Tool Integration Patterns

Gabor Karsai[1], Andras Lang, Sandeep Neema
Institute for Software-Integrated Systems
PO.Box 1829B
Vanderbilt University
Nashville, TN 37235, USA

Abstract

Design patterns have been widely recognized as important contributors to the success of software systems. This paper introduces and compares two patterns that solve specific design tool integration problems. Both patterns have been implemented and used in real-life engineering processes.

Keywords*: design patterns, software architecture, tool integration framework, metamodels, generative programming.*

Introduction

The development of complex engineering artifacts requires a number of computer-based design tools. This is especially true for embedded system development, where both hardware and software aspects of the design has to be handled, as well as design analysis and synthesis, not to mention the ultimate system integration. It has been estimated [1] that in order to develop a new cell phone, about 50 design tools are needed.

Typically these design tools are not integrated, and there is a definite need for being able to share engineering artifacts across multiple tools. Occasionally, tool vendors create tool suites, like Rational Rose [2], but if a development process includes ingredients not supported by the elements of the tool suite, one faces the tool integration problem again.

In this paper we introduce two approaches for design tool integration that have been tried out in experimental systems. Both approaches have used a metamodel-based technique: the integration solutions were created through building metamodels of (1) the tools and (2) the transformations among models. The first, based on an integrated model showed the viability of the approach for engineering processes where the key issue was sharing, while the second, which was based on a process model showed good results for processes where the focus was on engineering process flows.

Backgrounds

Tool integration has been recognized as a key issue in complex, computer-supported engineering processes [3],[4], yet there are very few tangible results or products that could help end-users who need solutions for these problems. Integration of complex tools is difficult, labor intensive, and not always an intellectually rewarding activity.

Tool integration is especially relevant for the model-based development of embedded systems [5]. In a model-based development process, engineers work on and manipulate various kinds of models: requirement models, design models, analysis models, executable models, etc. which have to seamlessly "work" together. More precisely, changes made in one model should be "propagated" to other models, and *overall conceptual integrity* of the models must be maintained.

It is our belief, that design patterns [6] and software architectures [7] are key ingredients to solve tool integration problems. In fact, many previous proposals and efforts [3][4][8][9][11] have been advocating an architecture-based approach. The two approaches described below are based on two architectural design patterns [7], derived from slightly different requirements.

One key requirement in both of these systems was the need for model transformations. We have used a metamodel-based approach in both cases, not unlike the style advocated by OMG's Model-Driven Architecture [12]. The common framework for building transformation tools is illustrated on Figure 1.

[1] gabor.karsai@vanderbilt.edu, Tel: (1)(615)343-7471, Fax: (1)(615)343-7440. Other author's email address: andras.lang@vanderbilt.edu, sandeep.neema@vanderbilt.edu

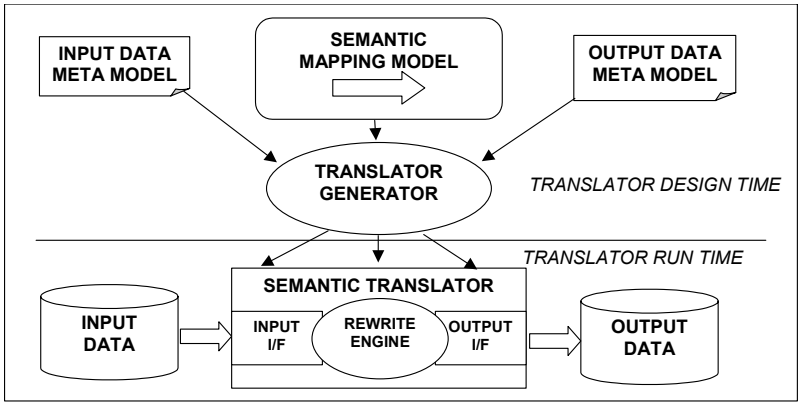

Figure 1: Metamodel-based model transformations

Whenever model transformations are needed, we create a metamodel for the input and the output models. Metamodels capture the abstract syntax and well-formedness rules (e.g. type constraints) of models. In addition, we create a metamodel for the semantic mapping that establishes the connection between the input and output domains. From these metamodels we synthesize (generate) a *semantic translator* that implements the model transformation. There are many implementations of this scheme, mostly distinguished by the methods and tools used for specifying the metamodels and the semantic mapping. We have used one [14] based on specifying the transformations with the help of the visitor pattern, and another one [15] based on graph transformation techniques.

Patterns for Tool Integration

In the following section, we describe two architectural patterns for tool integration. Both solutions provide a reusable *framework* for implementing tool integration solutions, so they are similar to other previous efforts, like Toolbus [10] and many others. The primary motivation for both approaches is the same: to facilitate tool data interchange. The secondary motivation was to provide a software infrastructure and (meta-level) tools to configure it in order to support a wide range of specific tool integration problems.

Specifically, the envisioned mode of operation for the tool integration is as follows. Individual engineers use their tools to create and/or modify "models": some of sort of design artifacts, and the primary repository for models is internal databases of the tools. However, models produced in one tool can be made available for use in other tools: the user of a "source" tool can publish the models for some "destination" tools. A tool integration solution (built from a generic framework) should provide all the support services to facilitate this sharing activity.

Integration based on Integrated Models

The motivating application for developing this framework came from an application domain: designing Prognostics and Health Management Systems (PHM) for aircraft. The PHM domain requires the use of many, widely different engineering tools: fault-modeling tools, diagnostics engines, FMECA[2] databases, and others. Each tool has a different function (design analysis, run-time diagnostics, data storage, etc.), but they are all related to a common physical artifact: the aircraft. The existence of the common physical artifact has a profound implication: there is significant overlap among the concepts and data elements used in the tools. We have created a tool integration solution that supports this overlap. The architecture has been reported in [13], here we briefly review its salient features. The architecture is shown Figure 3.

The architecture is based on the concept of an integrated data model (IDM). IDM is a metamodel that integrates the metamodels of the individual tools to be integrated. IDM is "rich enough" to represent models coming from any tools. Conceptually, IDM represents a set, whose elements are related to other sets representing the metamodels of tools through a bijection (Figure 2). Note that elements in the IDM have a corresponding element in at least one of the metamodels of the constituent tools, however there can be elements in the IDM that do not have an equivalent in some of the tools.

[2] FMECA: Failure Mode Effect and Criticality Analysis, a standard engineering technique used in complex, high-consequence engineering systems, like aircraft, space systems, nuclear power stations, etc.

The architecture contains two kinds of major components: the Integrated Model Server (IMS), and the Tool Adaptors (TA). The communication mechanism between the major components is implemented in CORBA (although any middleware package is suitable here). The functions of the components are as follows.

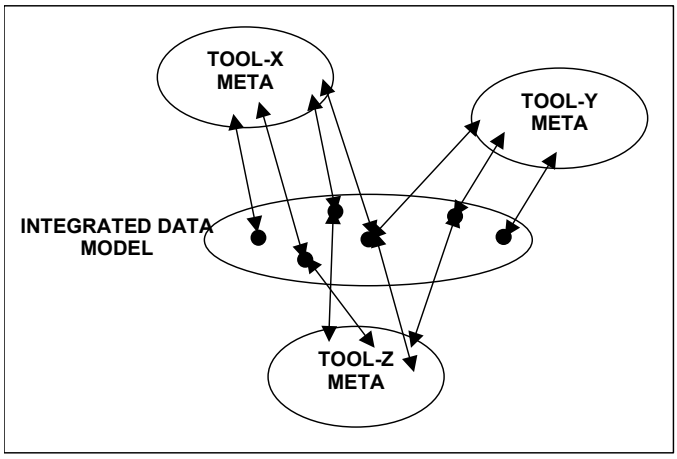

Figure 2: The concept of the Integrated Data Model

The IMS is responsible for providing *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.

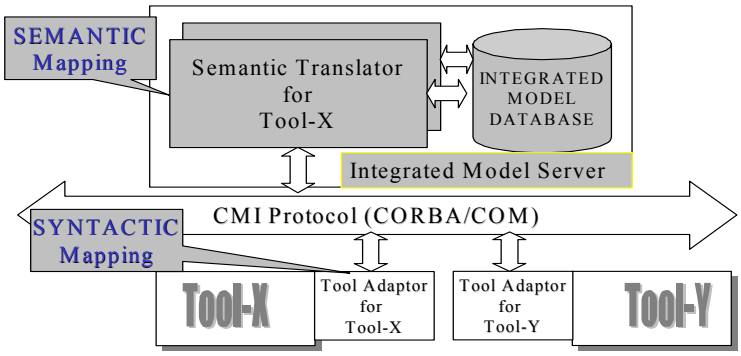

Figure 3: Tool Integration Architecture based on an Integrated Model

The TA-s are responsible for interfacing with the tool directly. Their goal is to read and write tool data, directly in the form the tool generates and expects it. The adaptors ship the data read to the IMS and receive data from IMS that they next send to the tool. The TA accesses the tool's data in whatever way it is possible and suitable: through a data file, a COM interface, or something else. Note that the TA performs a *syntactic translation* on the data from the native data format of the tool to that of the protocol used to communicate with the IMS.

In a general sense the architecture is used as follows. When a tool wants to make its data available for other tools, its TA is started. The TA fetches the data from the tool and converts it into the "network" format and ships it to the IMS. The IMS receives it, performs a semantic translation on it, and places the result into its repository. At this point the data is transformed into an IDM-compliant form. When another tool wants to use the data just translated, it accesses the IMS. The IMS performs a semantic translation on the data from the IDM-compliant data model into the tool-specific data model, and ships the result to the tool's TA. The TA will take the data in network form and convert it into the physical data format of the tool.

Note that the architecture separates the concerns of syntactic and semantic transformations, and assigns them to two different components: the TA-s and the IMS. This distinction makes the development of the integration solution easier. The binding between the major components is the middleware, specifically a protocol for data interchange.

Integration based on Process Flows

The motivating application for this tool integration solution came from a different domain: development of embedded software, in particular vehicle management applications that are part of an avionics suite. The engineering process identifies several contributors in the engineering process: (1) the component developer, who builds software components using standard CASE tool, like Rational Rose, (2) the system developer, who builds system configurations from predefined components using a domain-specific visual modeling language, (3) the analysis engineer who performs analyses on the design and verifies, for instance, schedulability using verification tools, and (4) the integrator and test engineer who actually builds the applications, runs them on the platform, and gathers test data. Like in the previous case, this process also had a profound implication on the solution architecture. Note that although there is a shared goal (producing an application), the individual players use different models: component models, system models, analysis models, executable models, etc. Therefore, in this architecture we did not use the integrated model concept, and realized a point-to-point integration instead. The resulting architecture is shown on [Figure 4].

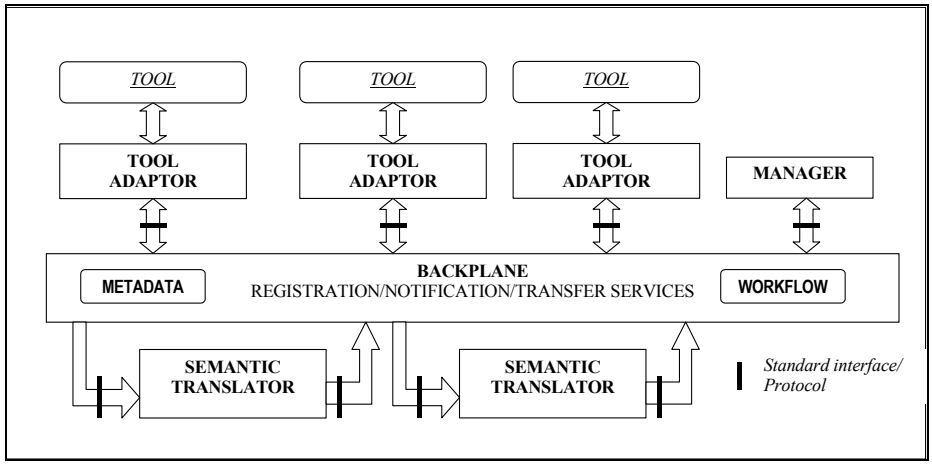

Figure 4: Tool Integration based on Process Flows

This architecture retains the concepts of TA-s and ST-s from the previous one, but the individual tools share data using a message-based approach: via a backplane component. The backplane provides routing services for shipping models from one tool to another, involving a semantic translation step if needed. The interface between the TA-s and ST-s is implemented using a middleware technology, but the backplane does not provide any kind of persistence services, as opposed to the previous case.
The architecture operates as follows. The backplane is initialized, and all the metamodels, translators and workflows are registered with it. Workflows represent which tools are publishers of and subscribers to what type of models, and how these tools are sequenced (as in a real workflow). Tools, when started, have to register themselves with the backplane. Whenever a tool wishes to make a model available to others, it invokes its tool adaptor, which then sends the model to the backplane. Based on the workflow specification, the backplane determines if there are registered consumer tools and what translation steps need to be executed to ship the (transformed) models to the consumer(s). It then invokes the appropriate translator(s). Whenever the translator finishes, the backplane routes the model to the consumer tool, which first gets a quick notification, and then, if the user chooses, can download the model. A manager tool is used to configure backplane and monitor its operation.

Common Extensions

Both of the above patterns allow further extensions and refinements. We discuss two issues in this section: incremental change propagation and traceability between tools.
As described above, the primary mode of operation is to share “models” across tools through a publish/fetch process, with semantic translations automatically inserted as needed. It is sort of implicit that we share entire models, however in many situations the propagation of incremental changes is

much more practical. Both of the architectures are suitable to implement a tool integration solution that supports this. The necessary refinements are as follows:
(1) The source TA has to be able to detect changes in the subject model, and express these changes in appropriate operations of the interaction protocol.
(2) The semantic translator has to be able to translate the changes in its input domain to changes in its output domain. This is perhaps the most difficult operation, and it may require access to the output data. Formally, the translator should not be a single-argument "function": $y = f(x)$, rather a two-argument function: $\Delta y = f(\Delta x, y_{old})$.
(3) The destination TA has to be able to update the output data with the "delta" received from the translator.
The approach based on the integrated data model is less suitable for supporting this change propagation (as it involves two translations), while the process flow based approach seems simpler.
Both frameworks are metamodel-based: they are configured through the use of metamodels. One has to create a metamodel for each tool to be integrated (plus the integrated data model for the first). When the transformations are also specified using a metamodel, one has an explicit representation of dependencies among the data elements in the various tools. The key here is that the transformations should be represented *explicitly*, and thus allowing traceability. One technique that allows this is based on graph transformations [15]. One can conceive a metaprogramming environment that is tailored for building tool integration solutions, through constructing the metamodels of tools and translations between tools, and which also supports the checking of metamodels with respect to completeness and consistency.

Comparison and Evaluation

The IDM approach assumes a significant overlap among the metamodels (i.e. the data models) of the individual tools, such that an IDM can be constructed and the mapping established. This approach works very well in cases when most of the valuable information is in the intersection set, i.e. there is a strong cohesion among the various models. The approach implements a full integration across N tools, using N (bi-directional) translators. The IDM is effectively a common, "universal language" that is used to interchange models.
The process-based approach does not assume any overlap and implements a pairwise integration among tools. This tool composition works well if the tools operate on different models, and tools distant in the tool chain are only very indirectly related. Although there is correlation between the models used in the tools, the cohesion is typically less than in the previous case. If there are N tools, typically there are <N, unidirectional translators.
Practical experience with the IDM approach showed that it becomes very complicated if the number of tools grows beyond three or four. To understand and maintain the mapping, where a change could have very serious consequences in four-five other places (translators, tool adaptors, etc.), is becoming an insurmountable task for an engineer. However, if the number of tools does not exceed three, the integration is very manageable.
The process-based approach does not have these shortcomings, as the changes are always localized. Changing a metamodel for a tool impacts only the translators that read and write models of that tool, but not others. This locality allows scaling to larger tool chains, and our experience with six tools shows that the approach is highly feasible. Interestingly, the process-based approach does not preclude the use of the IDM approach in a solution: one "merely" has to create a tool that acts as the integrated model server (IMS) —together with an appropriate translator.
Using a metamodel-based integration strategy enabled us the rapid construction of tool integration solutions by instantiating the framework from metamodels and using generative techniques [6]. We have devised a process for this instantiation that consists of the following steps: (1) identifying the tool chain elements and the workflow among these elements, (2) metamodeling of the tools, (3) metamodeling the semantic translations among the tools, and (4) developing the tool adaptors, and generating the semantic translators. This process enabled us to build and update instances of the framework with very manageable effort.

Summary and Future Work

We have shown two architectural patterns that specify frameworks for tool integration solutions. Both architectures are based on the principles of separating the syntactic and semantic transformations, and the use of metamodel-based techniques. The first architecture is based on an integrated model, but exhibits shortcomings with respect to scalability to larger tool chains. The second architecture is

based on a messaging system, which routes data according to a workflow specification, and implements a pairwise integration among tools.
The described solutions provide architectures that solve mainly the data integration problem. The implementation of the control integration among tools is subject to future work. The TA-s are currently hand-coded, and using metamodels and generative techniques for implementing them is another area of further work. As it was pointed out above, the architectures allow incremental propagation of changes in the models, but we have not built the supporting infrastructure for that yet. Finally, in geographically distributed tool integration scenarios, there is a need for a web-based backbone for integrating (localized) tool integration framework. We plan to address the issues of web-based frameworks in the future as well.

Acknowledgement

The Boeing Company and the NSF ITR on "Foundations of Hybrid and Embedded Software Systems" have supported, in part, the activities described in this paper. The effort was also sponsored by DARPA, Air Force Research Laboratory, USAF, under agreement number F30602-00-1-0580.The US Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright thereon. The views and conclusions contained therein are those of authors and should not be interpreted as necessarily representing the official policies and endorsements, either expressed or implied, of the DARPA, the AFRL or the US Government.

References

1. Personal communication with engineers from a world-leader telecommunication company.
2. http://www.rational.com
3. ECMA TR/55: Reference Model for Software Engineering Environments, NIST Spec. Pub 500-211.
4. PCTE Standard: ISO/IEC 13719.
5. Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty: “Model-Integrated Development of Embedded Software”, IEEE Proceedings Special Issue on Embedded Systems, January, 2003.
6. E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*, Addison-Wesley, 1995.
7. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-oriented Software Architecture: A System of Patterns. John Wiley & Sons, 1996.
8. EDA: http://members.tripod.com/~encapsulate/thesis.html
9. ETI: http://www.eti-service.org/
10. J. Bergstra and P. Klint. The discrete time ToolBus: A software coordination architecture. Science of Computer Programming, 31(2-3):205 229, July 1998.
11. E. W. Karlsen. The UniForM WorkBench --- a higher order tool integration framework. In International Workshop on Current Trends in Applied Formal Methods, October 1998.
12. http://www.omg.org/mda
13. Karsai, G, Gray, J.: Design Tool Integration: An Exercise in Semantic Interoperability, Proceedings of the IEEE Engineering of Computer Based Systems, Edinburgh, UK, March, 2000.
14. Karsai, G.: Structured Specification of Model Interpreters, in Proc. of International Conference on Engineering of Computer-Based Systems, 1999., Nashville, TN.
15. Agarwal, A., Karsai, G., Shi. Feng: Graph Transformations on Domain-Specific Models, in review for publication in “Journal of Software and Systems Modeling”.
16. Czarnecki, K. Eisenecker, U: *Generative Programming - Methods, Tools, and Applications,* Addison-Wesley, 2000.
17. Richard P. Gabriel. Patterns of Software: tales from the software community. Oxford University Press. 1996.
18. http://www.isis.vanderbilt.edu/Projects/mobies/default.html