Component Generation Technology for Semantic Tool Integration[1]

Gabor Karsai and Jeff Gray
Institute for Software-Integrated Systems
Vanderbilt University
PO Box 1829 Station B
Nashville, TN 37235,USA
(615)-343-7471
{gabor, jgray}@vuse.vanderbilt.edu

Abstract— The problem of tool integration often occurs in the design and implementation of large computer-based systems that rely on software-based engineering tools. Each specialized tool contributes to a crucial step in the engineering process. It would be beneficial to capture the information in the context of one tool and use it in a different tool. However, differences in file formats and variations in the method of user interaction can make the integration of tools a formidable challenge. This paper presents a new approach to the tool integration problem and describes the framework and process that has been used to successfully integrate the data models of several tools. The technique is centered on generators that create "componentized" semantic translators.

TABLE OF CONTENTS

1. INTRODUCTION

The integration of tools used in a sophisticated engineering process is a highly relevant problem for a variety of domains, including aerospace engineering. Previous approaches focused on data transformation and integration issues. Currently, there exist a large number of related technologies that help to solve this aspect of the problem (e.g., CORBA, COM, XML). However, the increased complexity and semantic richness of tools necessitates going beyond those basic capabilities. Information captured in one engineering tool must be expressed in the ontology of another tool, and vice versa. Solving the integration problem is not easy. The diversity of tools, the variations in their method of user interactions, and differences in file formats make the task difficult. However, the potential benefits are also great: using information captured in one tool in the context of another tool saves valuable, but often uninteresting, effort.

A *semantic mapping* or *semantic translation* approach is needed that explicitly captures the semantic relationships among the data models of the tools to be integrated. Once these relationships are identified, *semantic translator* tools can be developed that implement the mapping. Obviously, when the tools evolve and/or new tools are added, the semantic translators have to be revised and the entire translation framework has to evolve.

In this paper, we describe a technology that is based on a framework approach. The framework consists of infrastructural elements and tool-specific translators. The semantic translators are componentized, and they are generated from high-level specifications. There are three types of specifications: (1) the data models for the input and the output of the translator, (2) the semantic constraints to be enforced on the data (which cannot be expressed structurally), and (3) the mapping between the two. For trivial cases, the mapping is easy to express in an ad-hoc data mapping language. However, in our practice we have found that the full power of a programming language to express portions of the mapping is often helpful. On the other hand, the mapping can be easily tied to the process of the translation – the traversal of an input data structure and the generation of an output data structure. We express the translation process in the form of traversal sequences where visiting an object during traversal may result in an action that creates an output object. We have found that the specification of semantic translators in this manner, and their automatic generation from those specifications, greatly enhances the productivity of software engineers who

[1] 0-7803-5846-5/00/$10.00 © 2000 IEEE

develop integration solutions. The semantic tool integration framework is much easier to evolve and upgrade than using straight "hand-coding," and there are no performance penalties associated with the approach. The approach has been used to permit the integration of a number of tools. Experience indicates that the techniques are feasible for large-scale integration as well.

2. BACKGROUND

Computer based systems are often characterized by a tight coupling between software and hardware. This necessitates the use of various engineering tools that model and analyze all aspects of the system, including the computing system and the physical environment. Each tool is specialized to perform a particular task in a specific domain. The difficulty lies in the fact that these tools often do not have the capability to communicate with each other and share modeling information.

In the past, several techniques were created in an attempt to alleviate this problem, but unfortunately they ended up being quick solutions that eventually turned out to be insufficient. Some of these previous attempts are reviewed below.

File Translators

The most obvious technique that may come to mind is file translators. File translators are specialized programs which do nothing more than read data generated by one tool (typically the physical data file) and convert its contents into another data file suitable for consumption by another tool. These translators are very similar to commercial tools that allow a user to convert from one file format (e.g., Word, Excel, GIF) to another (e.g., WordPerfect, Lotus 1-2-3, JPEG). Unfortunately, this approach has some serious drawbacks. Arguably the biggest shortcoming of the approach is its inherent problem with scalability. To have full integration among n file formats, as many as n*(n-1) translators would be required. Also, whenever a new format is added, a translator needs to be created that will translate the new format into all of the previously existing formats.

Middleware: CORBA, COM

The tools that are to be integrated could be viewed as individual components. There currently exist several well-established standards for software component integration: CORBA [2] and COM [1] being the two major examples. At first, it may appear that these component oriented middleware solutions offer hope to the integration problem. However, this solution also has its own problems.

Most distributed object-oriented component models rely on a method of remote method invocation. This concept allows a programmer to create a "wrapper" object around existing components, or tools. These tools can then communicate with each other through remote method calls. The communication between two independent tools can be complex. The situation becomes unfeasible as a large number of tools are added to the integration pool. The difficulty comes from the fact that these middleware solutions provide relatively low-level facilities for tool interactions. All higher-level functions are usually built from scratch. The middleware approach can be helpful in solving the data migration problem, but it offers little assistance in terms of the translation that needs to be done. This still requires deep understanding of tool behavior and tool data structures. If every tool defines its own unique way of accessing internal tool data, then the problem becomes similar to that found with file translators; i.e., each new tool that is added may require the creation of numerous new translators.

Universal language

This solution uses a radically different strategy from the above, although it requires the support of at least one of the above techniques. One can think about the tool integration problem in the context of the particular engineering process where it is needed. Processes (and organizations) tend to have their own vocabulary and idioms. So the idea comes: why not design a universal "language" (a database schema, in practice) that will be used by all the tools across the process. Once a language is defined, we just have to write translators for each tool, or setup the middleware communications to use this shared language. This is a more efficient solution because the number of translators increases linearly by the number of tools. Unfortunately, where the approach breaks down is in the practical difficulty of coming up with this universal language. Project tools are often selected using an opportunistic approach, and it is very difficult to make changes to the "universal" language during the lifetime of the project. It seems that the "universal" language is not very "universal" at all because it can't be used on another project.

PCTE

The Portable Common Tool Environment (PCTE) is an ISO standard that received much interest in the late 1980s [10]. It was expected that PCTE would serve as a standard repository to allow for the integration of tools. Recent interest in this technology has subsided and no large commercial software projects are being developed using PCTE. Many tool vendors had problems adopting PCTE because it's monolithic scope required vendors to significantly modify the source code of their tools to take advantage of the PCTE services.

Lessons Learned

This discussion has highlighted a few important questions concerning solutions to the tool integration problem. Each question addresses the feasibility of a solution with respect to time and scalability.

- How much time and effort does it cost to integrate a new tool? If it takes a long time to perform the integration, then it may be in an organization's best interest to simply translate the data manually.
- How scalable is the integration approach? The addition of a new tool should not require a large amount of modifications, nor should it require the creation of a large number of new software packages.
- How much expert knowledge is needed to realize an integration solution? One needs a very deep understanding of the tool semantics before attempting any kind of integration. How fast this understanding can be turned into an integration solution will determine the success of any kind of integration paradigm.
- What is the coupling between the individual tools and the integration technology? An integration technology that requires a tight coupling does not allow for incremental/partial adoption. Tight coupling can also be problematic for the integration of many legacy tools if tool source code must be modified in order to take advantage of the integration technology.

In the next section we present a new approach that offers a solution to the integration problem. The section describes key components of the approach and discusses its feasibility in light of the above principles.

3. The Approach

The observations made concerning the approaches presented in the previous section clearly suggest that a tool integration solution should address the issue of *semantic interoperability*. We want our tools to work together towards a goal, and in order to do that, some mutual understanding, or *shared semantics*, is needed. The tool integration solution should be the implementation vehicle for this shared semantics. It is simply not enough to provide access to the tool's data. To solve the integration problem, a solution must also address the issue of expressing the relationship of a tool's data model to this shared semantics.

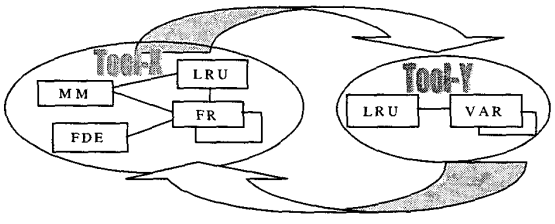

Figure 1 Tool Integration

If one associates semantics with *static semantics* (in the UML-sense [3, 8]), the tool integration problem can be visualized as shown in Figure 1. Static semantics can be described as the integration of a data model that captures the allowed entities and relationships in the tool's data with logical constraints; i.e., Boolean invariants. Given tool X, with a particular data model, we want to map the data model of tool X into the data model of tool Y. If we restrict the data model used to the "entity-relationship-attribute" variety [3], tool integration means solving the mapping problem between two database schemas. Unfortunately, if we have more than two tools, the mapping problem becomes complicated, and we get to the same scaling problem as we have seen with the file translators. It is more feasible to establish an *integrated data model* first and then map the data model of each tool into that, as shown on Figure 2. The integrated data model (IDM) can be defined as a *data model that is rich enough to contain data from any of the tools.* The IDM is the vehicle that implements the *shared semantics* across the tools.

Note that this integrated data model is neither the union nor the intersection of the data models of the individual tools because tool data models will overlap (although not completely). For example, two different tools may contain the same semantic element, yet have a different name for this element. Therefore, the union of the two names into the IDM would not be correct; each of the entities must be mapped to the same concept in the IDM. Similarly, with respect to intersection, two tools may call a semantic entity by the same name, yet have different semantics for the entity.

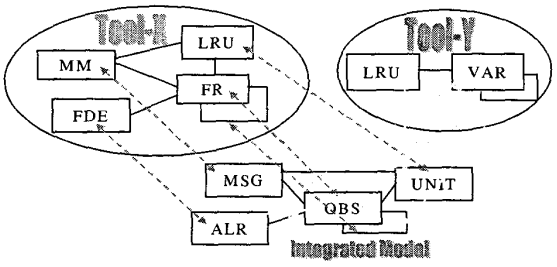

Figure 2 Tool Integration with Integrated Data Model

The Architecture

The architecture of the approach is shown in Figure 3. The two major components of this architecture are the Integrated Model Server (IMS) and the Tool Adaptors (TA). We chose CORBA [2] to implement the communication medium between these components.

The core responsibility of the IMS is to provide *semantic translation services* for the constituent tools. By semantic translation we mean a transformation of data from one data model into another one while preserving the semantics of the input data model and enforcing the semantics of the

output data model. Again, semantics is understood here as static semantics, expressed in the form of constraints on the data. The IMS also provides a short-term repository for storing the result of the translation. The schema used in the repository is that of the Integrated Data Model.

With regard to our specific implementation, the Microsoft Repository is used in the Integrated Model Database to provide meta-data management services. The underlying database that stores the models can be either Microsoft Access or Microsoft SQL Server.

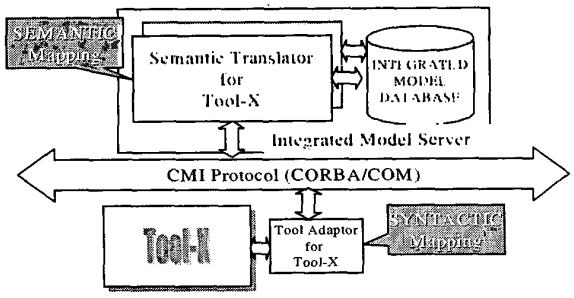

Figure 3 Tool Integration Architecture

The Tool Adaptors are responsible for reading the tool specific data and converting it into a model that can be sent to the IMS. Once the data is read from the tool, the adaptor performs a *syntactic translation* on the data from the native data format of the tool to that of the middleware data-structures. They also must be able to take a model from the IMS and convert it into a form that can be read by the tool. Thus, mature tool adaptors are bi-directional. Different tool adaptors will interact with their corresponding tools in different ways depending on the manner that a tool stores its underlying data. Therefore, one tool adaptor may read from a database that the tool uses to store data while another tool adaptor may need to read from a text file. It is even possible to have a tool adaptor access the underlying model through COM if it is supported by the tool.

There is a common sequence for interacting with the architecture whenever a tool wants to make its data available. First, the tool's associated TA must be started. The TA obtains access to the tool's data and begins to construct a network of objects that will represent a model that can be sent to the IMS. After the TA sends the model to the IMS, the IMS receives it and begins to invoke the semantic translator associated with that tool's data model. The translated model is then stored in the repository. At this point the data is transformed into an IDM-compliant form. The reverse process is very similar. If a particular tool requires a model stored in the repository, the IMS first retrieves the model from the repository and invokes the semantic translator associated with the tool. The result of the translation is then shipped to the TA, which converts it to the physical data format of the tool.

Notice that the principle of separation of concerns is observed. The concepts of syntax and semantics are handled in separate components. The IMS is concerned about semantic issues while the TAs are concerned only with the syntax of the data. This distinction makes the development of the integration solution easier. The two components are bound together by a common interface for data interchange. This approach is different from the one adopted by PCTE, where there is a tighter coupling between individual tools and the integration technology [10].

The Common Model Interface

The Common Model Interface (CMI) defines the rules of communication and the form of the data-structures used in the interactions between the IMS and the TAs. The CMI is the same across all the tools: this is the common, canonical "form" into which all tool adaptors translate their data. The CMI has many components related to data transfer and interaction with the IMS, but only the most significant aspects will be discussed here. As can be seen in Figure 3, the CMI is implemented as a CORBA IDL specification. The objects that receive the method invocations from the TAs always reside in the IMS.

The primary function of the CMI is to provide a method of data interchange between the TAs and the IMS. The data structure exposed through the CMI resembles the traditional entity-relationship-attribute data model. Data consists of attributed objects, which can be *models*, *entities*, and *relations*. An attribute is simply a key-value pair (the data type of values must be from a small, but powerful set of primitive data types, and arrays of primitive values are allowed). An entity is a simple attributed object, without any further structure. A relationship is an attributed object that has two collections of objects, called *roles,* associated with it: these collections contain entities or models that play those roles in the relation. A model is an attributed object that contains entities, relations, and other models.

It has been our experience that the CMI data model is capable of representing data from any tool. However, differentiation between the tools cannot be made with respect to which tool is represented by a particular data model. To overcome this problem, each data object also has a type tag that indicates the meaning of the object. That is, it is not enough to say, "this is an entity", but one also has to say, "this is an entity of type X of tool A".

The CMI makes this distinction apparent by dealing with the data on two-levels. *Meta-data* describes the data model of a particular tool. Physically, meta-data contains models, entities, and relations, but these are *meta*-models, *meta*-entities and *meta*-relations that describe the tool's data model. The IMS exposes the meta-data of each of the tools

as CORBA objects. Thus, each tool, or a generic browser, can access the meta-data for each of the tools. The *instance-data* is the actual data to be transferred. The instance-data contains models, entities and relations, where each data object is tagged with the corresponding meta-data object's id (technically an object reference). This tagging makes it possible for the IMS to determine the "real" type of a data object. Also, this makes it possible for a TA to get the same information.

As a TA constructs a model to be shipped to the IMS, it should tag every instance data object with the proper meta-data references. Likewise, when a TA receives a model from the IMS, it can create a new model in the native tool format by using the meta-tag relations to process each data item correctly.

Note that we are relying on the built-in CORBA mechanism for translating the object references. Another mechanism used here is for marshalling/unmarshalling: CORBA transforms complex data-structures into a network-compliant flat format suitable for transfer. Unfortunately, CORBA marshalling code typically cannot handle circular structures. Therefore, the data model uses a form of indirect object references in the case of relations: the roles do not directly "contain" (i.e. reference) the objects involved in the relations, instead an object id is kept which uniquely identifies the object.

Aside from providing the common data interchange interface, the CMI offers several other services. These services are expressed as a set of interfaces:

- Directory services. The contents of the IMS repository can be traversed and viewed as a directory hierarchy. Models and directories can be viewed by calling methods defined in this interface.
- Session management. Models can be fetched and stored by calling operations in this interface. Also, models may be removed from the repository using an operation in this interface.
- IMS access. A user or TA must log into the IMS before using it. This interface provides login/logout operations as well as the ability to receive the IMS system clock time.

The Evolution of the System

Evolvability is a key metric for assessing the feasibility of an integration solution. The architecture discussed above only gives the framework for implementing an integration solution: it does not speak about how the system evolves. An integration solution will never stay constant. New tools will always need to be either added or removed from the architecture. This continuous change necessitates the designer to place emphasis on how the system will evolve over time.

During the evolution of the system, the most frequent problem is the addition of new tools. This means a new tool adaptor has to be developed and the IMS should be upgraded to "understand" the new tool. The upgrade means changes in the IDM (for the repository), and the development of new semantic translators that can manage the data of the new tool. Both of these are non-trivial steps, especially considering that we can already have a number of tools integrated in the system.

The solution chosen here is closely related to work on Model-Integrated Computing (MIC) [9]. Model-integrated computing relies on the interpretation and use of domain-specific models in run-time environments. The domain models capture the relevant entities and their relationships in a specific domain and are used in a generation process to create executable systems. MIC has been successfully applied in the development of various computer-based systems, including aerospace, manufacturing industry, and testing applications. In MIC, domain models are used to generate components that implement a system.

There are two kinds of models that exist

1. The data model for each tool, as well as the integrated model itself, must be specified.
2. The translation model is a specification that describes how the semantic information in one model is to be mapped into the model semantics of another tool.

The evolutionary capabilities of an integration solution can be greatly enhanced if we can capture and utilize these models as components. The IMS was not designed to be a large stand-alone program. Rather, the design goal was to create a framework that accepts "pluggable" components.

Figure 4 shows the internals of a semantic translator in the IMS architecture. The reusable components contain the generic interfaces to the network side and to the repository side (accessible through the very same interface – just different implementations), the implementation of CMI services (directory, session and IMS access), and other housekeeping functions. To instantiate the IMS framework for a particular tool integration solution one has to build the semantic translators.

In the IMS, the semantic translators are not created by hand. Rather, generators are used to create the translators. The generators receive the data model and translation specification as input and generate C++ code that will perform the translation. Please see Figure 5 for a description of how translators are generated and structured within the framework. Each semantic translator has a set of static objects that represent the meta-data of the corresponding tool. These objects are created from the models by invoking a special generator. This generator requires three models: (1) the data model of source, (2) the data model of the destination, and (3) the translation specification of the particular semantic translator that will perform the conversion.

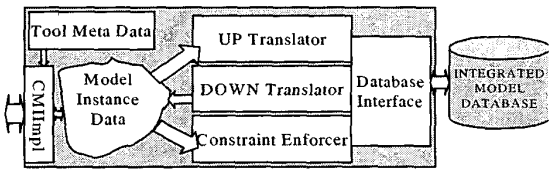

Figure 4 The Architecture of a Semantic Translator

Each tool has two semantic translators associated with it. One translator, the "up" translator, performs a translation from the tool data model into the integrated model. The "down" translator operates in the reverse; it will translate from the integrated model into the tool data model. The generated C++ code of each translator for every tool is linked (along with all other generated code and the framework library elements) to build up the IMS.

Naturally the specification of the translation and mapping is key to the whole solution. One might think that the mapping is easy to formalize in the form of mapping rules. In the most general sense, we have to describe the mapping:

$$map: (M,E,R,A) \rightarrow (M',E',R',A')$$

where M,E,R, and A stand for models, entities, relations, and attributes, respectively. It is very easy to invent a simple mapping language that captures this, and it is simple to use and simple to generate code from it.

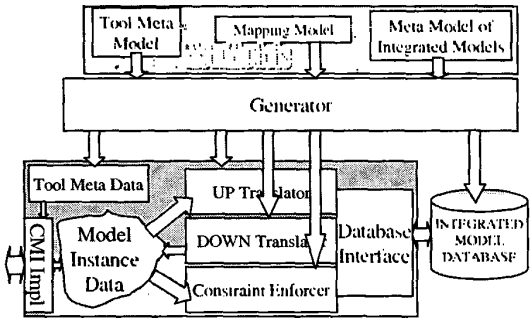

Figure 5 Generation of a Semantic Translator

However, after looking at the first domain data model of an actual application the mapping language idea was quickly abandoned. One problem with the data model was as follows. In one of the tools the surface data model was a simple object model, with a single level of containment. That is, models of type M contained entities of type E. In another tool the data model was hierarchical: models of type M' contained entities of type E' and models of type M'. At closer inspection, it turned out that the first tool's data was really hierarchical: each entity contained an attribute (a string), whose value indicated the position of the entity in a hierarchy. That is, the attribute value was used to encode a hierarchical relationship. This lead to the conclusion that the full power of a programming language is beneficial in most practical situations.

However, manually writing a translator from scratch can still be a daunting task. A practical engineering solution was chosen to help in this situation. In previous work on the specification of model interpreters [5], a language was defined to specify the actions of a translator. A corresponding generator tool now exists which supports the rapid construction of the translators using this language.

Lieberherr was one of the first to propose a navigational language to specify the traversal/visitor actions to be performed on an object structure [6]. A more recent work that is similar to our approach can be found in [7]. Their work is focused more on traversing structures that are expressed at a lower level (e.g., object structures in a particular programming language). Our work is focused on the traversal of higher level modeling structures that are specified in our modeling language notation.

Our approach is based on a variant of the *Traversal/Visitor* pattern [4]. The translator begins at a root point that represents a model. As it performs the traversal, possibly in multiple passes, it executes "actions". Actions can generate an output object, change an output object, etc. During the traversal process one can also pass along shared data structures that serve as a kind of context. These context variables are used to store state information during the traversal.

The actual form of the specification contains two parts: the traversal specification and visitor specification. *Traversal specifications* answer the following question: "if we are at node of type X, where do we go next?" The "next" should be an object that is reachable from objects of type X. *Visitor specifications* capture what should be done when visiting a particular kind of object. There are two options: either execute a "user action" (i.e. execute a piece of user-supplied code), or proceed with the traversal (i.e. call the traverser with the object being visited). These can be intermixed and/or omitted completely. Note that the specification has an outer, high-level language for describing the structure, while the inner parts are written in a procedural language – C++ in our case. The generator translates the above mixed form specification into straight C++ while building the code sequences for the traversal and iterative parts during the

process. The resulting translator code then is linked with the rest of the IMS framework.

Below is an example of a traversal specification. The name of the traversal is TRV and it makes use of a visitor named VIS. In particular, the example specifies the traversal sequence for a Primitve node. Notice that the traversal sequence is only relevant for a specific phase. Phases are user-defined names that signify various passes over the data structure representing the model. The sequence of nodes to traverse is specified in the list of names following the reserved word "to." Optionally, C++ code can be embedded at various points in the traversal sequence. The C++ code is surrounded by the << .. >> delimiter. A similar syntax is available for the specification of explicit actions for each visitor node.

```
traversal TRV using VIS {
  from Primitive[prim_pars]
    { in phase }
    << init_action >>
    to {<< pre_in_action>>
        inputs[in_pars]
        << post_in_action>>
    }
```

We have found that writing translators using the traversal/visitor approach is very convenient, because the uninteresting parts (pointer tracking, iteration, selection next steps), are automatically taken care of by the generator. For a specific example of the traversal/visitor approach, see [5].

Before a model is inserted into the IMS, we want to ensure that the data is compliant with the constraints of the tool data model. After the semantic translators have completed, it is possible to verify that certain invariant constraints were preserved during the translation. We use a derivative of the Object Constraint Language (OCL) to capture the static semantics of the data model [11]. After parsing the OCL constraints, we generate C++ procedures that "evaluate" the expression in the context of the result of the translation. If a constraint is not satisfied, an error is raised and the data is not inserted into the server. On Figure 4, the box labeled "Constraint enforcer" represents this function. As an example, the following expression would be used to represent the constraint that "all model names must be distinct from all entity names":

```
-- Models and Entities must have
-- different names
constraint UniqueNames(Model top) {

top->models->
      forAll(m | entities->
          forAll(e | e.Name <> m.Name))

}
```

With respect to tool adaptors, a model-integrated approach has also been used. A framework has been created that defines many reusable components that are helpful in writing tool adaptors. Much of the framework is focused on issues concerning the CMI. A generator has been created that builds "glue code" from the tool data model. This code serves as a wrapper around the CMI data structures and assists the adaptor writer by allowing access to CMI data structures by referencing concepts from within the tool's domain.

We are looking at opportunities to utilize other technologies to use in the data integration process. For example, we recently completed the development of a tool that allows a user to view the contents of an IMS model through a web browser. Our tool is a Java applet that connects to the IMS and allows the user to view both the meta- and instance data for a particular model. All of the information in the model can be visualized in a hierarchical tree control; see Figure 6. This tool performs its function by utilizing the CMI through CORBA. In the future, we hope to allow the user to edit/change a model from within this browser. Another obvious growth path for the approach is to make IMS data available in XML form.

In summary, the process of tool integration using the IMS relies heavily on the use of both data models and translation specifications in order to generate semantic translators. Obviously, as new tools are supported within the IMS, the integrated data model will also need to evolve. This is often a trivial process and usually allows the existing translators to remain unchanged.

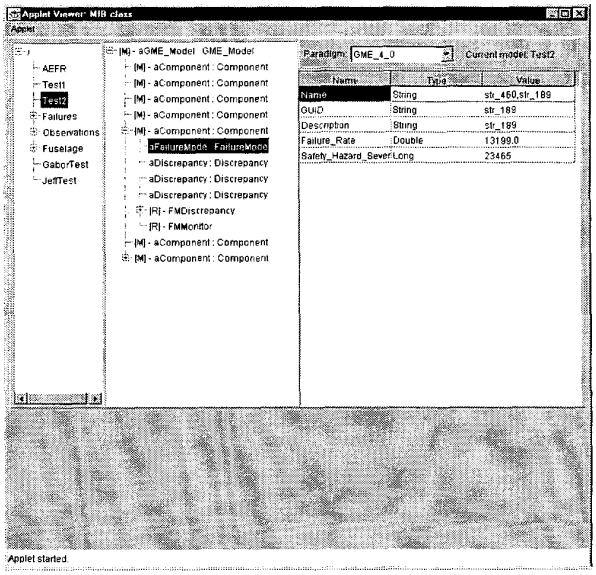

Figure 6 IMS Model Browser

Thus, adding a new tool to the integration process involves the following procedure:

1. The data model for the tool must be described using the notation that is recognized by the various generators.
2. Additions to the integrated model may be needed in order to represent data contained in the new tool.
3. The semantic translator has to be modeled and generated. The modeling involves the description of the translation process in terms of traversal/visitor specifications.
4. A tool adaptor must be constructed that reads/writes the native physical data format used by the tool. This requires an understanding of the physical model, whether it be a specific database, comma-separated text file, or even access using COM.

4. CONCLUSIONS

We have used the approach described in this paper to assist us in the task of tool integration. Our most recent project involved the integration of four different tools. The initial effort on this project was to understand the semantics of each tool and then formalize a representative data model. The second task was focused on the construction of the semantic translators. The average size of a translator was about 225 lines of traversal/visitor specifications and C++ code. The smallest translator required only 145 lines of specifications while the largest needed about 300 lines. After the translators were available, work then proceeded with the construction of tool adaptors.

There are several lessons that we have learned while working on this project:

- The fundamental principle of separation of concerns was found to be very powerful. Tool integration involves both semantic and syntactic transformations. It is conceptually cleaner to keep these issues separate.
- During the translation process, the complexity of the relationships of the underlying data may demand the capability to utilize the full power of a programming language.
- The evolvability and maintainability of a system is improved when a framework is used as the infrastructure for generating components from models.

Our experience has shown that this approach offers a new, yet feasible, solution toward integrating different types of engineering tools. With an architecture-centric focus, high-level models and code generators can be used to build integration solutions effectively.

5. ACKNOWLEDGMENTS

The DARPA/ITO EDCS program (F30602-96-2-0227), and The Boeing Company have supported the activities described in this paper.

6. REFERENCES

[1] Don Box, *Essential COM*, Addison-Wesley, 1998.

[2] *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, The Object Management Group, July 1995.

[3] Martin Fowler, *UML Distilled*, 2nd ed., Addison-Wesley, 1999.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[5] Gabor Karsai, "Structured Specification of Model Interpreters," in *Proceedings of International Conference on Engineering of Computer-Based Systems*, 1999, Nashville, TN.

[6] Karl Lieberherr, *Adaptive Object-Oriented Software*, International Thomson Publishing, 1996.

[7] Johan Ovlinger and Mitchell Wand, "A Language for Specifying Recursive Traversals of Object Structures," Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1999, Denver, CO.

[8] James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999

[9] Janos Sztipanovits and Gabor Karsai, "Model-Integrated Computing," *IEEE Computer*, pp. 110-112, April, 1997.

[10] Lois Wakeman and Jonathan Jowett, *PCTE: The Standard for Open Repositories*, Prentice Hall, 1993.

[11] Jos Warmer and Anneke Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.

7. BIOGRAPHY

Gabor Karsai is Associate Professor of Electrical and Computer Engineering at Vanderbilt University and co-director of the Institute for Integrated Information Systems. He has over twelve years of experience in software engineering. He conducts research in the design and implementation of advanced software systems for real-time, intelligent control systems, and in programming tools for building visual programming environments, and in the theory and practice of model-integrated computing. He received his BSc and MSc from the Technical University of Budapest, in 1982 and 1984, respectively, and his PhD from Vanderbilt University in 1988, all in electrical and computer engineering. He has published over 60 papers, and he is the co-author of four patents.

Jeff Gray received the BSc degree in computer science from West Virginia University in 1991 and the MSc degree in computer science from WVU in 1993. As a research assistant at ISIS, he is pursuing the PhD degree in computer science at Vanderbilt University. His interests are formal specification languages and aspect oriented programming. His current diversion is the creation of an extensive list of ambiguous/inconsistent statements (www.vuse.vanderbilt.edu/~jgray/ambig.html).