ModelBus

A MODELWARE White Paper

Xavier Blanc (Laboratoire d'Informatique de Paris 6)
04/12/05

1 - Context:

The success of MDA depends on the automation of software development. This automation is partially achieved by, what we call, modeling services, which are automated operations applied on models (i.e. operations having parameters that are models). Today many CASE tools propose modeling services such as model storage, model editing, model transformation, model verification or model execution. In the ModelBus context and in this document, we use the term "modeling service" as an operation having models as inputs and outputs.

To complete the automation of software development, there is a need to execute what we call an "MDA scenario". For example, a classical MDA scenario can be:

1. Obtain a UML1.4 class diagram model within a UML repository
2. Check this model against OCL expressions
3. Transform the checked model into an EJB model (UML model profiled by the UML for EJB profile)
4. Generate EJB code from the EJB model

The execution of an MDA scenario needs to at least realize the calls of the corresponding modeling services. But, as modeling services may be provided by heterogeneous tools the first problem that must be fixed is the problem of modeling service interoperability.

2 - Modeling service interoperability: ModelBus

Tools that provide modeling services are very heterogeneous:

1. They have their own model definitions (for instance Objecteering and Rational Rose have their own UML1.4 metamodel). It is then hard to check if two modeling services can be used conjointly (do they share the same metamodel?).
2. They have their own technology for representing models (JMI in memory, EMF in memory or Proprietary). Moreover, even if they use XMI for exchanging models, they do not share a consistent mechanism for importing and exporting XMI as there are different versions of XMI.
3. They have their own access mechanisms (GUI, Web Service, API or Command Line). Moreover, those access mechanisms can be service oriented or event oriented and maybe both.
4. And last but not the least, they can be distributed over a network and can run on different operating systems

Then if one wants to program an MDA scenario as the one previously presented in the context shown in Figure 1 (where modeling services are shared across those tools), he/she has to know for each tool what the model definition is, what the technology for representing models

is, what the access mechanism is and how tools distributed are. Programming such a scenario is therefore very difficult and expensive.

In order to facilitate this programming, a natural way to proceed is to define an abstraction layer that provides modeling service interoperability transparency. This layer is in fact a bus. We name it ModelBus and it aims at:

1. Providing a model abstraction layer. The programming of MDA scenarios is concerned with models, since modeling services operate on models. Providing a model abstraction layers means that the concern when writing an MDA scenario need only be focused on models, which can be of various types (UML Class Diagram models, UML Use Case models, QVT models, OCL models, etc.).
2. Providing model representation transparency. The programs executing MDA scenarios should not have to encode models into the adequate format accepted by tools (JMI, EMF, XMI).
3. Providing modeling service access transparency. The programs executing MDA scenarios must be independent of the technologies used by the tools. Programming an invocation of a modeling service provided by a tool must be uniform whatever the technology of the tool. The service access transparency includes event support and service support.
4. Providing modeling service location transparency. The programs executing MDA scenarios should be independent of the location of the tools providing modeling services, i.e., they should not have to include hard-coded location information. Tools can be either local or remote: communication with tools must reflect this.

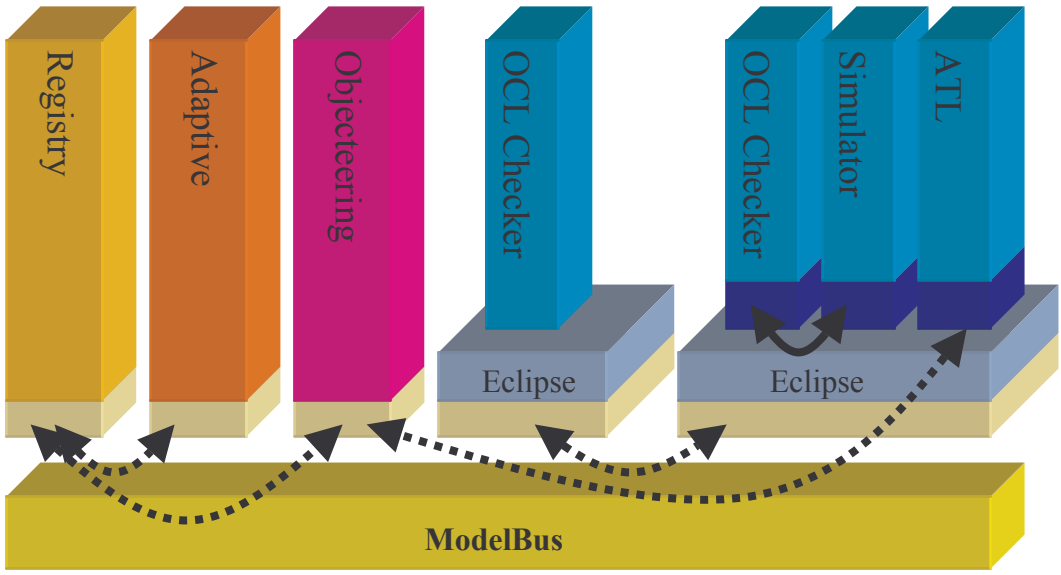

Figure 1: Modeling service interoperability with ModelBus (Both local/eclipse and remote)

3 – ModelBus within ModelWare

ModelBus does provide a model abstraction layer within ModelWare. This abstraction layer is expressed as the so called ModelBus abstract metamodel. Modeling services are represented as instances of this metamodel. They formally define the inputs and the outputs of the modeling services by expressing their types (those types are based on metamodels). In Figure

1, all tools are described thanks to this ModelBus abstract metamodel. Thanks to those descriptions, it is then possible to discover if tools at least share the same model definition and then are able to interoperate.

ModelBus does provide model representation transparency. ModelBus takes charge of model representation translation when need. For example in Figure 1, ModelBus will translate Adaptive model representation (e.g. XMI) into OCL Checker model representation (e.g. EMF) when they will interoperate. Within ModelWare model representation transparency will only be for XMI and EMF.

ModelBus does provide modeling service access transparency. This transparency is expressed by the so called adaptors. Adaptors take charge of the modeling service calls. Modeling service access transparency will support both local (i.e. within Eclipse) and remote interoperability. Modeling service access will support both synchronous and asynchronous interoperability. Moreover, modeling service access transparency will also support events. Tools will be able to send and receive model related events. From a technical point of view, the initial version of ModelBus within ModelWare will only support calls to modeling services that are provided by tools either as web services or as eclipse plug-ins.

ModelBus does provide modeling service location transparency within ModelWare. This transparency is expressed by the so called modeling service Registry that stores all needed information for calling modeling services.

ModelBus does not provide a dedicated language for expressing MDA scenarios. This language would be provided on top of ModelBus. But, adaptors provided by ModelBus for calling modeling services can be directly used for programming an MDA scenario.

ModelBus does not provide a mechanism for finding adequate modeling services. ModelBus does not manage any information categorizing modeling services. In the ModelBus architecture, this mechanism is considered as a specific modeling service.

ModelBus does not provide a graphical integration of tools. ModelBus is only concerned with modeling services interoperability. It only addresses service location transparency, service access transparency at a model abstraction layer.

4 – F.A.Q

Q1: What is the added value of ModelBus versus an Eclipse plugin mechanism?
The Eclipse plugin mechanism does not offer a model abstraction layer. A tool that provides modeling services is mainly defined by a Java API within Eclipse. The Eclipse plugin mechanism works only within Eclipse. Without ModelBus, there is no generic approach for calling modeling services provided by tools that are outside of Eclipse.

Q2: What additional technology is needed to run the bus?
ModelBus will be delivered as a standalone application. It will be developed as a Java application. All needed libraries for making modeling services calls will be integrated within it.

Q3: Do we need a repository or only registry to store addresses?
In ModelBus, we use a repository for storing the information needed for calling modeling services. This information contains the addresses of modeling services but also how the modeling services represent models.

Q4: Are there generic ModelBus services described in a standardized way?
We are working on a set of samples. Currently there is no intention to standardize some modeling services.

Q5: Is ModelBus tool-oriented or service-oriented?
ModelBus is modeling service oriented.

Q6: Is there a repository of ModelBus services?
Yes. This repository just stores information needed to call modeling services.

Q7: In local mode, ModelBus needs a Java Virtual Machine, and in a distributed mode ModelBus uses Web services. Why not use web services always?
With Web Services, models are encoded with XMI. We argue that XMI is not very well adapted for fine grain interoperability whereas Java local calls are good for that. Moreover, with Web Services, parameters are transmitted by value whereas it could be useful to transmit them by reference when it is possible.