MoBIES

Open Tool Integration Framework

OTIF

Revision	Date	Author
Initial HL Req. and Arch.	2/12/02	G. Karsai
HL description of protocols	2/27/02	G. Karsai
Discussion on UDM/Meta issues	2/28/02	G. Karsai

High-level Requirements and Architecture Design

The purpose of this document is to document the high-level requirements and design of an Open Tool Integration Framework for the MoBIES program. For simplicity, we call this framework as (M)OTIF. It is not to be confused with the X-windows-based GUI framework Motif, which is dead anyway.

Tool integration, as it is understood here, is the process of coupling different types of design tools: modeling tools, analysis tools, synthesis tools, verification tools, simulators, etc., in support of a large-scale design process. It is expected that integrated tools working together contribute to the success of development processes more, than individually applying non-integrated tools to different portions of the design process. We call a particular instance of tool integration as "tool integration solution".

(M)OTIF is a framework for building tool integration solutions. It contains generic software components, but it also defines protocols for component interactions and an engineering process for creating a particular instance of the framework –a tool integration solution- that integrates a specific set of tools.

(M)OTIF addresses a number of requirements that have been identified as relevant. Below we list these requirements, and "it" refers to the tool integration framework.

- It shall clearly separate syntactic, semantic, and control issues in tool integration.
  When different tools are integrated, there are at least three different aspects to be considered: syntax, i.e. how to handle the syntactical differences among tools, semantics, i.e. how to handle semantic differences among tools, and control, i.e. how to handle the differences in the control and interactions among tools. The framework should provide mechanisms for solving all these issues in a non-interfering manner (to the extent possible).
- It shall be able to integrate tools without modifying them, if that is not feasible.
  Tools have typically three interfaces that can be used for integration purposes: persistence interface (e.g. file import, export), API (e.g. direct COM-based API to access the tool's internals), and the GUI (e.g. an interceptor mechanism that "taps into" the event stream and drawing commands between the main tool component and its visual front-end). The framework should be able to work with any of these.
- It shall support integration of tools that are deployed as web-services.
  As a new trend in software deployment, expensive tools are often provided as a service accessible and usable via the web (and not as a directly downloadable and installable package). The framework should be able to naturally integrate these tools.
- It shall support transforming the product of one tool into the input of another tool.
  Pipelining data from tool to tool does the simplest kind of tool integration. Because of syntactical and semantic differences, the pipeline frequently involves transformations. However, transformations could also incorporate other operations than strict "rewriting": for example, merging. For instance, data produced by tool A and tool B must be merged to serve as an input to tool C. The framework should allow the organization of arbitrary transformations (1-to-1 and many-to-1).
- It shall support simple techniques for simple translation needs.
  Frequently, translations are trivial textual rewriting the input data into the output data. A number of techniques are available to solve these translation problems, e.g. search and replace using regular expressions; transformation of XML files using XSLT scripts, etc. The framework should allow the implementation of simple transformations using the available tools.
- It shall support batch, transaction-oriented, and notification-based integration.
  There are number of different strategies for managing the control across different tools. In a batch-based approach, a producer tool produces a dataset, which is then passed along to the next tool in the chain. In a transaction-oriented approach, a producer tool executes a "write" transaction on a shared database, which will result in changes on that database, and a

consumer tool should execute a “read” transaction on that database to retrieve the data. In a notification-based approach, fine-grain changes are performed by a producer tool, which then sends notification messages to consumer tools (who subscribe to these notifications), which then perform appropriate incremental changes on their own. The framework should be able to support any and all of these techniques.

Based on these requirements, the following architecture is proposed for (M)OTIF (see Figure 1 below).

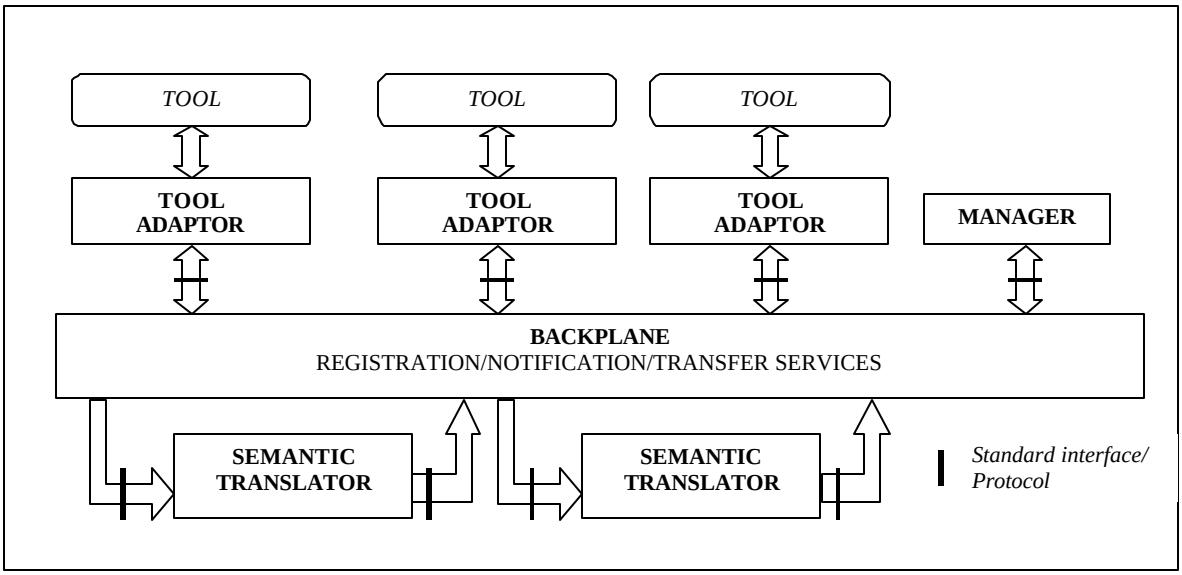

Figure 1: (M)OTIF Architecture

The architecture consists of the following components (in addition to the design tools to be integrated):

- Tool Adaptor. This component is responsible for realizing the interface wit the tool (any of the methods mentioned above) and performing syntactical transformations on the tool’s data. The tool adaptor should convert all data coming from the tool into a canonical form, and pass it along to the backplane. Similarly, data coming from the backplane should be converted by the tool adaptor into tool-specific physical data. In the case of notification-based integration, the same applies to events generated and consumed by the tools: the tool adaptor performs the syntactic conversion on the events. Tool adaptors may have state for to support stateful interactions between tools.
- Semantic translator: This component is responsible for performing the semantic translation on data (or events) among different tools. In the simplest case, it performs a mere data rewriting, but in more complex cases the translations could quite sophisticated. The translators relate producer tool(s) to consumer tool(s), although the most general, many-to-many case is possibly very rare.
- Backplane: This component is the backbone of the integration framework. It provides coordination services between the other components. The services include: registration and identification of components, notification, and physical data transfer. The backplane is typically distributed across multiple machines.
- Manager: This is a utility component for administration and debugging purposes. Administration involves enabling and disabling tools and users, etc., debugging operations allow run-time monitoring and troubleshooting the backplane.

The most challenging component in the above schema is the semantic translator. The semantic translator realizes the connection: the conceptual bridge between two (or many) tools. However, all semantic translations should operate in a common framework. This common framework could

be grounded in the abstract syntax of the tools to be integrated. The abstract syntax defines what concepts a tool works with, what association exists among those concepts, what attributes belong to those concepts and associations, and what integrity constraints exist among the concepts and associations. Tool data should always comply with the abstract syntax of the tool. We approach the semantic translation problem by expressing it in terms rewriting between two (or many) abstract syntax trees. On the "lowest level", the translators are transforming data compliant with one abstract syntax definition into data compliant with another abstract syntax. Another view of semantic translation is that of transformations between type systems: the data is always typed and the translation can be defined between a –perhaps quite complex- mapping.

The architecture also identifies the following standard interfaces and protocols. The interfaces are pair-wise: both parties implement a set of interfaces, and the protocol governs the sequence of interactions across those interfaces.

- Tool Adaptor / Backplane
- Semantic translator / Backplane
- Manager / Backplane

All of these interfaces/protocols are similar: they transfer data and events between the two participants.

The architecture above supports a number of the requirements outlined above, but for supporting transaction-oriented interactions another component is needed. This component acts as the data repository that other tools use for sharing. However, a repository from the viewpoint of the framework is just another tool, which uses the same protocols. If the protocols are rich enough to provide transaction-oriented behavior, and the repository has to be configured for the purposes of the particular integration solution anyway, there is no need for defining another, special-purpose architectural component. This is not to say that certain services of the framework, e.g. the registration service of the backplane, do not require repository-like services. However this is considered an implementation detail.

The architecture introduced above can be discussed in terms of a number of usage scenarios. Below we consider a number of these scenarios.

1. Batch-oriented pipelining of tool data files. The user of a producer tool finishes the work that produces a new dataset. She then invokes a tool adaptor and uses that to send the data to the backplane. The tool adaptor reads the tool data in its physical form, and converts it into a canonical form, and then it sends it to the backplane. The backplane determines who are the consumers of this data, and invokes the appropriate semantic translators for those consumers. The semantic translator receives the data in canonical form through the standard interface, performs the translation and sends the resulting data in canonical form back to the backplane through another standard interface. The backplane then routes this data to the consumer tool adaptor(s) that will convert it into physical data for their tools.
2. < some other scenario >
3. < another scenario>

The architecture above is generic, and it as to be customized for every tool integration solution. This process is called the instantiation of the architecture. The instantiation involves the following steps.

1. Identification of the tools to be integrated.
2. Identification of what tool-to-tool dependencies exists.
3. Identifying the concrete and abstract syntax of the tool, and how a tool adaptor can interact with the tool.

   This step is a crucial point as it builds a comprehensive meta-model of the tool, which captures the abstract syntax and the tool adaptor/tool interaction protocol. The abstract syntax is needed for implementing the semantic translators, as the semantic translation is expressed in terms of a rewriting one abstract syntax tree into another abstract syntax tree.

4. Identifying the semantic mapping and the control integration between tools that need to interact.

This is another crucial step, as it builds a meta-model for the semantic translation and the control integration between the tools. It is expressed in terms of mapping between the abstract syntaxes and the interaction protocols of the tools, identified above.

5. Developing the tool adaptors for the tools.
6. Developing the semantic translators.

These two steps involve the physical implementation of the adaptors and the semantic translators. Tool adaptor development may involve development of sophisticated parsers and unparsers, as required by the tool.

7. Integration and test.

The process described above gives a recipe for building a tool integration solution in terms of the above architecture. However, the specific details of the steps and the tools to be used in those steps are dependent on the specific design choices made.

Core Protocols

The (M)OTIF architecture is centered around a number of core protocols that govern the interactions between the tool adaptors, the semantic translators, and the backplane. The protocols are defined with the help of object interfaces and the sequencing of operations on those object interfaces.

The interfaces and protocols are divided into the following groups:

- UDM
  - Structures for meta data
  - Structures for instance data
- OTIF Management
  - Register/unregister metadata
  - Register/unregister translator
  - Utility operations
- OTIF Tool adaptor
  - Logon/logoff to/from the backplane
  - Browse backplane cache
  - Subscribe to documents
  - Publish document
  - Get notifications of publishing events
  - Handle user input requests from translators
- OTIF Translators
  - Receive document
  - Send document
  - Solicit user input

The *UDM* group is not a full-fledged protocol, it merely defines a set of structures and interfaces for representing metadata and instance data in the system. Meta-data is descriptive in the sense that it captures the metamodel of a tool. Instance data is substantive, in the sense that it is the vehicle for carrying the actual useful information. For both metadata and instance data generic structures are used. However, in order to make sense of the instance data on the receiving end, each element in the instance data has to be tagged that precisely define what metadata the element belongs to (i.e. what is its type).

The *OTIF Management* group defines the interface between the backplane and the manager. Here, the main operations include registering and unregistering of metadata, registering and unregistering a translator, and various housekeeping functions. In general, registration means

that the backplane is informed about the existence and structure of an entity (metadata or translator).
In general, the backplane can be thought of as a server with persistence. When metadata is registered with the backplane, that metadata is placed into the persistent store of the server, and the backplane will “know about” that data, and is able validate instance data with respect to it. The metadata is placed into the internal (persistent) data structures of the backplane, and it stays there until an explicit removal. Only the manager component can register or unregister metadata with the backplane.

Translators are registered with the backplane similarly. Translators are executable components that perform transformations on the instance data. Specifically, they transform instance data compliant with one meta-model into instance data compliant with another meta-model. Translators are activated and controlled by the backplane. Translators can be implemented using various technologies, and at registration time the backplane is informed about how the translator can be activated. Only the manager component can register or unregister translators with the backplane.

Housekeeping functions allow the manager to look at the current persistent configuration and the dynamic state of the backplane, and to modify it if necessary. The backplane may have an internal cache to store intermediate results, which the manager observe, and modify if necessary.

The *OTIF Tool Adaptor* group consists of the portion of the protocol, which deals with the interaction between the T/A and the backplane. When a T/A is started, it must log on to the backplane. During logon it has to supply what is the metamodel of the tool it connects to. The backplane will verify that, and if the metamodel is unknown for the backplane, then the logon fails. If the logon is successful, the T/A can work together with the backplane. At the end of the session the T/A should logoff from the backplane.

After a T/A has entered into a session with the backplane it can publish documents, as well as it can subscribe to published documents. In order to perform these activities the backplane provides services for publishing and subscription. Subscription happens by informing the backplane that a T/A is interested in receiving certain types of documents, and publishing happens by simply submitting the document to the backplane.

When a T/A ha subscribed to a specific type of document, it will receive notifications from the backplane whenever a document is produced (typically by a translator). At this time, the T/A may ask the backplane to supply the document to the T/A. The backplane may also maintain a limited-length cache of published documents that a T/A can browse and fetch if needed.

The *OTIF Translator* group governs the interaction between the semantic translators and the backplane. The translators are executables that are controlled by the backplane. The backplane starts the executable, make the documents available to the translator, and then it receives the results from it. After startup, the translator is responsible for pulling the document from the backplane, and after translation, handing the result back to the backplane. The translator may be implemented using different technologies, e.g. XSLT, UDM-based, etc. To treat all these techniques uniformly, the translators are wrapped into code that handles all the backplane interactions. Translators may have persistent state, but this is an implementation detail.

Some translators may require user input for completing a translation process. Therefore translators can request user input from the backplane, which will route these requests to the appropriate T/A. The T/A will get a notification from the backplane about the user input request, and shall ask the operator for input. The response is then handed back to the waiting translator such that it can proceed with the translation.

Issues with transferring data between components using UDM

The UDM technology provides a reflective component: every UDM-generated API is such that at run-time a meta-data network is created, which stores a meta-model —effectively the UML class diagram— of the data. If there are multiple components in the system (e.g. T/A-s, the backplane, and the translators) each one will have its own copy of the metadata, which is undesirable.

A proposed solution to this problem has two aspects:

1. The backplane should be an executable, which can dynamically load (and unload) meta-data. This can be accomplished by implementing a function in the backplane, which
    a. Receives a meta-model as a UDM data network (with type “Uml”), and
    b. Instantiates the Uml package classes (e.g. Uml::Diagram, Uml::Class, etc.) based on that to build up the meta-data structures, and
    c. Wraps them such that they become CORBA objects visible through the ORB, and complying with the UDM protocol specification.

    One consequence of this is that in the design a single entity: the backplane owns all the meta-data.
2. A client of the backplane (T/A or translator) is also compiled with UDM, so it has its own copy of the meta-data. This causes a problem in the case when local UDM objects are created and shipped to the backplane, as meta-data references will point to the local copy, not to the common shared copy in the backplane. However, the CORBA backend of UDM should be implemented such that it converts the meta-data references upon transfer. The scenario for transferring UDM data from a T/A to an S/T is as follows:
    a. T/A prepares UDM data network.
    b. Before shipping the data network to the backplane, the UDM backend converts the meta-data pointers to remote object pointer references (obtained from the backplane via CORBA). This allows the handling of the pointers by the ORB.
    c. The backplane receives the data network and ships it to an S/T. The ORB will convert the pointers to local references to the meta-objects in the backplane, but this is transparent to the user.
    d. The S/T receives the data network, with references to the (remote) meta-objects. The UDM backend converts the meta-data references to references to the local copy of the meta-data, such that the UDM data network becomes correct.