi according the Ei. In other words, the new values of the properties are copied from Ei to OEi.

R3: Link update.

- For all links <Ei, Ej> in the copy: The stub creates the link <OEi, OEj> in the original model (unless already exists).
- For all links <OEi, OEj> in the original model: If the link <Ei, Ej> does not exist, then the stub removes <OEi, OEj>.

Notifying of model element creation/removal. At the end of the invocation, the caller stub returns two collections of pointers to the caller tool: a collection of the pointers to the newly created model elements (created element list) and to the removed model elements (removed element list). The created element list allows the caller tool to locate newly created elements. The removed element list allows the caller tool to eliminate the pointers to the removed elements. The caller tool may need to free the memory associating with those removed elements, depending on the programming languages used (not required in languages with garbage collection).

6. EXPERIENCE IN A CONCRETE EXAMPLE

We have realized the example presented in section 3.2 with the UML editor, provided by the EMF toolkit, and the MT engine, which is implemented ourselves using the Java programming language.

The integration of both tools consist in building the caller and callee stubs. Both stubs are hand-written in the moment. However, we aim to generate them automatically in the future. The caller stubs provides an interface to the UML editor to invoke pullupMembers (interface MTEngineCallerStub). The MT engine needs to implements an interface for receiving service execution request from the callee stub (interface MTEngine). Both interfaces are illustrated in table 2.

MTEngineCallerStub provides method invoke_pullupMembers allowing the UML editor to apply the modeling service as if it were local. The UML editor specifies the modeling service parameter (i.e. a model) with argument diagram. A model is represented by as collection of Java objects (class Model); each object corresponds to a model element contained in the model.

As the result of invoke_pullupMembers, the model will be in-place updated, i.e., the content of the objects contained in the Model's collection will be modified, and the collection is modified according to the newly created model elements or removed model elements. This method also returns the notification of model element creation/ removal (class UpdateNotification).

MTEngine provides the method exec_pullupMembers allowing the MT engine to receive the model and to update it. This method enables the MT engine to manipulate the remote model (specified as argument diagram) in the same way as manipulating local models (in fact, the MT engine manipulates the copy of the remote model). After exec_pullupMembers ends, the callee stub transmits the updated model back to the caller stub for update propagation.

The mechanism proposed in the paper is generic. We have implemented a library that can be shared by the stubs of several tools. This library including the following functionalities: 1) model marshaling/ unmarshaling with preservation of model element correspondence, 2) model transmission via RPC and 3) model update propagation.

The model marshaling/ unmarshaling part has been developed for Eclipse Modeling Framework (EMF) and Java Metadata Interface (JMI) representations, which are widely used in current MDA-based tools.

The model transmission part has been implemented in our previous work [5]. It is based on the Web Services RPC mechanism, which is widely used for large scale software integration. As Web Services are platform-independent and domain-independent, they enable the transmission of any kinds of models between any tools.

The model propagation part has been developed for the EMF representation. It enables models represented as EMF Java objects to be remotely updated by any tools, which can uses other model representations (not necessarily EMF).

Table 2. Interfaces for stubs

```
public interface MTEngineCallerStub {
  UpdateNotification invoke_pullupMembers(Model diagram)
}
public interface MTEngine {
  void exec_pullupMembers(Model diagram)
}
public class UpdateNotification {
  public Collection createdModelElements;
  public Collection removedModelElements;
}
public class Model {
  public Collection modelElements;
}
```

7. RELATED WORKS

Our work concerns two domains: CASE Tool integration and parameter passing techniques in RPC.

As regards tool integration, Kath and al. [20] proposes a framework for integrating CASE tools with a shared repository. All models are stored in the same repository; therefore they can be accessed by all tools. This repository is based on CORBA, so it

provides remote access to the tools. Our approach is different in the following points. First, our approach preserves the existing model representations of tools, which include local data structures. Therefore, it enables the integration of tools which have not been planned for integration. Second, our approach supports the notion of services, i.e., tools can invoke the services of other tools to extend their functionalities. On the other hand, their approach supports only data integration: tools can exchange models but can not request the services of other tools.

Another approach for tool integration is based on the exchange of XMI difference file [21]. XMI difference is a part of the XMI standard that proposes the representation of changes to models. In this approach, a tool can update the model of another tool but sending it an XMI difference file. This approach is different from ours as follows. 1) No transparency to the updater tool: The tool needs to calculate difference in order to produce the XMI difference file. Our approach enables tools to update remote models as if they were local. The tools have no extra tasks such as calculating difference. 2) No transparency to model-owner tool: The tool needs to parse the XMI difference file and apply update to the model in its memory. In our approach, the in-memory model update is directly performed at middleware layer.

As regards the parameter passing, Kono and al. [22] has proposed a mechanism for transmitting pointers (to parameter values) as parameters in RPC. The data referenced by those pointers are transmitted and copied automatically to the callee side when the pointers are dereferenced (accessed). At the end of the invocation, the copied data is transmitted back to the caller side, and replaces the same address as the original data. However, this approach is not transparent to the callee when it wants to create new elements or remove elements. The callee must use the special operations for allocating and freeing remote memory (extended_malloc, extended_free). Moreover, this approach does not enable the caller to get notified of model element creation/removal. Our approach enables the callee to use local memory allocation/freeing operations and also supports the notification to the caller.

8. CONCLUSION AND FUTURE WORKS

We have presented the technique for integrating CASE tools, in order to support MDA software development. Our idea is based on RPC, enabling a CASE tool to extend its functionalities by invoking the modeling services provided by other CASE tools. This paper focuses on the inout parameter passing mechanism, which allows tools to update models owned by other tools. This mechanism hides the complexity of model update from tools. First, it enables a tool to update models regardless heterogeneous model representations. Second, it enables a tool to update models located in the memory of another remote tool transparently, as if the models were local. Third, it ensures the integrity between the updated models and the tool that owns the models.

This work has been realized in the environment for tool integration called ModelBus. The implementation consists in creating a stub of each tool that is responsible for managing parameter passing in service invocation. Our parameter passing mechanism is generic. It has been implemented as a shared library that is reusable by the stubs of several tools. The current implementation is based on the Web Services RPC mechanism, which is widely used for large scale software integration.

ModelBus is a part the MODELWARE project, which aims to provide an environment for MDA software development. As future works, we aim to apply this technique for integrating the tools provided by the project partners, such as Objecteering [29], the OCL engine from Kent Modeling Framework [1], ATL model transformation engine [7]. This integration will prove the usability of our solution.

From the technical points of view, we also consider the following future works. In this work, we assume that the model update is done in synchronous RPC (The caller is blocked until receiving responses). In order to extend our work, we want to apply our approach in asynchronous context. For example, a tool T1 sends a model to another tool T2. T2 updates this model asynchronously. Then, T1 reconnects to T2 in order to receive the update.

Another future work concerns the transmission of partial models. It is inspired by the fact that a model usually contains a large volume of data; however, when it is transferred to another tool, only a part of it is actually used by the tool. Therefore, we would like to take advantages of this fact for optimizing tool integration.

9. REFERENCES

[1] Akehurst, D., Patrascoiu, O. OCL: Implementing the Standard for Multiple Metamodels, In OCL2.0- Industry standard or scientific playground?, *Proc. of UML'03 workshop*, 2003.

[2] Assmann, U. Graph rewrite systems for program optimization, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(4), July 2000.

[3] Blanc, X., Bouzitouna, S. & Gervais, M.-P. A Critical Analysis of MDA Standards through an Implementation: the ModFact Tool, In *Proc. of 1st European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, 2004.

[4] Blanc, X., Gervais, M.-P., Sriplakich, P. Model Bus: Towards the Interoperability of Modeling Tools, In *Proc. of the European Workshop MDAFA 2004*, LNCS 3599, Springer, 2005.

[5] Blanc, X., Gervais, M.-P., Sriplakich, P. Modeling Services and Web Services: Application of ModelBus, In *Proc. of the Int'l Conf. on Software Engineering Research and Practice*, 2005.

[6] Bernstein, P. A. & al. Microsoft Repository Version 2 and the Open Information Model, *Information Systems* 24(2), 1999.

[7] Bézivin, J., Hammoudi, S., Lopes, D., and Jouault, F. Applying MDA Approach for Web Service Platform, In *Proc. of the 8th Int'l IEEE Enterprise Distributed Object Computing Conf.*, 2004.

[8] Cao, F. & al. Marshaling and unmarshaling models using the entity-relationship model, In *Proc. of the 20th Annual ACM Symposium on Applied Computing*, 2005.

[9] Costagliola, G., Deufemia, V., Polese, G. A Framework for Modeling and Implementing Visual Notations With Applications to Software Engineering, *ACM Transactions on Software Engineering and Methodology* 13(4), 2004.

[10] Crawley, S., Davis, S., Indulska, J., McBride, S., Raymond, K. Meta-Meta is Better-Better, In *Proc. of the IFIP WG 6.1 Int'l Working Conf. on Distributed Applications and Interoperable Systems*, 1997.

[11] Czarnecki K., Helsen S. Classification of Model Transformation Approaches, In *Proc. of the 2nd OOPSLA Workshop on Generative Techniques in the context of MDA*, 2003.

[12] Eclipse, Eclipse Modeling Framework, http://www.eclipse.org/emf

[13] Eden, A., Yehudai, A., Gil, J. Precise Specification and Automatic Application of Design Patterns, In *Proc. of the Int'l Conf. on Automated Software Engineering*, IEEE, 1997.

[14] A Value Transmission Method for Abstract Data Types

[15] Herlihy, M.P., Liskov B. A Value Transmission Method for Abstract Data Types, *ACM Transactions on Programming Languages and Systems*, 1982.

[16] IBM, Rational Software Architect, http://www.ibm.com/software/awdtools/architect/swarchitect

[17] Java Community Process, *Java Metadata Interface (JMI) Specification version 1.0*, http://www.jcp.org, 2002.

[18] Java Community Process, *The Java API for XML Based RPC (JAX-RPC) 2.0*, http://www.jcp.org , 2004.

[19] Jørgensen, J., Christensen, S. Executable Design Models for a Pervasive Healthcare Middleware System, In *Proc. of the 5th Int'l Conf. on the Unified Modeling Language*, 2002.

[20] Kath, O. & al. An Open Modeling Infrastructure integrating EDOC and CCM, In *Proc. of the 7th IEEE Int'l Enterprise Distributed Object Computing Conf.*, 2003.

[21] Keienburg, F., Rausch, A. Using XML/XMI for Tool Supported Evolution of UML Models, In *Proc. of the 34th Annual Hawaii Int'l Conf. on System Sciences*, IEEE CS, 2001.

[22] Kono, K., Kato, K., Masuda, T. Smart Remote Procedure Calls: Transparent Treatment of Remote Pointers, In *Proc. of the Int'l Conference on Distributed Computing Systems*, IEEE CS, 1994.

[23] Mira da Silva, M., Atkinson, M. P., Black, A. P. Semantics for Parameter Passing in a Type-complete Persistent RPC, In *Proc. of the 16th Int'l Conf. on Distributed Computing Systems*, IEEE, 1996.

[24] Porres, I. Model Refactorings as Rule-Based Update Transformations, In *Proc. of the 6th Int'l Conf. on the Unified Modeling Language*, 2003.

[25] OMG, *MDA Guide Version 1.0.1*, document no: omg/2003-06-01, 2003.

[26] OMG, *Meta Object Facility version 1.4*, document no: formal/2002-04-03, 2002.

[27] OMG, *XML Metadata Interchange (XMI) Specification version 2.0*, document no: formal/03-05-02, 2003.

[28] Richters, M., Gogolla, M. Validating UML Models and OCL Constraints, In *Proc of the 3rd Int'l Conf. on the Unified Modeling Language*, 2000.

[29] Softeam, Objecteering, http://www.objecteering.com

[30] Thomas, I., Nejmeh, B. Definitions of Tool Integration for Environments, *IEEE Software*, pp. 29-34, 1992.

[31] Tokuda, L., Batory, D. Automating Three Modes of Evolution for Object-Oriented Software Architectures, In *Proc of the 5th USENIX Conf. on Object-Oriented Technologies and Systems*, 1999.