AFRL-IF-RS-TR-2006-189
Final Technical Report
May 2006

WEB-BASED OPEN TOOL INTEGRATION FRAMEWORK

Vanderbilt University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. N892/00

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2006-189 has been reviewed and is approved for publication

APPROVED: /s/

ROGER J. DZIEGIEL, Jr.
Project Engineer

FOR THE DIRECTOR: /s/

JOSEPH CAMERA, Chief
Information & Intelligence Exploitation Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MAY 2006	Final	Aug 02 – Dec 05

4. TITLE AND SUBTITLE

WEB-BASED OPEN TOOL INTEGRATION FRAMEWORK

5a. CONTRACT NUMBER

5b. GRANT NUMBER
F30602-02-2-0202

5c. PROGRAM ELEMENT NUMBER
62302E

6. AUTHOR(S)

G. Karsai

5d. PROJECT NUMBER
MOBI

5e. TASK NUMBER
00

5f. WORK UNIT NUMBER
04

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Vanderbilt University
Div. Spons. Research, Station B#357749
Nashville TN 37235

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DARPA
3701 N. Fairfax Dr
Arlington VA 22203-1714

AFRL/IFED
525 Brooks Rd
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2006-189

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; distribution unlimited. PA# 06-375

13. SUPPLEMENTARY NOTES

AFRL Project Engineer: Roger J. Dziegiel, Jr., IFED, Roger.Dziegiel@rl.af.mil

14. ABSTRACT

The OTIF project described in this report addressed the problem of building integrated design tool chains for embedded system development. The project has developed, implemented, and applied an open tool integration framework that provides a software infrastructure for building specific tool integration solutions. The framework is based on reusable components and industry-standard protocols, and uses metamodeling and model transformation technology to facilitate the tool integration task. The report summarizes the technological contributions of the project, and the actual prototype tool chains constructed.

15. SUBJECT TERMS

Design tool integration, embedded system design tool chains

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Roger J. Dziegiel, Jr.
U	U	U	UL	48	19b. TELEPONE NUMBER (*Include area code*)

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI-Std Z39-18

Table of Contents

Table of Contents i
List of Figures ii
Glossary iii
1. Introduction 1
2. Project Overview 2
3. Results 4
3.1 Core Technology Results: OTIF 4
3.2 Application Domain Results: Toolchains 7
3.3 Further extensions 11
4. Summary 14
Publications 14
Appendix: Major papers 15

List of Figures

Figure 1: The Open Tool Integration Framework Architecture 3
Figure 2: MCP Toolchain 8
Figure 3: The VCP Toolchain 9
Figure 4: The SPP Toolchain 10
Figure 5: Tool Integration for the BioComp toolchains 11

Glossary

AIF	Analysis Interchange Format. An XML file format used to represent models of real-time embedded systems in a manner suitable for architectural analysis.\
AIRES	Real-time system analysis tool developed by Prof. Kang Shin at University of Michigan.
API	Application Programming Interface.
DESERT	Design-Space Exploration Tool. A metaprogrammable software tool that supports the constraint-based exploration of design variants.
ECSL	Embedded Control System Language. A modeling language for constructing embedded controllers for automotive applications.
ECSL/GME	The instance of the GME editor that supports the ECSL.
ESML	Embedded System Modeling Language. A modeling language designed for modeling mission computing applications built using the Bold Stroke framework of Boeing.
ESML/GME	The instance of the GME editor that supports the ESML.
Giotto	A time-triggered coordination language developed by Prof. T. Henzinger of UC Berkeley for implementing time-triggered systems on conventional real-time operating systems.
GReAT	Graph Rewriting and Transformations. A language and toolsuite for constructing model transformation programs.
GME	Generic Modeling Environment. A metaprogrammable visual model editor.
MCP	Mission Computing Platform. A prototype toolchain built using OTIF.
NCA	Network Connectivity Analysis tool that processes information captured in gene/transcription factor maps. Used in systems biology.
OEP	Open Experimental Platform. A software infrastructure that implements parts of the Bold Stroke framework of Boeing, used in the “Model-based Integration of Embedded Systems” program of DARPA
OSEK	OSEK is an abbreviation for the German term "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug" (English: Open Systems and the Corresponding Interfaces for Automotive Electronics). It is a standards body that has produced specifications for an embedded operating system, a communications stack, and a network management protocol for automotive embedded systems
OSEK/OIL	A configuration language for OSEK software applications.
OTIF	Open Tool Integration Framework.
PAINT	A clustering analysis tool that produces gene/transcription factor maps. Used in systems biology.
Ptolemy	A modeling and simulation environment developed by Prof. Ed Lee of UC Berkeley for studying models of computations for embedded system design.

SBML	Systems Biology Markup Language. An XML-based markup language for systems biology modeling.
SPML	Signal Processing Modeling Language. A modeling language for constructing high-performance, embedded signal processing applications.
SPP	Signal Processing Platform. A prototype toolchain built using OTIF.
SPML/GME	The instance of the GME editor that supports the SPML.
TNA	Task Network Architecture. A format for representing complex task structures for use in sequencing operations in a complex mission computing application.
UDM	Unified Data Model. A software package that generates C++ and Java API-s from UML metamodels, that could be used to access models stored in GME, in XML files, or as CORBA structures.
UML	Unified Modeling Language. A modeling language for modeling in object-oriented analysis and software design.
VCP	Vehicle Control Platform. A prototype toolchain built using OTIF.
XML	Extensible Markup Language.
XSLT	XML Stylesheet Language for Translations. XML-based scripting language for describing simple transformations on XML data.

1. Introduction

The objective of this project was to investigate how a Web-based Open Tool Integration Framework (OTIF) can be constructed, to design and construct a prototype framework, and to demonstrate how it can be used in various software development tasks, with emphasis on embedded software systems. The need for an OTIF is motivated by the fact the complex software development processes necessitate customized toolchains where design tools interoperate in a seamless manner. This is especially true for embedded software, where requirement capture, design modeling, design analysis, code generation, testing and debugging, etc. are often done with different tools, provided by different vendors. The use of different tools requires an open approach to tool integration: one that does not limit the integration of new tools. Commercial vendors are not motivated to build such toolchains rather they are interested in "locking-in" customer into their own tool infrastructures.

The project has created an architecture-based solution, called OTIF (to be discussed in the chapters below), which relies on industry standards and technology developed under a related project titled "Model-Based Synthesis of Generators for Embedded Systems".

The specific industry standard OTIF relies on is CORBA and its basic services for remote object invocation, the Naming service. These are well-defined and documented standards, with multiple commercial and open source implementations. However, in the high-level design of OTIF there are no CORBA-specific choices, and all CORBA-specific components are well isolated in the implementation. Specifically, porting OTIF to another middleware standard, like .NET, is quite feasible.

The specific technology OTIF uses from the related project is the UDM (Universal Data Model) tool for model representation and access, and the GReAT (Graph Rewriting And Transformations) tool for implementing complex model transformations. Similarly to CORBA, the GReAT/UDM specific components are well-isolated, and, if necessary, can be replaced with other model transformation technology, while keeping the rest of the architecture intact.

During the course of the project, we have designed and implemented a prototype for OTIF, and used that prototype for building toolchains for the MoBIES program. These toolchains have been used in actual MoBIES tool evaluations, and later has been transition to the ESCHER effort. Both OTIF and the toolchains are available via the ESCHER[1] website at www.escherinsitute.org. In the later phase of the project we have built extensions to OTIF (including a Java implementation of UDM and support for XML namespaces), that were tested and used in building toolchains for the BioCOMP DARPA effort.

[1] ESCHER is an independent, non-profit research institute dedicated to the transition of government-sponsored information-technology out of the research environment and into practical use by industrial and government end users.

2. Project Overview

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".

Our solution, 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.

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 was developed for OTIF (see Figure 1 below).

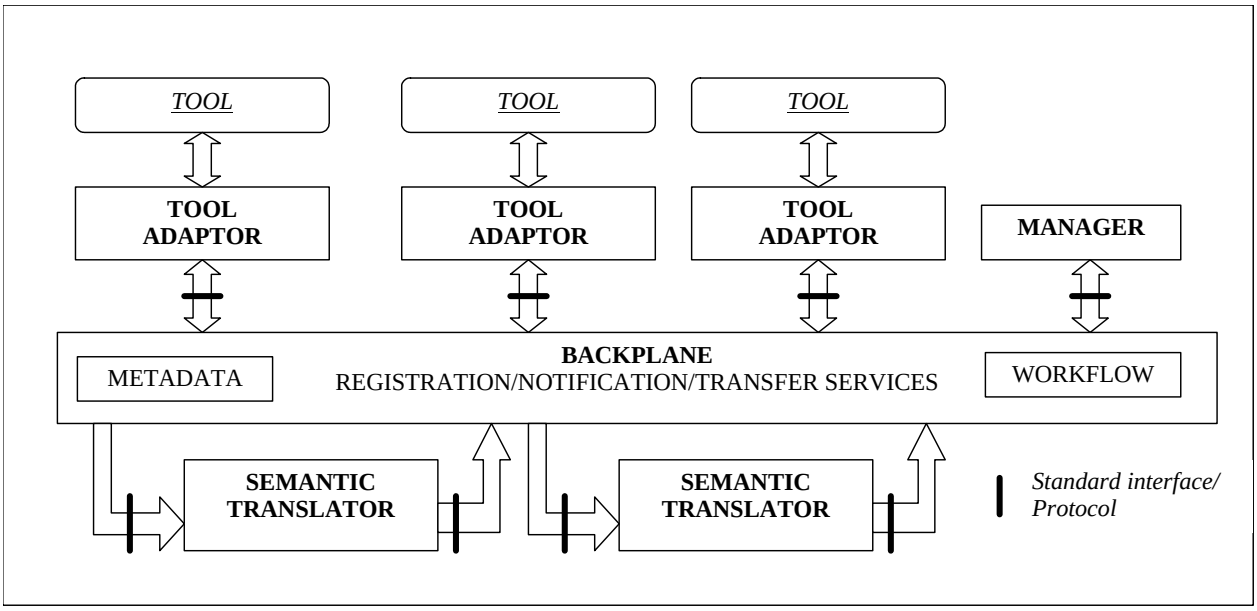

Figure 1: The Open Tool Integration Framework Architecture

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 with 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.
    - Workflow: Workflow models are loaded into the backplane and are used to facilitate the data transfer between tools and translators. Specifically, the backplane uses these models how to route the data (i.e. the models) among the different components.
    - Metadata: The metadata comprises the metamodels of all the tools registered with the backplane. These metamodels are loaded at initialization time and used for consistent labeling of model elements across the tools.
- 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 –perhaps quite complex— mappings. For the actual implementation of the semantic translators, we have utilized the results from another research project: the GReAT tool and framework for implementing complex transformations.

3. Results

In this section we will summarize the project results. The Open Tool Integration Framework developed is available for download from the ISIS website: http://escher.isis.vanderbilt.edu, together with some of the prototype toolchains we have constructed. This website is also accessible via the ESCHER website mentioned earlier.

3.1 Core Technology Results: OTIF

The architecture introduced above has been implemented, and it supports a specific tool integration scenario outlined below. We call this “batch-oriented pipelining of tool data files”. The user of a producer tool finishes the work that produces a new dataset. They then invoke a tool adaptor and uses that to send the data to the backplane. The corresponding 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.

The architecture introduced above is generic, and it is 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.
(5) 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.
(6) Developing the tool adaptors for the tools.
(7) 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.
(8) 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.

The OTIF architecture is centered on 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. For details, please see the OTIF documentation included in the software distribution.

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

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 exchanging the actual model 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 its type is).

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 could observe, and modify if necessary.

The OTIF Tool Adaptor group consists of the portion of the protocol, which deals with the interaction between the Tool Adaptor (T/A) and the backplane. When a T/A is started, it must log on to the backplane. During logon it has to identify 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 log off 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 has 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 documents are sent to and received from the backplane in the form of UDM instance data structures, where each data element is tagged with the corresponding metamodel elements present in the backplane. As the backplane is the ultimate holder of the metamodels, these tags are always unique, for all parties in the architecture.

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.

3.2 Application Domain Results: Toolchains

We have instantiated the OTIF architecture for a number of toolchains that were used in the course of the MoBIES Program. Here, we briefly summarize the toolchains.

3.2.1 Mission Computing Platform (MCP) Toolchain

This toolchain was designed for assisting the development of large-scale (>1,000 components), distributed (1-4 CPUs), (soft) real-time embedded system applications, like the ones built using Boeing’s Bold Stroke framework. The toolchain is illustrated in the figure below.

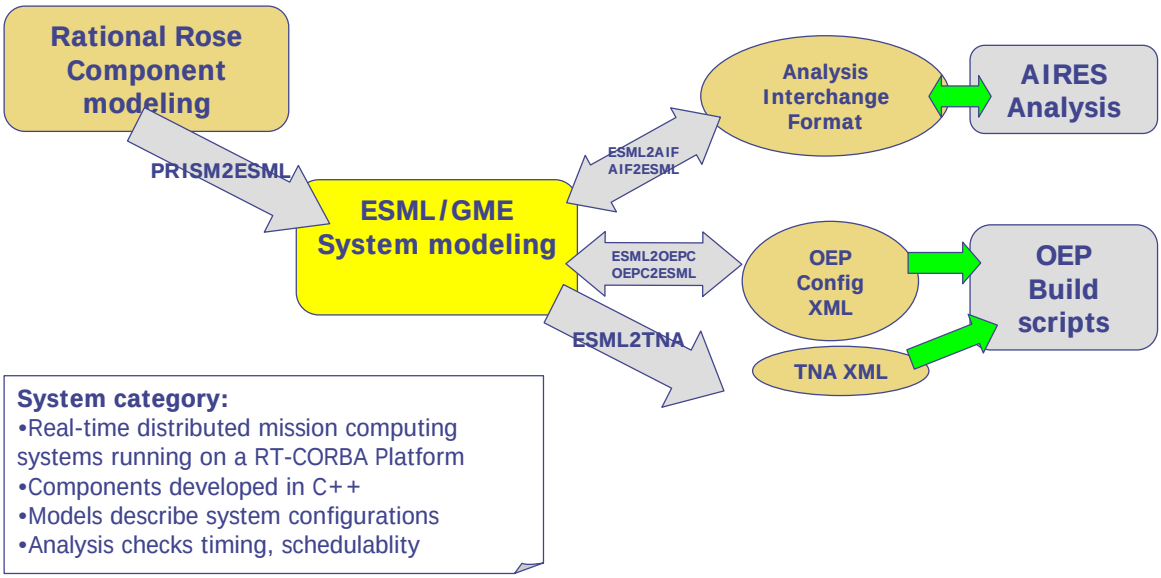

Figure 2: MCP Toolchain

The toolchain integrates Rational Rose with a GME-based modeling environment (ESML/GME), an analysis tool (AIRES), and the build process tools (supplied by Boeing). In the MCP development process, component design and modeling happens using the Rational Rose toolset. These component models are then imported into the ESML/GME environment, where engineers can perform the system-level modeling and architecting by specifying component interactions, component deployment, and task networks representing specific operational scenarios. Component assemblies augmented with deployment information can be converted into an analysis format (AIF), which is consumed by the analysis tool AIRES. The tool performs schedulability and other analysis on the models. In case of timing violations, it could generate alternative deployment plans that satisfy the timing; these are sent back to the modeling tool. From the system-level models two XML files could be generated (and subsequently used in the automated build process): the OEP Configuration XML, and the TNA XML. The former is used to generate all the code needed to instantiate and link the components in the system, while the latter is used to configure the run-time task network engine. These XML files directly interface with the Bold Stroke build process and tools.

3.2.2. Vehicle Control Platform (VCP) Toolchain

This toolchain was designed to assist in the development of code for embedded controllers in automotive applications. The toolchain is illustrated in the figure below.

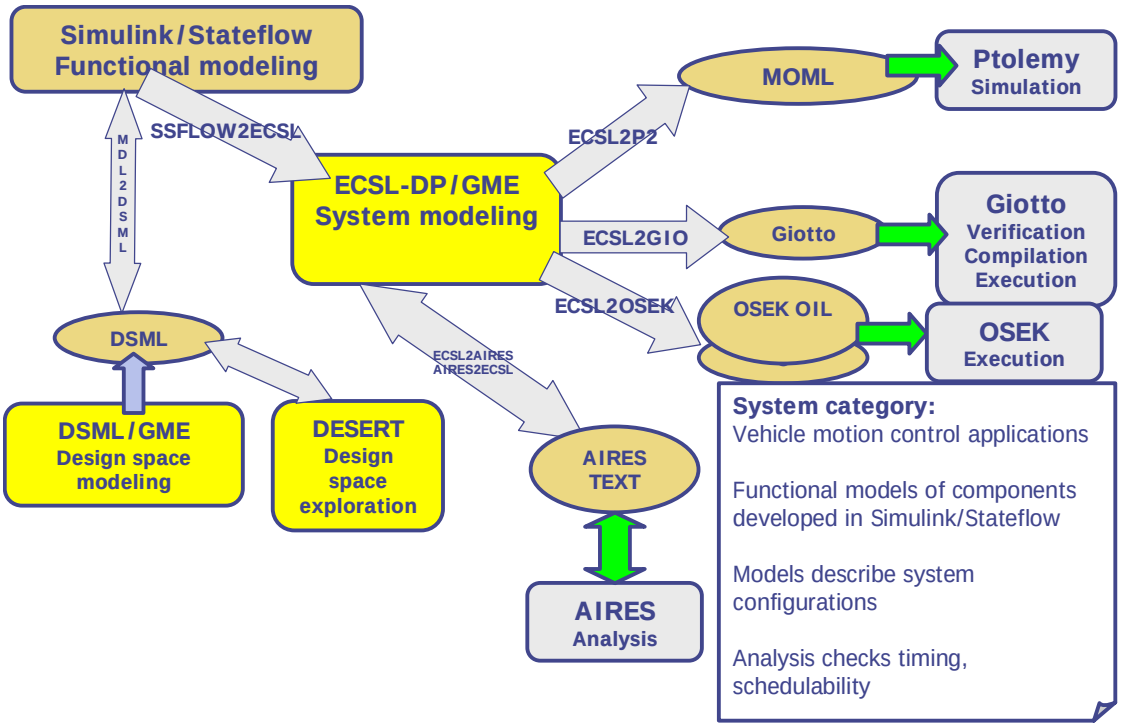

Figure 3: The VCP Toolchain

In this toolchain, the process starts with functional modeling, performed using Simulink/Stateflow. The designers can perform design space exploration on these models, using the DESERT tool. Once the functional models are ready, they can be imported into a GME-based environment (ECSL-DP/GME), which adds new modeling aspects to the ones available in Simulink/Stateflow. These new modeling aspects allow component modeling (i.e. which functional model blocks form a component), hardware platform modeling (i.e. what electronic control units will host the components, what communication links are available, etc.), and deployment modeling (i.e. how software components and links map to hardware components and links). From the system level models, a number of artifacts can be generated, including (a) simulation models (for simulating the system in Ptolemy), (b) Giotto code (for executing the models as a Giotto program), (c) analysis models (for schedulability analysis using AIRES), and (d) C code and OSEK OIL files (for compiling, linking, and deploying the controllers on actual hardware). In the final, "production" version of the toolchain (c) and (d) are fully supported.

3.2.3. The Signal Processing Platform (SPP) Toolchain

The purpose of this toolchain was to assist in the design of high-performance embedded real-time signal processing applications, typically found in video-driven missile guidance systems and automatic target recognition systems. The toolchain is illustrated below.

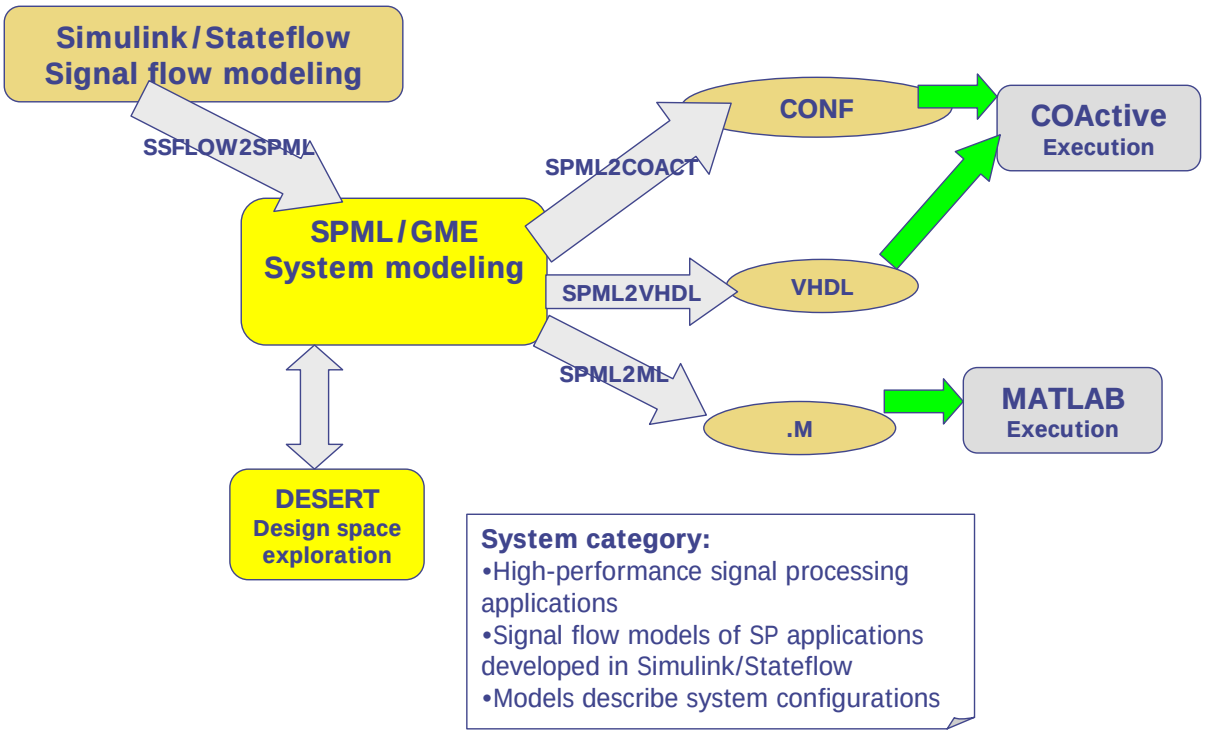

Figure 4: The SPP Toolchain

In this toolchain, the engineering process starts with signal flow modeling, done in Simulink/Stateflow. These signal flow models are then imported into a GME-based environment (SPML/GME), where they could be annotated and extended with system-level and deployment-specific information. The SPML modeling environment can also be used in conjunction with the DESERT design space exploration tool. From the SPML environment, one can generate configuration, C source code, and VHDL source code files for deployment (execution on the COActive execution platform), and Matlab script files for simulated execution.

3.2.4 Toolchains for the BioComp Program

Per request from the sponsoring agency, we have created tool integration prototype toolchains for supporting the BioComp program. We have created three, proof-of-concept prototypes using the OTIF and the translator technology that have been delivered and demonstrated. The prototypes are shown on the figure below.

The first integration solution — (a) on the figure — involved the PAINT tool (from T Jefferson University) and the NCA tool from UCLA. We have created a translator that connected the two tools, to form a simple toolchain. The PAINT tool generates its gene transcription factor association data as an SBML document, while the NCA tool expects the association information as a MathML document. Utilized the SBML, and MathML meta-models listed above, we have developed a model transformer using our graph-rewriting tool GReAT. This transformer has been packaged as an XML wrapped analyzer for the Dashboard platform.

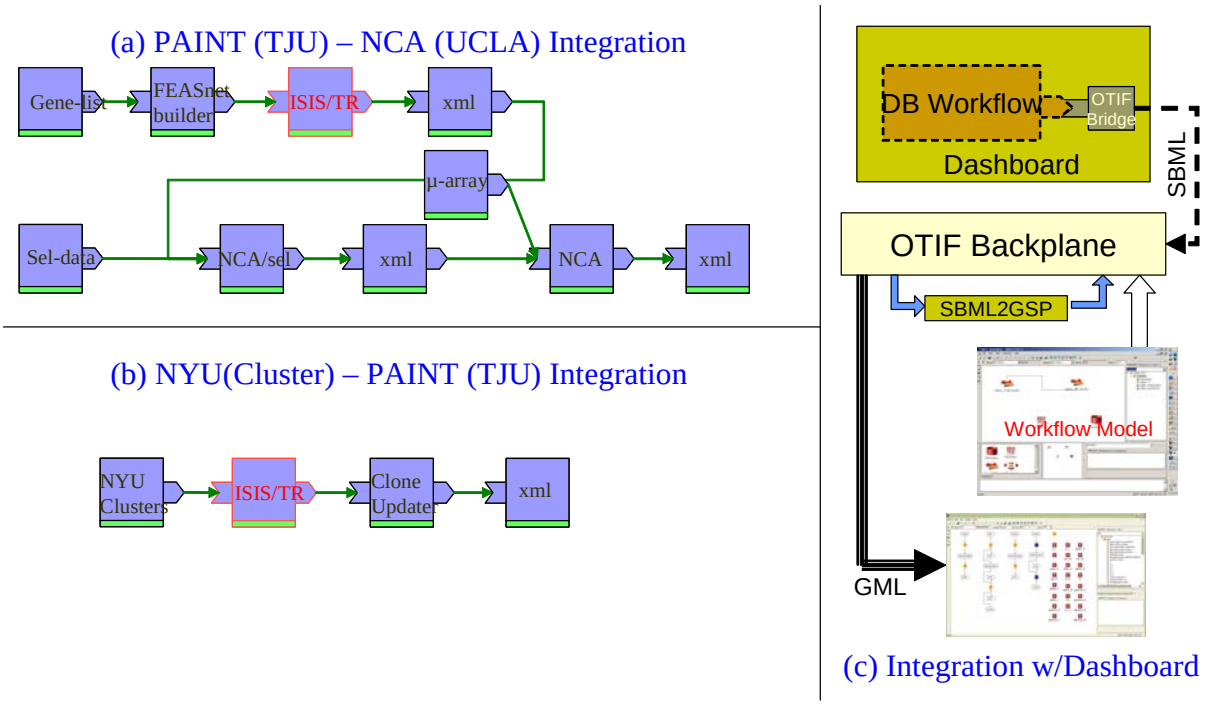

Figure 5: Tool Integration for the BioComp toolchains

The second integration solution — (b) on the figure— included a gene clustering tool from New York University and the PAINT tool from TJU. Similarly to the previous one, we have built a semantics translator using the same technology as the one used in OTIF. For the (a) and (b) cases we have used the Dashboard infrastructure for deploying the toolchain for the reason that researchers on the program were already experienced with it (although the OTIF backplane, etc. could have been used as well).

As a third example — (c) on the figure— we have implemented a bridge between OTIF and Dashboard, using the SBML (Systems Biology Modeling Language); the accepted common language for model interchange on the BioComp program.

Within a short period of starting transition activities to the BioCOMP program, we have been able to facilitate integration between two major tools (NCA tool from UCLA, and PAINT tool from TJU) in the program. The results have been a major value addition in terms of the ability of the NCA tool to process much larger data sets being generated by biologists in the program

3.3 Further extensions

After the first prototype of OTIF has been built, and the project has started using it in developing toolchains, a number of shortcomings were identified and extensions were introduced that allowed better tool integration. These extensions are summarized below.

3.3.1 Binary Large Objects

In many tool integration scenarios, models include binary data which does not lend itself to the structured representation and storage imposed by the UDM approach used in OTIF. For the

efficient exchange of data of such type we have made provisions in the protocols (more precisely: in the instance data formats of UDM), such that binary data can be incorporated into the message sent to and received from the backplane. However, care has to be exercised as the framework does not interpret and does not manipulate such binary data, it simply passes it through. Specifically, if the byte order of the sending and receiving parties is different, the appropriate by swapping must be performed by one of the parties.

3.3.2 Workflow modeling and engine

In the first implementation of OTIF the association between the senders and recipients was based on the type of the tool (i.e. the metadata). However, this approach is not efficient and is incorrect where tools consume and produce data of the same kind. Hence, we have introduced a workflow engine into the backplane of the architecture.

The workflow engine operates on a workflow model. The workflow model is created by a GME-based modeling tool that allows capturing the possible workflows among the tools in a flow-diagram. These GME models are compiled into an XML which is then loaded into the backplane using the manager tool.

The backplane's workflow engine works as follows. Whenever a document is received from tool, the engine looks up in the workflow model which translators could be applied to this document, and sends it to those translators. The result of the translation is similarly compared to the workflow model, which is now used to determine the destination tool (adaptor) for the document.

The workflow extension left most of the protocols unchanged; only the manager protocol group had to be extended.

3.3.3 Multi-input translators

In some tool integration problems we have identified the need for semantic translators that receive input from multiple sources. One such example was in a version of the MCP toolchain (described in the MoBIES final report), where the AIF files need to be updated with information collected from the execution environment. This required a translator that consumed one AIF file and one XML log file, and produced a new AIF file. To support this, we have extended the protocols, such that a (fixed-size) group of documents could be sent to and received from a semantic translator.

3.3.4 Change propagation and stateful translators

Again, in some applications we have recognized the need for translators that perform incremental translations on documents (i.e. models). Incremental translation means that the result of the translation depends on the result of a previous translation plus some new information. This means that the translators must be "stateful", i.e. must store results of previous translation activities as needed.

We support this behavior by allowing versioning the documents. When a tool adaptor uploads a document, it can designate the document as a new version of an existing one, such that a (stateful) translator can identify the previous version, etc. and perform the incremental translator. This solution was chose because the exact semantics of “versions” and “changes” is highly tool dependent, and a generic framework must remain neutral with respect to these details.

3.3.5 Web-based interface to the framework

We have built a C++ library for constructing tool adaptors, and we have also constructed a “Generic Tool Adaptor” that can generate tool data files that follow the UDM data representation techniques (i.e. XML files and GME project files). We have also created a version of the Generic Tool Adaptor (GTA) that integrates with the Eclipse framework, such that an Eclipse tool is available for uploading and downloading documents to and from the backplane.

To make the technology more generic and widely usable, we have created a web-based, generic tool adaptor. This was implemented as follows. The machine hosting the backplane runs a simple web-server that runs the generic tool adaptor code. The web server provides an interface to the GTA using standard HTTP. Any web browser can interact with the web server, allowing the upload and download of documents into the framework running under the control of the backplane. We created a prototype web-page as the interface, but it could be easily customized to support arbitrary user interface concepts.

3.3.6 Namespace support for UDM

In the course the project, when we created tool chains for the BioComp research activities, we were asked to introduce support for namespaces in XML files. Many BioComp tools rely on the use of XML namespaces, and thus the UDM libraries that read and write XML files had to support them. These extensions were incorporated into the UDM design, such that XML files produced by other BioComp researchers could be used directly as documents in the toolchains. Minimal extensions were made to the OTIF protocols in order to support these.

3.3.7 Java interface to UDM

Also in the course of the developing BioComp toolchains, we realized the need for Java access to UDM data. For this reason, we have created a Java interface as part of the UDM package. The Java interface consists of the following elements:

- UDM compiler/Java generator. The UDM compiler can be instructed now to generate Java source code for accessing the UDM objects. The generated code consists of class definitions for all the classes described in the UML models, similarly to the generated code in C++.
- UDM/Java interface libraries. These libraries provide the generic, Java implementation of core UDM classes, and link to the C++ implementation libraries. The linkage to the C++ implementation code was accomplished via the Java Native Interface libraries and tools.

The implementation is a mixture of generated Java code, handwritten Java code, and generated Java/C++ interface code. This approach reuses the core UDM libraries (hence only one version

should be maintained), and also provides high performance (that could be better than a pure Java-based approach). The Java interface has been successfully used in a number of BioComp tools.

4. Summary

The Open Tool Integration Framework described above has been fully implemented and used in constructing actual toolchains. These results were disseminated using the ESCHER organizational framework and its website. Starting from the original architecture, we have incrementally improved the framework, and used it in a number of toolchains of increasing sophistication.

There were two approaches for transitioning the technology developed in this project. One was the ESCHER organization, which directly resulted in toolchains used by Boeing (MCP), Raytheon (SPP), and GM (VCP). The other was the Model-Integrated Computing (MIC) Platform Special Interest Group (PSIG) of the Object-Management Group (OMG). The MIC PSIG has met several times in the past three years, and created a Request for Proposals document that invites standard proposals for an OTIF-like framework. This RFP is currently being discussed within OMG and is expected to be formally issued soon. For details on the PSIG, please see: http://mic.omg.org/. When that happens, our research team and industrial partners will submit a proposal based on our experience and results with OTIF.

Publications

1. Karsai G., Lang A., Neema S.: Tool Integration Patterns, Workshop on Tool Integration in System Development, ESEC/FSE, pp 33-38., Helsinki, Finland, September, 2003.
2. Karsai, G., Lang, A., Neema, S.: Design Patterns for Open Tool Integration, Vol 4. No1, DOI: 10.1007/s10270-004-0073-y, Journal of Software and System Modeling, 2004.
3. Karsai, G., Agrawal, A : Graph Transformations in OMG's Model-Driven Architecture: AGTIVE 2003, LNCS 2062. pp. 243-259.
4. Gabor Karsai: Tool Integration Aspects in the Model-Driven Architecture, presented at the 2004 Monterey Workshop on Software Engineering Tools: Compatibility and Integration, to appear in Lecture Notes on Computer Science, Springer.
5. Open Tool Integration Framework - Draft RFP, http://www.omg.org/cgi-bin/doc?mic/2004-08-01

Appendix: Major papers

Softw Syst Model (2005) 4: 157–170 / Digital Object Identifier (DOI) 10.1007/s10270-004-0073-y

Design patterns for open tool integration

Gabor Karsai, Andras Lang, Sandeep Neema

Institute for Software-Integrated Systems, Vanderbilt University, PO Box 1829B, Nashville, TN 37235, USA
e-mail: gabor.karsai@vanderbilt.edu

Published online: 10 November 2004 –

Abstract. Design tool integration is a highly relevant area of software engineering that can greatly improve the efficiency of development processes. Design patterns have been widely recognized as important contributors to the success of software systems. This paper describes and compares two large-grain, architectural design 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 have to be handled, as well as design analysis and synthesis, not to mention the ultimate system integration. It has been estimated (personal communication from the telecommunication industry) 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 [33], but if a development process includes ingredients not supported by the elements of the tool suite, one faces the tool integration problem again.

Even today, there are a large number of development tools available: requirement capture tools, design modeling tools, analysis tools, and tools that assist directly in the software development process: syntax directed editors, compilers, and debuggers. Still, these tools are often not integrated. State-of-the-art Integrated Development Environments (IDE) offer some integration, but the integration is so tight with a *particular* IDE that developers are forced to run multiple IDE-s open simultaneously, in order to utilize the individual tools [2]. Arguably, an *open tool integration* approach would remedy this situation.

By open tool integration we mean an approach that separates the tools to be integrated from the framework used to facilitate the integration. The framework, in fact, becomes a platform for integration, which provides generic, reusable machinery for building *tool integration solutions*: specific tool chains that support specific engineering process. The framework must be open and extensible, such that a wide variety of tools can be integrated. We believe that this approach offers a superior alternative to today's closed tool suites, typically provided by tool vendors.

In this paper we describe two architectural approaches: design patterns for design tool integration that have been tried out in experimental systems. These patterns have been employed in two, different frameworks, both of which have been used in building many, specific tool integration solutions. Both approaches used a metamodel-based technique: the actual 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 [5, 13],

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 [13]. 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 the *overall conceptual integrity* of the models must be maintained.

A recent effort in the industry: the Eclipse framework [5] introduced the concept of open tool integration to the desktop development environments. Eclipse is a platform for integrating software development tools, and it provides APIs for tool/platform coordination. However, there are three cases where it comes up short: (1) tools must share data through files (i.e. tools must be able to import/export files as needed), (2) tools must be under the control of the same desktop environment (i.e. there is no support for web-based, cooperative work), and (3) a tool must be adapted to the specific file formats used by other tools (i.e. there is no generic support for solving the data translation problem). However, Eclipse clearly indicates that (1) tool integration is a valid problem, and (2) generic, architectural solutions are viable.

Arguably, design patterns [13] and software architectures [5] are *the* key ingredients to solve tool integration problems. In fact, many previous proposals and efforts [5, 13, 15, 18] have been advocating an architecture-based approach. The two solutions described in this paper are based on two architectural design patterns (in the style of [5]), but they derived from slightly different requirements.

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 [3], ToolNet [2], 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 the internal database 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.

Common framework: Metamodel-based integration

Before discussing the architectural patterns, the common foundation for them should be introduced. Both approaches follow a metamodel-based technique. There are many different approaches for using metamodels in system development; we have used the method described in [13], which we summarize briefly here.

In this approach, every design artifact (requirement, design model, test, dataset, etc.) produced and used in the design process is expressed using the constructs of some Domain-Specific Modeling Language (DSML). We assume that each DSML is precisely and formally defined. Specifically, a DSML is a five-tuple of concrete syntax (C), abstract syntax (A), semantic domain (S) and semantic and syntactic mappings (M_S, and M_C) [9]:

$$L =< C, A, S, M_S, M_C >$$

The *concrete syntax* (C) defines the specific (textual or graphical) notation used to express models, which may be graphical, textual or a mixture of the two. The *abstract syntax* (A) defines the *entities (E), relationships (R),* and *integrity constraints (O)* available in the language (i.e. A is a tuple (E, R, O)). Thus, the abstract syntax determines all the (syntactically) correct "sentences" (in our case: models) that can be built. (It is important to note that the abstract syntax includes semantic elements as well. The integrity constraints, which define well-formedness rules for the models, are frequently called "static semantics".) The *semantic domain* (S) is usually defined by means of some mathematical formalism in terms of which the meaning of the models is explained. The mapping M_C: $A \rightarrow C$ assigns concrete syntactic constructs (graphical, textual or both) to the elements of the abstract syntax (i.e. it defines how an element of the abstract syntax is to be expressed in the concrete syntax). The semantic mapping M_S: $A \rightarrow S$ relates syntactic constructs to those of the semantic domain. The definition of the (DSM) language proceeds by constructing metamodels of the language (to cover A and C), and by constructing a metamodel for the semantics (to cover M_C and M_S).

Using the above definition, we say that the formally specified DSML is a *metamodel* of all the models that could be legally built in the language, and we solve the tool integration problem *in the context of the metamodels*. Note that the definition for metamodels here is slightly different from the usual definition for metamodels used in the MDA context [29]. In here, by *metamodel* we mean the definition of a modeling language, expressed in the form of UML class diagrams, while in MDA UML class diagrams are typically called *models*. This difference is

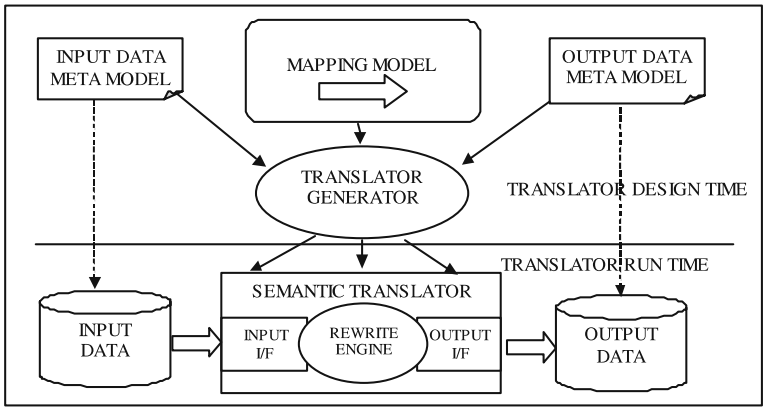

Fig. 1. Metamodel-based model transformations

a consequence of the fact that we have domain-specific modeling languages for creating the models, and not only UML is used. It is merely a convenience that we use UML to define the languages. However, in all cases the language to describe the language to define metamodels (i.e. the *meta-meta model)* is MOF, the Meta-Object Facility (MOF) [30].

In both architectural patterns described below there is a common notion of model transformations. In order to support semantic interoperability we must be able to interchange models across tools, which requires model transformations. In both approaches we have used a metamodel-based technique to describe the model transformations, not unlike the style advocated by OMG's Model-Driven Architecture [29]. The transformations are formally specified, in terms of the metamodels of the inputs and the outputs of the transformations, as illustrated on Fig. 1.

Wherever model transformations are needed, we formally specify the DSML for the input and the output by creating a metamodel for the input and the output models. Metamodels capture the abstract syntax and well-formedness rules of models. In addition, we create a model 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. We believe it is correct to call this translator a "semantic" one, as its goal is to facilitate semantic equivalence between the input and the output models. More precisely: if L (L') is the input (output) DSML, and m (m') is an input (output) of the translator, then $M_S(m)$ and $M'_S(m')$ must be equivalent. This equivalence is defined in terms of the respective semantic domains, one example being behavioral refinement, as discussed in [5].

There are many implementations of this metamodel-based model transformation scheme, mostly distinguished by the methods and tools used for specifying the metamodels and the semantic mapping. We have used one [21] based on specifying the transformations with the help of the visitor pattern, and another one [34] based on graph transformation techniques.

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[1] 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 and its components, their functions and failure modes, etc. The existence of the common physical artifact has a profound implication: there is significant overlap among the concepts used in the multitude of tools. This overlap motivated the creation of a tool integration solution, whose architecture has been reported in [21]. Here, we briefly review its salient features. The architecture is shown in Fig. 3.

The architecture is based on the concept of an integrated data model (IDM). Concepts used in the tools are represented by (one or more) metamodel elements, thus that overlap can be facilitated (1) by creating an additional metamodel: the IDM, and (2) by defining the mapping between elements of the metamodels of the tools and the IDM. Note that the IDM should be "rich enough" to represent models coming from any tools, and thus it is custom-made for a particular set of tools that are integrated. Conceptually, IDM is a set of metamodel elements that are related to the elements of the metamodels of tools through a mapping (Fig. 2). Note that elements in the IDM have a corresponding element in at least one of the metamodels of the constituent tools, there can be

[1] 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.

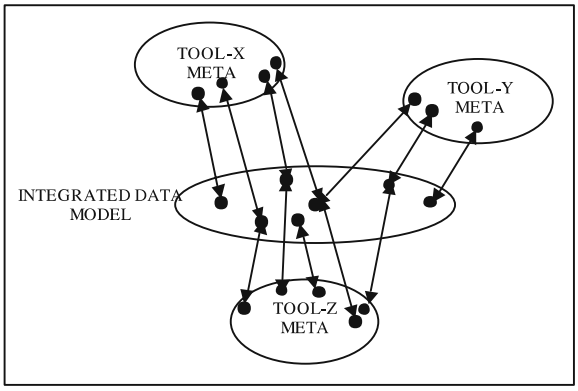

Fig. 2. The concept of the integrated data model

elements in the IDM that do not have an equivalent in some of the tools.

Formally, the IDM can be described using a DSML: L_{IDM}, which is constructed by composing the DSML-s L_i of the individual tools. We make the assumption here that the concrete syntax is irrelevant with respect to tool integration (e.g. there is no need for the visualization of models of IDM). A_{IDM} is constructed as the distinguished union of A_i-s, with respect to an equivalence operator $\approx$.

$$
\begin{aligned}
&A_{\mathrm{IDM}} = (E_{\mathrm{IDM}}, R_{\mathrm{IDM}}, O_{\mathrm{IDM}}), \text{ where}\\
&E_{\mathrm{IDM}} = \bigcup_{\approx} E_i\\
&\quad = \{e^p_{\mathrm{IDM}} \,|\exists_1 i, q : (e^p_{\mathrm{IDM}} \approx e^q_i, e^q_i \in E_i) \vee \exists i, j, q, r :\\
&\quad\quad (e^p_{\mathrm{IDM}} \approx e^q_i \wedge e^q_i \approx e^r_j \wedge i \neq j, e^q_i \in E_i, e^r_j \in E_j)\},
\end{aligned}
$$

and R_{IDM} and O_{IDM} are similarly defined. The definition means that the entities of the IDM are formed from the entities of the individual tools (E_i) by taking the union of those under a special equivalence operator $\approx$. The equivalence operator $\approx$ returns true if two entities (or relations or integrity constraints) are considered equivalent in some common semantic framework. An entity e^p_{IDM} is either equivalent to a single entity e^q_i from a single tool's E_i, or it is equivalent to an entity e^q_i from one tool's E_i which, in turn, is also equivalent to another entity e^r_j from another tool's E_j. Note that for O_{IDM} we require that the individual integrity constraints o_{IDM} do not conflict with each other.

Note that the IDM, as a DSML, has an abstract syntax, which is defined using UML class diagrams. In other words, the IDM has a metamodel, just like any DSML of a tool participating in the integration. The metamodels are "comparable", as they are all defined with UML class diagrams that share a common meta-metamodel: MOF.

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 IMS is responsible for (1) hosting *semantic translation* (ST) *services* for the constituent tools, and (2) providing *model storage services* (according to the IDM). By semantic translation we mean a transformation of data from one data model into another one, with preserving the meaning of the data, while observing the constraints of the input data model and enforcing the constraints of the output data model. The IMS also provides a short-term repository for storing the result of the translation, and the schema used in the repository is that of the IDM. Note that translators may create a completely new data set as the result of the translation, or update one that already exists in the repository.

The TA-s are responsible for interfacing the tool with the IMS. Their purpose is to read and write tool data, directly in the form the tool generates and/or expects that data. The adaptors (1) ship the data from the tool to the IMS and (2) receive data from the IMS that they send to the tool. The TA accesses the tool's data in whatever way it is possible and suitable: through a data file, a programmatic interface, or something else. Note that the TA

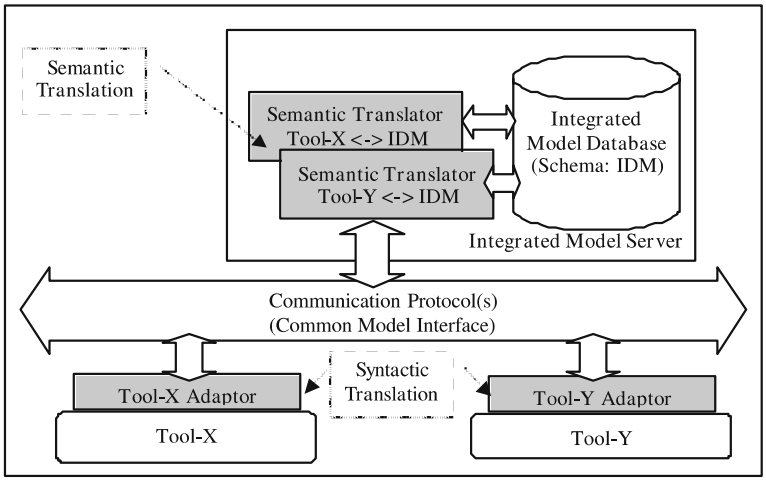

Fig. 3. Tool integration architecture based on an integrated model

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.

The Common Model Interface (CMI) protocol is used to communicate and transfer data between the adaptors and the IMS. This protocol is not dependent on the metamodels of the data: all data shipped in this protocol is in a canonical, "network" form. This is achieved the following way: The protocol includes low-level data structures that are able to express objects (with attributes) and links among objects (including containment). Each object and link is tagged with a unique tag that is derived from the corresponding meta-object of the object or link: a class or association. The abstract syntax component of the metamodel of the DSML of a particular tool consists of classes and associations, and these exist in the form of explicit objects stored present in the IMS, and these objects may provide the unique tags.

The "network" form for shipping tool data can be implemented using different techniques. One straightforward choice is to use XMI [29], but other encoding schemes can be used as well. In a practical implementation we have designed data structures using data types supported by CORBA IDL, and were able to achieve acceptable performance when transferring large models.

When a data set is constructed in a TA, the TA will access the meta-objects in the IMS, retrieves the unique tags for classes and associations, and these tags are added to the (generic) objects and links that are constructed from the tool's data. This process is illustrated on Fig. 4. When a semantic translator receives this data set, it is able to look up the "type" of each object and link in the data based on the tag, and thus determine how the objects should be transformed. A similar process works in reverse when a data set is shipped from the IMS to a TA.

In the general sense, the architecture is used as follows. When a tool wants to make its data available for other tools, its TA is activated. 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 available in 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 decouples syntactical issues from semantic issues such that they can be addressed independently. The binding between the major components is the middleware, implementing a protocol for data interchange.

The definition of the architectural design pattern is as follows.

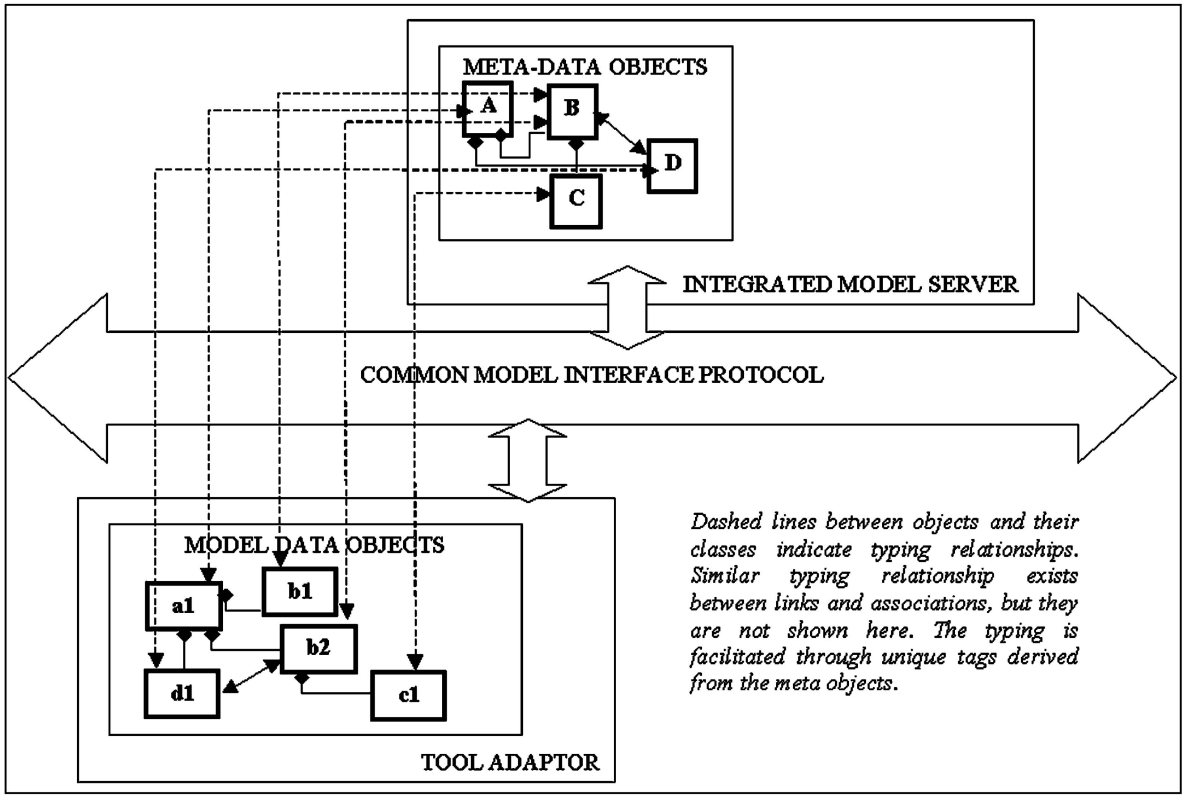

Fig. 4. Typing of model data objects

Name: Tool Integration via Integrated Data Model.

Intent: Provide a generic architectural solution for Open Tool Integration when there is a significant overlap among the data present in the tools to be integrated.

Motivation: Design tools that have overlapping data must be integrated to support a development process. The integration solution must facilitate the interchange of data among tools: data produced in one tool should be usable in another tool. The architecture should not depend on the physical representation of the data used in the tools, and it should support a "post/fetch" style of operation.

Applicability: Use this pattern if: (1) there is significant overlap among data elements in the various tools, (2) well-defined metamodels can be developed for the individual tools, (3) the publish/fetch style of operation fits the needs of the engineering process supported.

Structure: See Fig. 5 and discussion above.

Participants:

Tool Adaptor: Responsible for converting tool data between the physical form and the canonical form used to interchange data with the Integrated Model Server.

Integrated Model Server: Responsible for hosting the Integrated Model Database (for short-term storage of the tool data) and the various semantic translators that map tool-specific data into data in the Integrated Model Server.

Tool: Some engineering tool, whose data must be shared with other design tools.

Collaborations:

- The tool adaptor interacts with the tool by reading and writing the tool's data.
- The tool adaptor and IMS interact by exchanging data sets.
- Within the IMS the semantic translators map between the tool-specific data sets (in canonical form) and the content of the Integrated Model Database.

Consequences: The architectural pattern has the following benefits and liabilities.

- It isolates the syntactical transformation on the data from the semantic transformation on the data. The former is done in the TA, the latter is done in the IMS.
- The core components within the architecture are reusable: they are not dependent on the particular tools used, as the tools are represented through their metamodels. The same reusability applies to the core protocols.
- The IMS operates with a schema that is derived using the procedure described above. For a large number of tools, it could be difficult to derive and maintain this integrated data model.

Implementation: The approach can be implemented using any middleware facility (e.g. CORBA, COM, etc.) that supports remote object invocation and the transfer of complex data structures. For large datasets it may be necessary to design a separate high-performance data management layer, which ships data in canonical form between the components.

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 software suite.

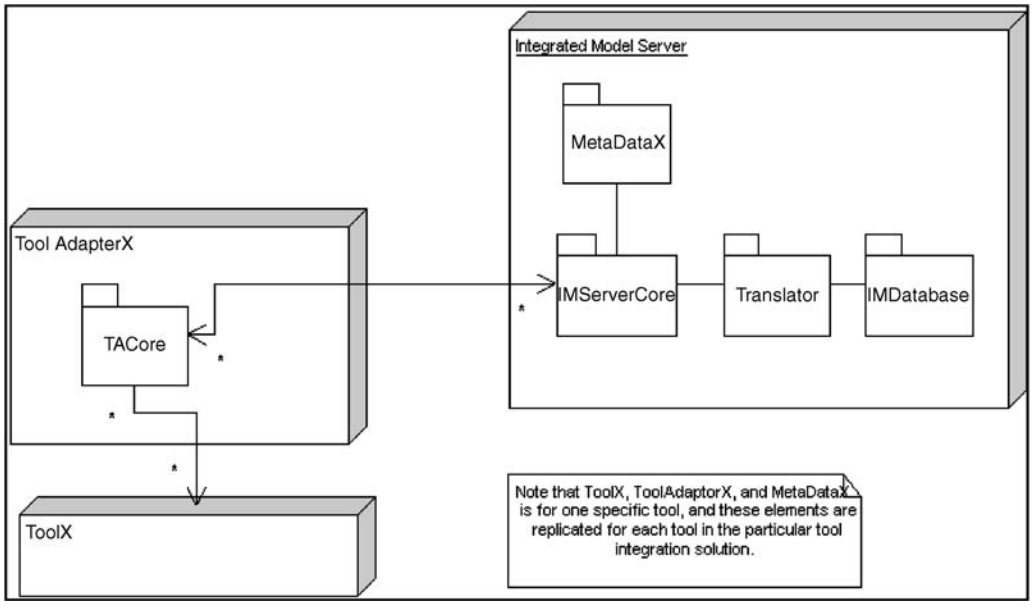

Fig. 5. Design pattern structure for "Integration based on integrated data model"

The engineering process identifies several contributors in the engineering process: (1) the component developer, who builds software components using standard CASE tools, 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. Similarly to 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, but realized a point-to-point integration instead. The resulting, notional architecture is shown on Fig. 6.

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, and the ST-s are not part of a single architectural element anymore. 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.

In this architectural pattern, we have a more sophisticated model of the workflow than in the previous one. There, the workflow is somewhat ad-hoc, defined through the "publish/fetch" activities of the individual tool adaptors. If a TA published a data set that got translated and deposited into the IMS, then any other tool adaptor TA' that had a semantic translator that could translate from the IMS into the DSML of the TA' was able to fetch it. Here, the workflow among the tools is more restricted, explicitly represented in and enforced by the backplane. A workflow is represented as a flow-graph connecting specific tools. An example workflow model is shown in Fig. 7. In the example, ESCM_UDM_UPDATER, ESCM_UDM_PUBLISHER, ESML_RECEIVER_PUBLISHER, CONFIG_RECEIVER_PUBLISHER, AIF_RECEIVER, and AIF_UPDATER are tool adaptor types (interfacing specific tools to the backplane), and all the other intermediate elements (RR2ESML_ESML, ESML2CONFIG, CONFIG2ESML, ESML2AIF, IIF2AIF) are semantic translators. A translator is always placed between two tool adaptors, and its placement indicates that the translator will receive the data published by a "producer" tool adaptor, and then sends the results to a "consumer" tool adaptor. Tool adaptors and translators can have multiple inputs and outputs. In our implementation of the architectural pattern, we have used a DSML for representing workflows, and the backplane component included a workflow engine that was configured through the workflow models.

The architecture operates as follows. The backplane is initialized, and all the metamodels, translators and workflows are instantiated. This step is required, as we assume that the backplane component is generic, and all configuration of it is done at run-time. A manager tool is available to configure the backplane and monitor its operation. Workflows represent which tools are publishers of and subscribers to what type of models, and how these tools are sequenced. Tool adaptors, 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. The TA in this stage performs the syntactic transformation on the data, just like in the previous architecture. The backplane receives the tool's data in the canonical, "network" form, which has each data object tagged (and thus typed) with the corresponding meta-object. Based on the workflow specification, the back-

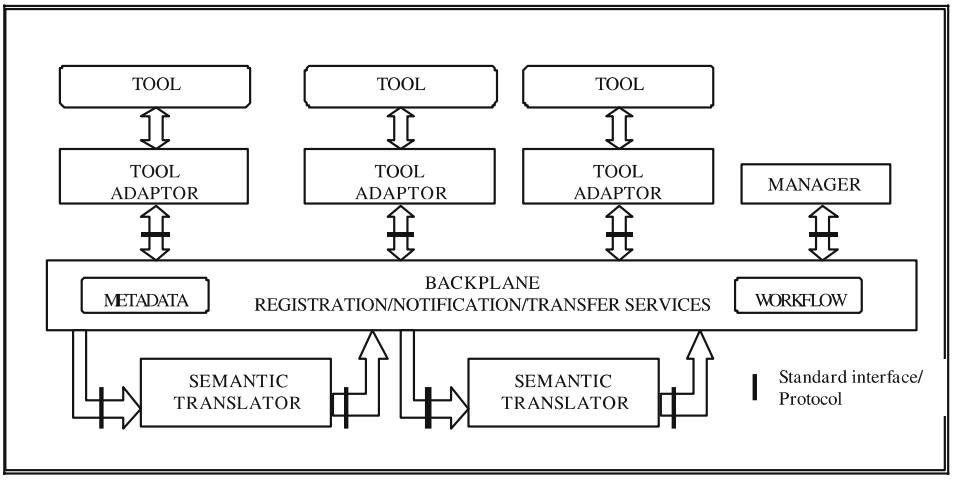

Fig. 6. Tool integration based on process flows

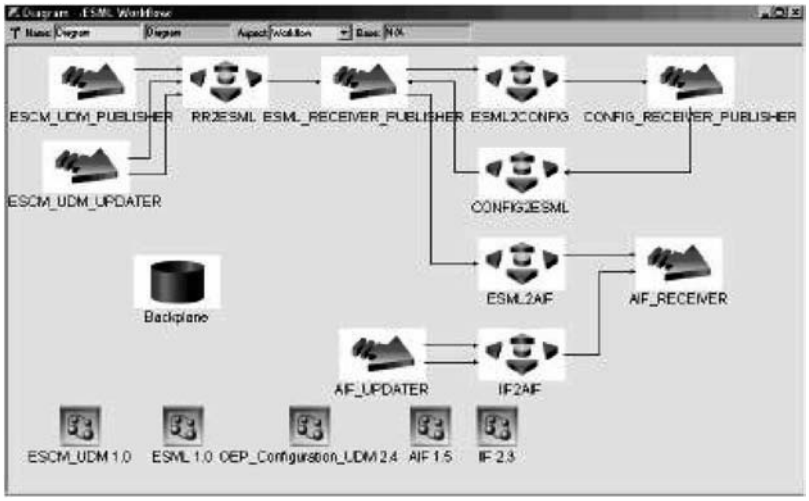

Fig. 7. Example workflow

plane 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) and feeds the data set (still in canonical form) to the translator. The semantic translator performs the translation step, generates a data set in canonical, "network" form, compliant with the metamodel of its output, and sends this data set to the backplane. When this data set arrives at the backplane, the backplane routes the set to the consumer tool, which first gets a notification, and then, if the user chooses, can download the data. The consumer tool adaptor performs the translation from the canonical form into the tool's physical data format (as in the previous case).

The definition of the architectural design pattern is as follows.

Name: Tool Integration via Process Flows (or Workflows).

Intent: Provide a generic architectural solution for Open Tool Integration where there is a clearly defined workflow among tools to be integrated.

Motivation: Design tools that have overlapping data and precisely defined workflows must be integrated to support a development process. The integration solution must facilitate the interchange of data among tools: data produced in one tool should be usable in another tool. The architecture should not depend on the physical representation of the data used in the tools, and it should support a "publish/fetch" style of operation, where the "publish/fetch" happens always between two specific tools.

Applicability: Use this pattern if: (1) there is a well-defined workflow among tools used in the process, (2) well-defined metamodels are available for the individual tools, (3) the workflow style of operation fits the needs of the engineering process supported.

Structure: See Fig. 8 and discussion above.

Participants:

Tool Adaptor: Responsible for converting tool data between the physical form and the canonical form used to interchange data with the Backplane.

Backplane: Responsible for facilitating the workflow among tools and receiving and routing data sets between to/from tool adaptors and semantic translators.

Semantic Translator: Responsible for translating data sets in canonical form. The input data set is compliant with the metamodel of a "publisher" tool, and the output data set is compliant with the metamodel of a "consumer" tool.

Tool: Some design tool, whose data must be shared with other design tools.

Collaborations:

- The tool adaptor interacts with the tool by reading and writing its data.
- The tool adaptor and backplane interact by exchanging data sets.
- The backplane feeds data to and receives data from specific translators. The exact routing depends on the type of the data set and the workflow.
- The semantic translator collaborates with the backplane by receiving the data sets and feeding the results of the translation back to the backplane.

Consequences: The architectural pattern has the following benefits and liabilities.

- It isolates the syntactical transformation on the data from the semantic transformation on the data. The former is done in the TA, the latter is done in the semantic translator(s).
- The core components within the architecture are reusable: they are not dependent on the particular tools used, as the tools are represented through their metamodels. The same reusability applies to the core protocols.

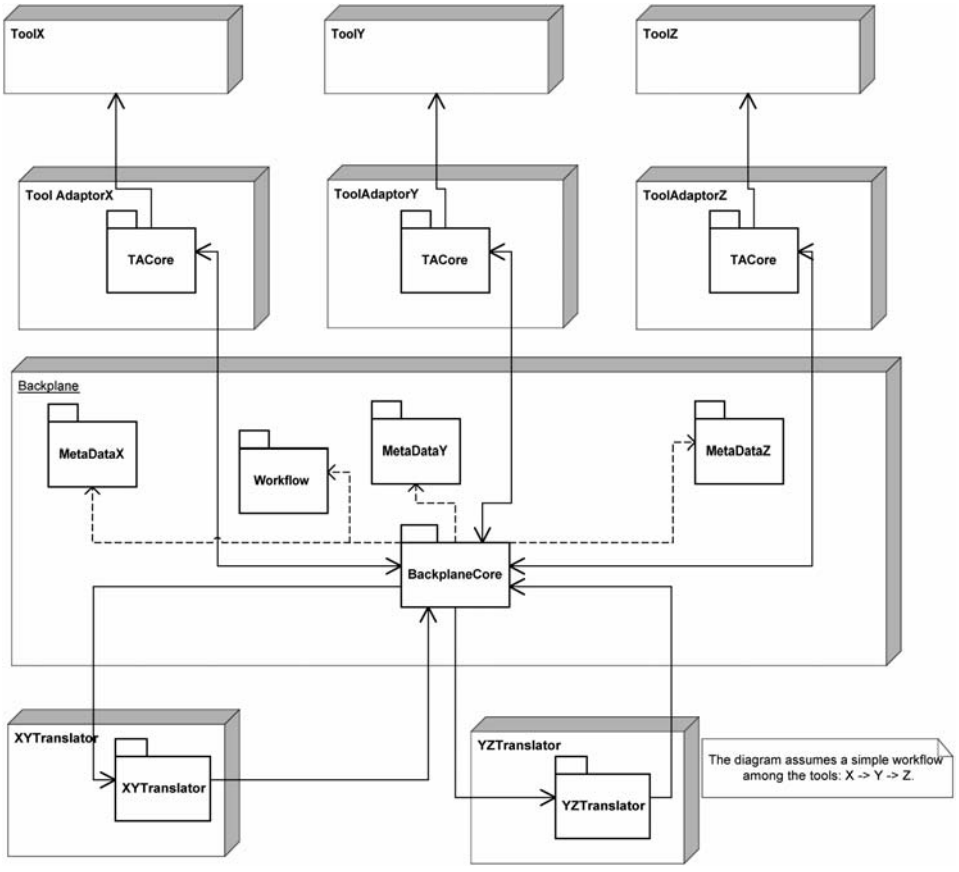

Fig. 8. Design pattern structure for "Integration based on process flows"

- Depending on the structure of a workflow, a large number of translators may be necessary to build all the possible paths among tools.

Implementation: The approach can be implemented using any middleware facility (e.g. CORBA, COM, etc.) that supports remote object invocation and the transfer of complex data structures. For large datasets it may be necessary to design a separate high-performance data management layer, which ships data in canonical form between the components.

Illustrative example for integration based on process flows

Here we describe a tool integration solution: a particular tool chain that we built to support an engineering process to develop avionics software. The process includes various participants:

- The component developer, who builds software components
- The system developer, who configures and integrates components to build full systems
- The system analysis engineer, who analyzes the system configurations for, e.g. schedulability
- The test engineer, who compiles and runs executables and gathers data from the execution.

Note that occasionally a person can do more than one of these tasks.

We have created a tool integration solution, which supports a process that connects these participants into a coherent workflow. The schematic representation of the solution is shown in Fig. 9. Because a central component in the tool chain is a design modeling language called ESML [23], we call it the "ESML Toolchain".

In the tool chain, the component developer is using Rational Rose to model the application components and to generate the API-s of components. The behavioral code for the components is hand-written. Component models are exported from Rational Rose in standardized form, called ESCM (Embedded System Component Model), which can be easily generated from the XMI representation of models. For system level modeling, the system developer is using the ESML language (supported by a visual modeling environment, called ESML/GME). ESML allows defining the components of a system, specifying the potential interactions among components, describing the hardware configuration of the system, and allocating the software components to hardware elements. The analysis engineer is using various tools for schedulability analysis. For being able to interface a number of different analysis tools to ESML, we have defined a common Analysis Interchange Format (AIF), which is supported by many, different analysis tools. The test engineer uses

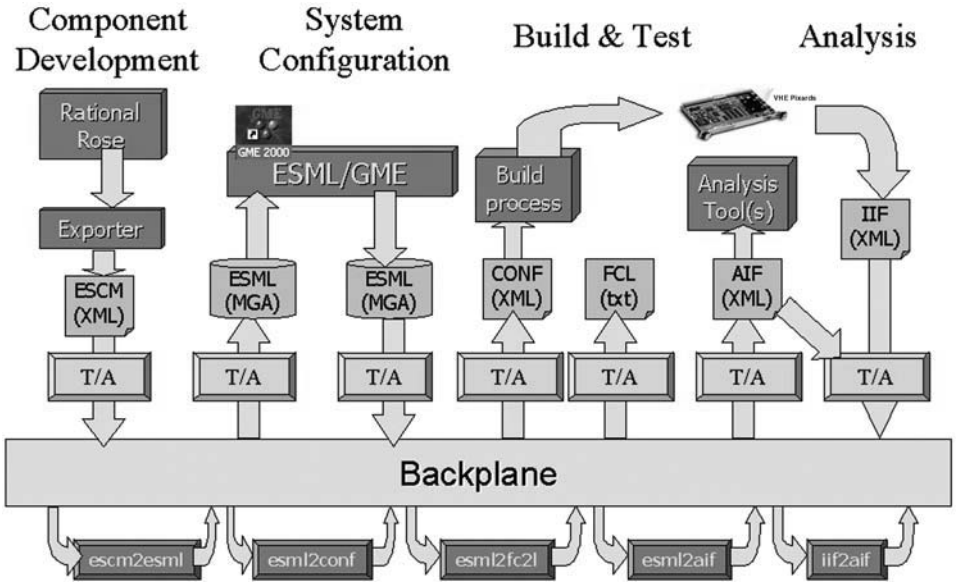

Fig. 9. Tool integration solution for the ESML tools

XML-based configuration files, and generates executables based on the content of those files using some build scripts. The executables are run on the embedded platform, which is equipped with some software instrumentation tools that gather (for instance, timing-related) data from the running system. There exists a version of the run-time platform (which supports the component execution) that allows specification of Quality of Service (QoS) properties, and dynamically adapts the scheduling policies used in order to satisfy the QoS requirements. This variant of the run-time system is configured with a special file, called the FCL file. The performance data gathered during execution is made available using a standard interchange format call Instrumentation Interchange Format (IIF). The data can be used to update the AIF format of the system models, to add information like Worst-Case Execution Time for the components.

The above process is supported by a tool integration solution, which includes a number of tool adaptors and translators. The tool adaptors read or write ESCM, ESML, CONF, AIF, FCL, and IIF data, while the translator translate between the various formats as required (shown below the backplane on the figure). This tool chain has been tried and evaluated by embedded software engineers of a major aerospace manufacturer, and was found extremely useful to address typical problems in development.

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 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: $\Delta y = f(\Delta x, y_{\text{old}})$.
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. Recent proposals for XMI [31] introduce extensions to XMI to handle delta-interchange, which could be used support change propagation. At this time, the creation of the incremental translator seems to be the most problematic: deltas can often be understood only with respect to a previous "state" (i.e. a previous version of a model), thus the translators may need to cache models that they have translated.

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*. By traceability we mean the ability to trace relationships among model elements across multiple tools. Traceability allows, for instance, what elements depend on what other elements, etc. One technique that allows this is based on graph transformations [34]. The model transformations can be expressed in the form of graph transformation rules (which match typed subgraphs on the input and construct typed subgraphs for the output), while the strict type system enforces that only syntactically correct models could be produced. Writing translators using these high-level rules not only enhances productivity but also allows reasoning about the transformations, including reasoning about traceability. As the transformation rules explicitly relate elements of the input to elements of the output, the information needed for tracing is available. We have designed a language: GReAT (for Graph Rewriting And Transformations) and a set of associated tools (visual programming environment, transformation rule interpreter, code generator, debugger) [25] that support building tool integration solutions, through constructing the metamodels of tools and models of the translation between tools.

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. By "significant overlap" we mean that a high percentage of the classes and associations in the IDM have a direct correspondence with classes and associations in the metamodels of the tools. The approach implements a full integration across N tools, using N (bidirectional) translators. The IDM is effectively a common, "universal language" that is used to interchange models. The shortcoming of this tool integration pattern is apparent if one tries to integrate tools with widely different metamodels: if the coupling among the elements of the metamodels of the tools is weak (but nevertheless present) than it becomes difficult to determine the correct mapping. One major problem is that of the "reasonable defaults": if concepts C_A and C_B are present in tools T_A and T_B, respectively, and there is a partially defined mapping between C_A and C_B, then it is difficult to come up with an algorithm that maps the instances of C_A into instances of C_B, as it is not known what default values to choose for properties that $C_B$has but C_A does not. These kinds of translations may require user input to make the target models complete.

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 $\ll$ 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.

Both of the design patterns have been tried out in experimental systems: they formed the underlying architecture of two, independent tool integration frameworks. These frameworks then have been used to build several, specific integration solutions.

In one practical experiment, we have created an integration solution using the IDM-based approach for four tools that were used in building the health management system of an aircraft. The metamodels of the tools were typically simple ($\sim$ 10–20 model elements). The IDM for this particular application was of similar complexity (with $\sim$ 30 model elements). The typical average effort taken by integrating one tool was about 2 engineer-months, evenly divided between developing a tool adaptor and developing a translator. However, by the time integrated the fourth tool we noticed that the time required increased, and significant effort was spent in tracing the mapping relationships from tool metamodel to IDM to tool metamodels. Note that this is an indication that the IDM approach, although conceptually simple, may not be feasible to integrate a large number of design tools. Developers who create tool integration solutions need some sort of tool support to manage the complexity of multiple, overlapping, complementary, or contradictory data models. While in the prototype application there was a common artifact (the aircraft), it was not sufficient in itself to make the problem 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 the appropriate translator(s).

In a separate experiment, the process-based approach fared much better: while adding a new tool took about the same effort as in the IDM-based case, it did not get worse by the increasing number of tools. In this domain the metamodels were more complex ($\sim$40 model elements), accordingly the translator and tool adaptor implementation became more complex. However, we were able to take advantage of the graph transformation technology

described above, which increased productivity. While it is hard to give a single number for the effort required to add a new tool, the $1+1$ engineer/month effort seems like a good average.

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 [10]. 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) modeling 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 a reasonable effort.

There are a number of other, relevant aspects of patterns that could be used in deciding which one to use. Regarding the traceability of model elements across tools, the IDM-based solution fares better as it is easier to recognize the links through the single IDM, than in the case of the process-based solution. Regarding the bidirectionality of transformations, the IDM-based solution requires it more often than the process-based solution, as processes are often of the "feed-forward" type. Incremental transformations are more problematic in the IDM-based solution, as the incremental change may affect multiple tools, than in the process-based solution. Regarding scale-up to a large number of tools, the IDM-based solution is clearly inferior to the process-based solution, as it was observed in the experimental systems.

Related work

The need for integrating (software) design tools has been recognized since the appearance of CAD systems, and many, architecture-based approaches have been developed. Below we summarize and compare a few of them to the approaches described here.

Electronic Design Automation (EDA) [34] is an electronics industry standardization effort that provides a framework for integration of electronic design tools. It uses VHDL as common format for representing tool data. The basic tool integration is achieved by transferring design data between tools using VHDL as the intermediate format and using tool specific translators convert to and from VHDL. Conceptually the approach is similar to the IDM, but with a fixed, common interchange language: VHDL, which limits its applicability to the domain of VLSI design tools.

UniForM Workbench [18]is a universal environment for formal methods, which follows the ECMA Reference Model [13] that outlines the abstract functionality required to support the various dimensions of a tool integration process. It encapsulates existing development tools, and uses Haskell as the integration language (i.e. "glueware") to integrate the tools. It provides a number of common services like Repository Manager (for data integration), Subsystem Interaction Manager (for control integration), and User Interaction Manager (for presentation integration). The interaction model employed is similar to the BackPlane concept of the process flow based integration. However, integration requires programming in a complex, high-level language, and the integration is not standards-based (like CORBA, or MOF).

The work described in [35] introduces an integration framework that supports traceability across software engineering tools, and the "linking" to facilitate this is implemented using CORBA services. In a sense, this is yet another architectural pattern for tool integration. An event-based integration approach is discussed in [18], where the publish-subscribe pattern is used to support low-level, fine-grain, event-driven integration. The technique discussed in [2] is similar to the design patterns presented here ("tool adapters" and "information backbone") however it is more service-oriented and does not rely on metamodels. The platform-based approach described in [26] provides an integration approach based on fine-grain coordination across tools, and uses a tool description language, similar (in spirit, if not in details) to the metamodels described here. The use of explicit metamodels and model transformations to facilitate interchange is discussed in [5], however architectural details for the implementation are not described. Another metamodel-based integration approach is described in [7], where the (data) integration is explicitly modeled. This is an alternative approach to the IDM described above, and, possibly, it can be used to compute the specific IDM. A model integration approach described in [35] solves the run-time integration of (active) models using a message (CORBA)-based framework, and it presents another architectural pattern. The work presented in [18] describes techniques that could be used to generate (at least part of) the code for the translators in the IDM-based pattern. The "homogenizer wrappers" implement precisely the kind of operations the translators must perform in order to (re-)express tool models in the integrated data model.

In contrast, the approaches presented here provide architectural solutions combined with the use of metamodels and semantic translators, in a common, reusable framework. The framework supports large-grain model integration, where the integration relies on the use of metamodels and mappings among them (that are used to implement model transformations). Elements of the patterns can be recognized in other solutions (notably [2, 3, 5, 26]), and, arguably, the pattern can be used to extend those solutions as well. For instance, the second pattern can be used in the implementation of [5], while the metamodel-based techniques can be used to enhance [2].

The application of the patterns described benefits the software engineer who is responsible for building a tool integration solution. The patterns give a generic solution

for a recurring design problem: how to integrate a set of design tools, and some elements (e.g. architectural components and protocols) of the patterns can be made fully reusable across multiple domains. The patterns, with reusable implementation of elements can be packaged as a library that can be used to *instantiate* the patterns for specific cases. We have instantiated the second pattern in three different cases, for three different toolchains (of comparable complexity as described above), and experience showed that new tool chains can be integrated with 1+1 engineer month's effort, on the average.

Summary and future work

We have shown two architectural patterns that can be used to build 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.

Acknowledgements. 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. Altheide F, Dörfel S, Doerr H, Kanzleiter J (2003) An Architecture for a Sustainable Tool Integration Framework. In: ESEC/FSE Workshop on Tool-Integration in System Development, Helsinki, Finland, pp 29–32. Available from: `http://www.es.tu-darmstadt.de/english/events/tis/documentation/Proceedings.pdf`
3. Bergstra J, Klint P (1998) The discrete time ToolBus: A software coordination architecture. Science of Computer Programming 31(2–3):205–229, July
4. Boekhudt C (2003) The Big Bang Theory of IDE-s. ACM Queue 1(7):74–83
5. Braun P (2003) Metamodel-Based Integration of Tools. In: ESEC/FSE Workshop on Tool-Integration in System Development, Helsinki, Finland, pp 45–30. See [2]
6. Broy M, Dederichs F, Dendorfer C, Fuchs M, Gritzner TF, Weber R (1993) The design of distributed systems – an introduction to FOCUS. Technical Report TUM-19202-2, Institut für Informatik, Technische Universität, München, January
7. Burmester S, Giese H, Niere J, Tichy M, Wadsack JP, Wagner R, Wendehals L (2003) Tool Integration at the Meta-Model Level within the Fujaba Tool Suite. In: ESEC/FSE Workshop on Tool-Integration in System Development, Helsinki, Finland, pp 51–56. See [2]
8. Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M (1996) Pattern-oriented Software Architecture: A System of Patterns. John Wiley & Sons
9. Clark T, Evans A, Kent S, Sammut P (2001) The MMF Approach to Engineering Object-Oriented Design Languages. In: Workshop on Language Descriptions, Tools and Applications (LDTA2001), April
10. Czarnecki K, Eisenecker U (2000) Generative Programming – Methods, Tools, and Applications. Addison-Wesley
11. Eclipse Framework (2004) `www.eclipse.org`
12. ECMA TR/55 (1993) Reference Model for Software Engineering Environments. NIST Spec. Pub 500-211
13. ECMA (1994) Portable Common Tool Environment (PCTE) – Abstract Specification. European Computer Manufacturers Association, 3rd edition, Standard ECMA-149
14. EDA (1995) `http://members.tripod.com/~encapsulate/thesis.html`
15. Braun V, Margaria T, Steffen B (2003) The Electronic Tool Integration Platform (ETI) and the Petri Net Technology. Petri Net Technology for Communication-Based Systems 2003:363–382
16. Gabriel RP (1996) Patterns of Software: tales from the software community. Oxford University Press
17. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design Patterns. Addison-Wesley
18. Haase T (2003) Semi-Automatic Wrapper Generation for a-posteriori Integration. ESEC/FSE Workshop on Tool-Integration in System Development, Helsinki, Finland, pp 84–88. See [2]
19. Hansen KM (2003) Activity-Centred Tool Integration. Using Type-Based Publish/Subscribe for Peer-to-Peer Tool Integration. ESEC/FSE Workshop on Tool-Integration in System Development, Helsinki, Finland, pp 11–16. See [2]
20. Karlsen E (1998) The UniForM WorkBench – a higher order tool integration framework. In: International Workshop on Current Trends in Applied Formal Methods, October
21. Karsai G (1999) Structured Specification of Model Interpreters. In: Proc. of International Conference on Engineering of Computer-Based Systems, Nashville, TN
22. Karsai G, Gray J (2000) Design Tool Integration: An Exercise in Semantic Interoperability. In: Proceedings of the IEEE Engineering of Computer Based Systems, Edinburgh, UK, March
23. Karsai G, Neema S, Abbott B, Sharp D (2002) A Modeling Language and its Supporting Tools for Avionics Systems. 21st Digital Avionics Systems Conference, August
24. Karsai G, Sztipanovits J, Ledeczi A, Bapty T (2003) Model-Integrated Development of Embedded Software. In: Proceedings of the IEEE, vol 91, no 1, pp 145–164, January
25. Karsai G, Agrawal A (2004) Graph Transformations in OMG's Model-Driven. In: Applications of Graph Transformations with Industrial Relevance, Charlottesville, Virginia, September. Lecture Notes of Computer Science, vol 3062. Springer, pp 243–259
26. Karsai G, Agarwal A, Shi F, Sprinkle J (2003) On the Use of Graph Transformation in the Formal Specification of Model Interpreters. Journal of Universal Computer Science 9(11):1296–1321

27. Margaria T, Wübben M (2003) Tool Integration in the ETI Platform – Review and Perspectives. In: ESEC/FSE Workshop on Tool-Integration in System Development, Helsinki, Finland, pp 39–44. See [2]
28. MOBIES Project (2004) http://www.isis.vanderbilt.edu/Projects/mobies/default.html
29. OMG MDA (2004) http://www.omg.org/mda
30. OMG MOF (2004) http://www.omg.org/mof
31. OMG XMI FTF (2004) http://www.omg.org/techprocess/meetings/schedule/MOF_2.0_XMI_FTF.html
32. PCTE Standard (1998) ISO/IEC 13719
33. Rational Corporation (2004) http://www.rational.com
34. Schettler O (1995) Encapsulating design tools in the EDA. http://members.tripod.com/~encapsulate/thesis.html
35. Schopfer G, Yang A, Marquardt W (2003) Tool-Integration in Chemical Process Modeling. In: ESEC/FSE Workshop on Tool-Integration in System Development, Helsinki, Finland, pp 79–83. See [2]
36. Wilcox P, Weiss D, Russell C, Smith MJ, Smith AD, Pooley RJ, MacKinnon LM, Dewar RG (2003) A CORBA-Oriented Approach To Heterogeneous Tool Integration; OPHELIA. In: ESEC/FSE Workshop on Tool-Integration in System Development, Helsinki, Finland, pp 1–5. See [2]

Gabor Karsai is Associate Professor of Electrical and Computer Engineering at Vanderbilt University and Senior Research Scientist at the Institute for Software-Integrated Systems at Vanderbilt. He got his BSc, MSc and Technical Doctorate degrees from the Technical University of Budapest, Hungary, in 1982, 1984, and 1988, and the PhD degree from Vanderbilt University, in 1988. He conducts research in model-integrated computing (MIC), in open tool integration frameworks, automatic program synthesis and the application of MIC in various government and industrial projects. He is a senior member of the IEEE Computer Society and the TC on Computer-Based Systems.

Andras Lang is currently a project leader with a small software company in Hungary. He was a Staff Engineer at the Institute for Software-Integrated Systems at Vanderbilt University for the last two and a half years. His research interests includes: design tool-integration frameworks, infrastructures for implementing domain-specific modeling languages and model-based design of embedded systems. He received his master's degree from Budapest University of Technology and Economics in 2001.

Sandeep Neema is a Research Assistant Professor of Electrical Engineering and Computer Science at Vanderbilt University. His research interests include: design tool-integration frameworks, dynamic adaptation for QoS assurance in distributed real-time embedded systems, model-based design of embedded systems, aspect-oriented program composition techniques, design space exploration and constraint based synthesis of embedded systems, and fault-tolerance in large-scale computing clusters. He received his PhD from Vanderbilt University in 2001.

Tool Integration Aspects in the Model-Driven Architecture

Gabor Karsai

gabor.karsai@vanderbilt.edu
Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37235, USA

Abstract

Proponents of the MDA vision seem to agree that it will become reality only if we have the proper tools to practice it. Using models in software development poses interesting challenges for the tool developers: tools are needed (1) for modeling on varying levels of abstraction, (2) for transforming models between modeling paradigms (and code), and (3) for analyzing and verifying properties of models (to ensure we build the right system correctly). In addition to the need for usable tools (to avoid becoming "shelfware"), tools must talk to each other and work together as a seamlessly integrated ensemble. This paper outlines the various aspects of the model-driven development process, the specific tool categories needed, and highlights the integration problems arising in tool suites.

Introduction

The Model-Driven Architecture (MDA) [1] is a recent conceptual framework for software engineering and development practices, mostly promoted by the Object Management Group (OMG). The key aspect of MDA is the overwhelming use of models in the software development process: models for capturing requirements, models for describing the design, models for analyzing the system before it is built, and models for generating (at least parts of) the final product. Similar approaches, like Software Factories from Microsoft [2], IBM's Rational Rose tools [3], and Borland's Enterprise Studio [4] promote familiar concepts and techniques.

Tool-supported software development is not new and has been tried before under the name Computer-Aided Software Engineering (CASE) [5], with varying success. Arguably, the CASE movement made two lasting impacts: (1) the introduction of the automatic code ("application") generators for specific domains (like controller design, see, e.g. Matrix-X [6] and Matlab [7]), and (2) the introduction of Interactive Development Environments (IDE-s) (that integrate the most commonly used software development tools, like editors, compilers, debuggers, etc. in an interactive framework).

In this paper we analyze how and why MDA is different, what kind of tool integration approaches are available, and how an MDA process can be equipped with specific software tools. As illustration, we discuss our experiences with two tool integration projects.

MDA and Tool Integration

MDA proscribes a development process that relies on models of the software, the system and its environment to build the software product, and thus it is necessary to have tools that create, manipulate, and transform these models. It is hard to envision a single, all-encompassing tool that can do everything. Rather, separate tools that support specific activities in the process are envisioned, which form tool chains. One can recognize an orderly progression from the simple code generators towards domain-specific tools, as outlined below. Below, we look at three approaches and highlight the fundamental research questions related to tool integration in each.

Single-stage generation

As mentioned above, one very useful outcome of the CASE activities was the introduction of Application Generators [8] for restricted domains. The two main examples are Mathworks' Matlab/Simulink/Stateflow tool and National Instruments' Matrix-X tool, both of which support software development for embedded controllers. Early visual modeling tools (e.g. Software-Through-Pictures [9]) and first-generation UML modeling tools (e.g. early versions of Together by Borland [10]) have also had similar capabilities. These tools followed a single pattern, as shown on Figure 1 that emphasized one stage: generating code from models. We call this the "single-stage generation" approach to distinguish it from the more sophisticated techniques discussed later.

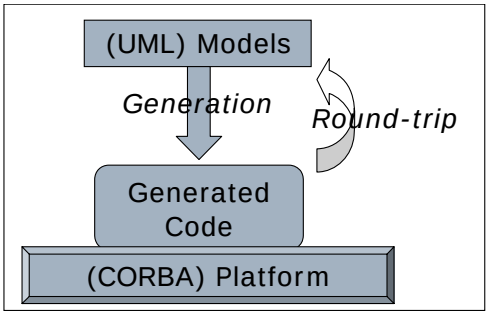

Figure 1: Single-stage generation

Here, the existence of a run-time platform is assumed, such that (1) the generated code is executed on that platform, and (2) the platform provides some OS-like services. For example, for Matrix-X the platform was an Ada run-time system (possibly running on top of a real-time operating systems, like VxWorks [11]). The generator was to map the modeling concepts (e.g. dataflow blocks, finite-state machine diagrams) into platform-specific concepts (e.g. tasks or code fragments). In a sense, the main conceptual problem here was the integration of the model-level concepts with the platform-level details. In other words, the question was how the model semantics was implemented using the capabilities of the platform. Arguably, the problem was solved (as numerous auto-coders have been implemented), however the exact mapping was not or (often) poorly documented.

Yet another, typical service of these single-stage generator tools was a *round-tripping* service. Generated code is considered just another artifact in a development process, but it could also be hand-modified by the developer. This means that changes introduced on the code had to be reflected back to changes on the model and vice versa. This requirement necessitated the development of sophisticated algorithms that detected changes in the code, and reflected those changes on the models such the models and the code were always kept synchronized. Naturally, the mapping between code and models is not bijective, making this step difficult.

Arguably, the need for synchronization is a perceived requirement that is derived from the underlying code-based development process: the ultimate product is the source code that can be modified by the programmer. However, the usefulness of round-tripping is questionable. If one accepts that some parts of the system are implemented in (programming language) code, but other parts are implemented in models, then there is not much need for round-trips. However, another problem arises: namely, how to interface (high-level) models with (hand-written) code? We strongly believe that this model/code interface problem is crucial for MDA and tools are needed that assist developers.

To summarize, some of the research problems related to tool integration that arise in single-stage generation approach are as follows:

- How to describe and implement the code generation process?
- How to ensure and/or verify the correctness the generation process?
- How to maintain synchronization between code and models?
- What is the "conceptual interface" between models and code?

Two-stage generation

In the MDA conceptual framework, as outlined by OMG [1], the single-stage generation is replaced by a two-stage generation process, as shown on Figure 2.

Here, first platform-independent models (PIMs) are created, which are then transformed into platform-specific models (PSMs) that are then used in generating (platform-specific) code. Arguably, the approach was designed to address the need for multi-platform applications that are required to run on different platforms (e.g. CORBA[12], or J2EE[13], or .NET[14]).

Note that two, rather different stages exist here: one for the model transformation (PIM to PSM), and another one for code generation. The developer is supposed to work (primarily) with PIMs, but manual modification of PSMs (and possible the generated code) is still possible. The first stage transformation maps the higher-level, more abstract models into platform-specific models, from which it is easier to generate code. In this model transformation step the (generic) transformation tool may rely on explicit platform models that capture platform specific details in a form understood by the transformer.

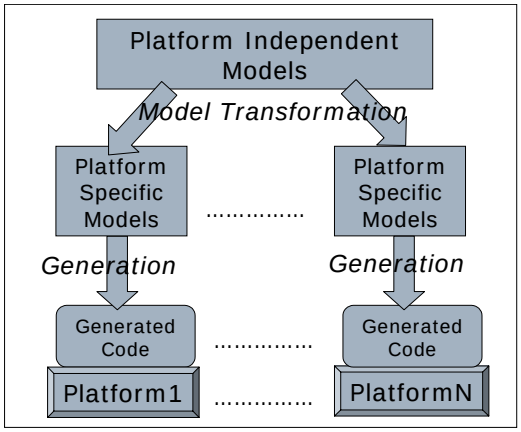

Figure 2: Two-stage generation in MDA

Some research problems specific to the two-stage approach are as follows:

- How to describe and implement the model-to-model transformation process?
- How to ensure/verify the correctness of the transformation?
- How to capture platform models and how these models are to be used in the transformation?
- How do we maintain consistency across PIMs and PSMs?
- How do typical development activities map into this framework? When do we transform models in development? How do we use the transformed models?

Domain-Specific MDA: Model-Integrated Computing

One often unstated assumption in MDA is that UML is *the* modeling language and all models are expressed as UML models. Any extensibility (or domain-specificity) is to be handled through the use of the extension features in UML, namely stereotypes and profiles.

Model-Integrated Computing (MIC) [15] goes one step beyond MDA through relaxing this assumption and advocates the use of domain-specific modeling languages and tools in the development process. MIC is similar to MDA in its advocacy for the ubiquitous use of models, but refines that through allowing and emphasizing the domain-specificity of models. It extends the concepts of domain-specific languages - which, arguably, enhance the programmer's productivity into the model-driven development process. One notional view of MIC is shown on Figure 3.

In MIC, developers use domain-specific model languages for creating models of the application. Often, multiple, yet related modeling languages are used. These domain specific models are then transformed into other, intermediate models or directly used in generation. The intermediate models are used for generation (and thus they subsume the role of PSM-s), and for analysis.

Mathworks' Matlab/Simulink/Stateflow provides an example for an MIC development process. Engineers with expertise in signal processing and controls develop complex applications using the Simulink and Stateflow visual and the Matlab textual languages — both of which are domain-specific. Mathworks' code generator tool creates executable code from the models that can run a platform (e.g. an embedded controller). Third party analysis tools (e.g. SAL [16]) could be used to analyze, for instance, safety properties of the controllers.

The MIC development process can be generalized to arbitrary domains by introducing a higher-level layer for metamodeling [17]. Metamodels provide the formal and computer-readable definition of modeling languages, which are then used in configuring generic tools to support the development process.

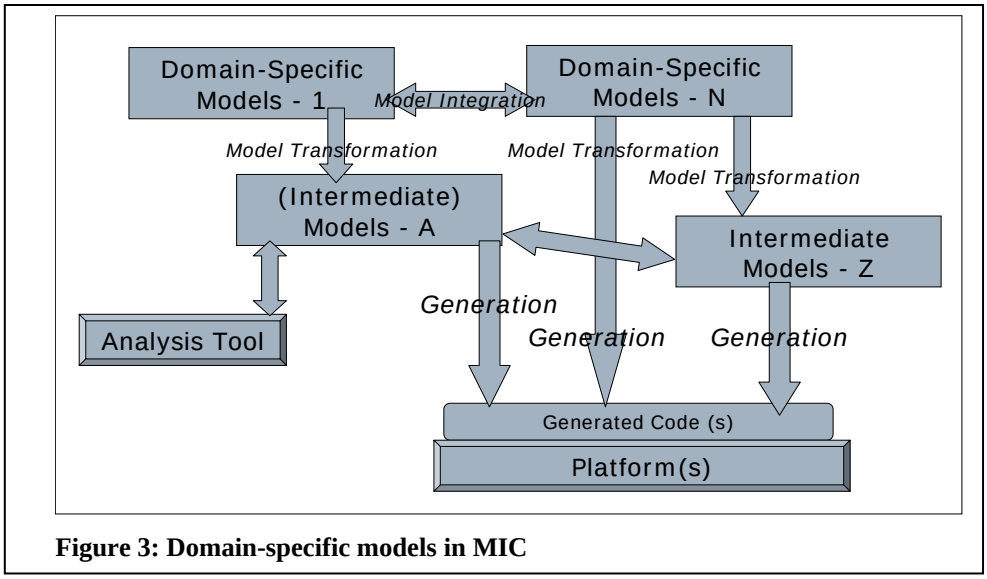

Figure 3: Domain-specific models in MIC

Metamodeling and the use of metamodels in defining MIC environments have been discussed earlier [15].

MIC highlights two main categories of research problems for tool integration. On one hand, the multitude of the domain-specific modeling languages to be used in developing complex applications necessitates "model integration": the integration across the modeling languages. On the "meta" (language definition) level, modeling (sub-) languages for specific domains should have clearly defined interfaces that one can use to compose them. On the operational (implementation) level one has to solve the "single data entry" problem: i.e. information should be entered only once, and data shared across the different modeling paradigms must be kept synchronized.

On the other hand, the domain-specific MIC development environments often require tool integration across the different functional tools: model building tools, simulators, model analysis tools, generator tools, etc. To instantiate a generic MIC process for a specific domain one needs model transformation technology that helps building —affordably— sophisticated tool chains. One needs translators to connect the elements of these toolchains (where the elements are often specialized, custom tools with their own language).

In summary, MIC promotes the use of domain-specific approaches (tools, techniques, languages) in the development process. This is made feasible by the use of metamodels and highly configurable, meta-programmable tools. When tools are also domain-specific, one needs to integrate them into the development process, such that the developers do not have to deal with tool-specific details (e.g. the details of the input language of an analysis tool).

Integration Patterns

As pointed out above, tool integration is essential to a model-driven development process, especially if domain-specific tools are used. Methods and architectural approaches for tool integration have been developed in the past, and in this section we briefly review a few of the major techniques. For a more detailed analysis, see the paper [18].

"Star"

In this approach tools effectively share models with each other through a common database, as shown on Figure 4. The database has a schema that is capable of representing all the data that need to be shared

across the tools. Producer tools publish their data and consumer tools fetch that data via the common database. Tools interact with the database using adaptors and translators that address the syntactic and semantic details of integration, respectively.

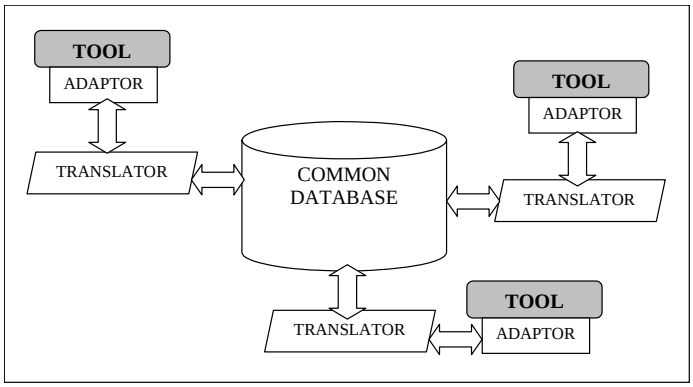

Figure 4: "Star" tool integration pattern example

The "star" approach works well for a small number of tools with significant overlap across the data models of the individual tools [19]. In these situations, the shared (common) data model is easy to design. However, the approach does not scale well for a larger number of tools. Keeping track of related schema elements across more than five tools have been found very difficult in practice. Additionally, the approach does not have a well-defined workflow, as the publishing and fetching of data is completely opportunistic.

Flows

The limitations of the "star" integration pattern led to the development of a different pattern that follows the logical workflow in a toolchain. Instead of a single, centralized database that all tools use to share data, integration happens here in a pairwise manner: the tools interact with each other only as the workflow dictates. For an example see Figure 5.

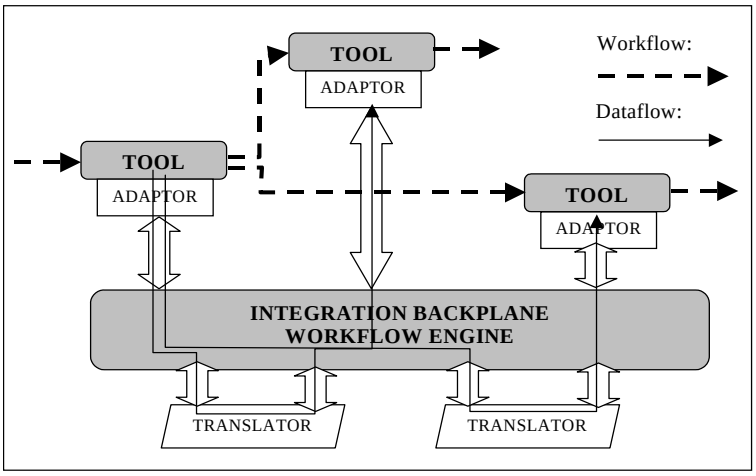

Figure 5: "Flows" tool integration example

The individual tools are interacting with a messaging framework, hosted in the "Integration Backplane" that also includes a workflow engine. This component ships data published by tools to the appropriate translators and then to subscriber tools as the workflow proscribes it. Note that the logical workflow (dashed lines on the figure) is different from the physical dataflow (thin lines), as the data to be interchanged still travels via a central entity: the backplane.

The "flows" approach works well for larger number of tools and it imposes regularity on the operation of tools. However, in itself it is not suitable for keeping a history of operations, and external, repository-like tools are needed for that.

Links

The previous two integration patterns are based on the requirement that data ("models" in MDA) needs to be shared across multiple tools in an engineering process. Often there is a different requirement for tool

integration: namely data sets that are dependent each other must be kept synchronized. This problem can be solved using a third pattern, shown on Figure 6.

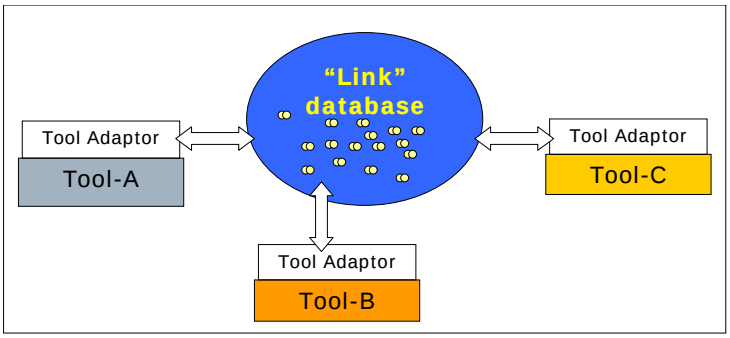

Figure 6: "Links" tool integration example

In this approach the tool integration does not share data across tools, rather, it shares changes to the data [20]. There is a centralized database here as well, but it merely maintains the links that link existing data elements existing in the tools. When changes to data elements are made, the tool adaptor of the source uses the database to determine the dependent and effected data elements, and notifies their tool adaptors, which, in turn, can make the necessary changes in the dependent tools.

All of the tool integration patterns discussed above have been tried and used in a number of toolchains, many of which were not related to only software development [19]. These three patterns provide a conceptual framework for building tool integration solutions that are necessitated by MDA and its MIC variant. They are all patterns, in the sense that they have to instantiated for specific tools and specific problems. This instantiation process can be supported by tools (actually, meta-tools) as discussed below.

GReAT: A tool for model transformations

The "Star" and "Flows" integration patterns necessitate a component called the translator, which translates data from tool to tool. Naturally, the cost of integrating tools depends on the cost of creating such translators, thus a technology is needed for the efficient construction of these translators. Note we mean "programmer's efficiency" here, and not necessarily the efficiency of the translator itself. Note also that translators for tool integration will also implement the model transformation functions needed in MDA and MIC.

In our view, the model transformation problem is best approached by providing a technology that developers can use for the rapid development of model transformation tools. In the XML world, documents are often translated using scripts written in XSLT: the transformation language for XML documents. However this transformation technology does not scale up to large and semantically complex documents or models that are expected in MDA. Motivated by this requirement, during the past few years our research group has built a technology and toolsuite for building model transformation tools, which we summarize here. The technology is called "Graph Rewriting And Transformations" (GReAT), as it uses graph transformation techniques [21]. For more details please see the paper [22].

GReAT is based on a language that supports the high-level specification of model transformation programs. The language is graphical (though it allows textual specifications in selected places), and its programs describe model transformations in terms of sequenced graph rewriting rules. The features of the language are summarized on Figure 7.

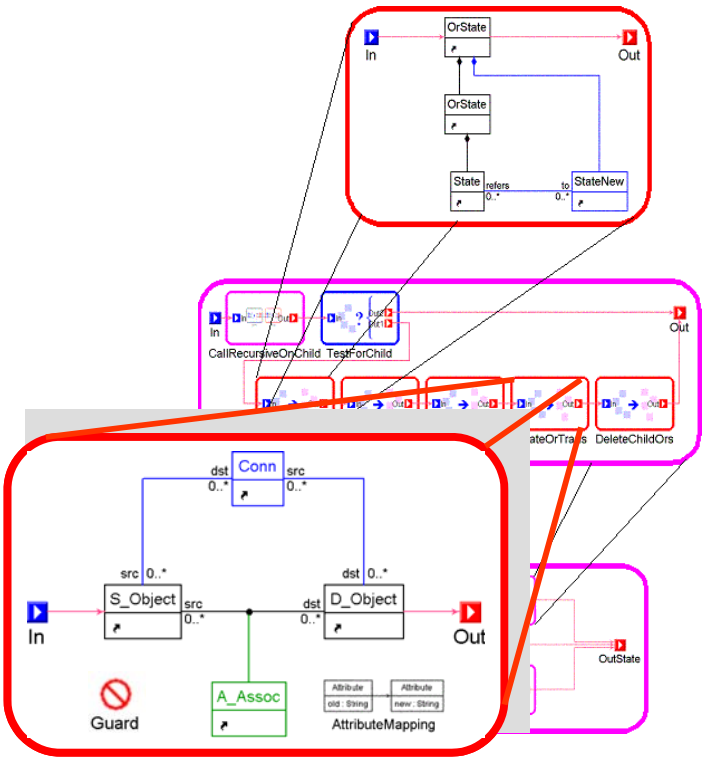

Figure 7: Summary of the GReAT language

A model transformation in GReAT is broken down into elementary graph rewriting steps that (1) recognize a subgraph in the input graph, and (2) create a portion of the output graph. As it is expected, both the inputs and the outputs of the transformation are considered graphs; more precisely, typed and attributed hypergraphs, where the node and edge types correspond to classes and associations of a UML class diagram capturing the metamodel of the input (or output). An individual rewriting rule is shown in the box at the bottom of Figure 7 that includes a graph pattern to be recognized, and the action to be taken. Another rule can be seen in the top box of the figure. On the left and the right side of the rule one can recognize small icons labeled "In" and "Out": these denote ports that are bound to specific nodes in the input (or output) graph before and after the rewriting rule is executed, respectively. These local binding allows very efficient searches for subgraphs, as the rule execution engine will look for a match only in a limited context. The rewriting rules are sequenced, as shown in the middle of Figure 7, and this sequencing supports a small but powerful set of control structures. The sequencing happens by connecting the input and output ports of the rules. Sequences can be encapsulated into higher-level rules, and features like recursion, branching, and non-deterministic choice are supported.

GReAT is equipped with a full suite of tools that support modeling, execution, compilation, and debugging. The graphical model construction is supported through a visual modeling environment based on the Generic Modeling Environment (GME): a metaprogrammable modeling tool [33]. Execution is supported through a Graph Rewriting Engine (GRE) that interprets GReAT programs directly. Compilation is supported by a code generator that performs a partial evaluation on the transformation programs and generates executable C++ code from them. Debugging is supported by an add-on component of GRE that allows interactive control of execution of GReAT programs.

GReAT and its tools have been used to develop a number of model transformation tools model-based development toolchains that were reported elsewhere [32].

Two examples

The real results of tool integration in model-driven software development can be assessed only through specific examples. In this section we review two specific toolchains that we have worked on and report on our experiences.

An MIC toolchain and its impact

The development of mission computing applications for high-performance aircraft is a central topic for distributed, real-time, embedded (DRE) systems [30]. Mission computing systems perform navigation functions, manage other flight systems (e.g. fuel system), run the pilot's interface, and, in general, belong to the classes of soft real-time systems. A typical mission computing application runs on 1-4 processors, and consists of a few hundred to a few thousand software components, each component having about 1,000 lines of source code in a high-level language (C++). The most difficult activity in development is the configuration and integration of the final application.

We have developed a toolchain and a few domain-specific languages for supporting the model-based configuration and integration of mission computing applications. The toolchain is illustrated on Figure 8. The main, system-level domain-specific modeling language is called Embedded System Modeling Language (ESML), and it has been reported elsewhere [23]. The toolchain is named after the modeling language.

The ESML toolchain focuses on the model-based integration of large-scale DRE systems for mission computing applications. Component design and development is done using conventional tools (IBM Rational Rose for design an modeling, and C++ IDE for coding). The subsequent steps in the process rely

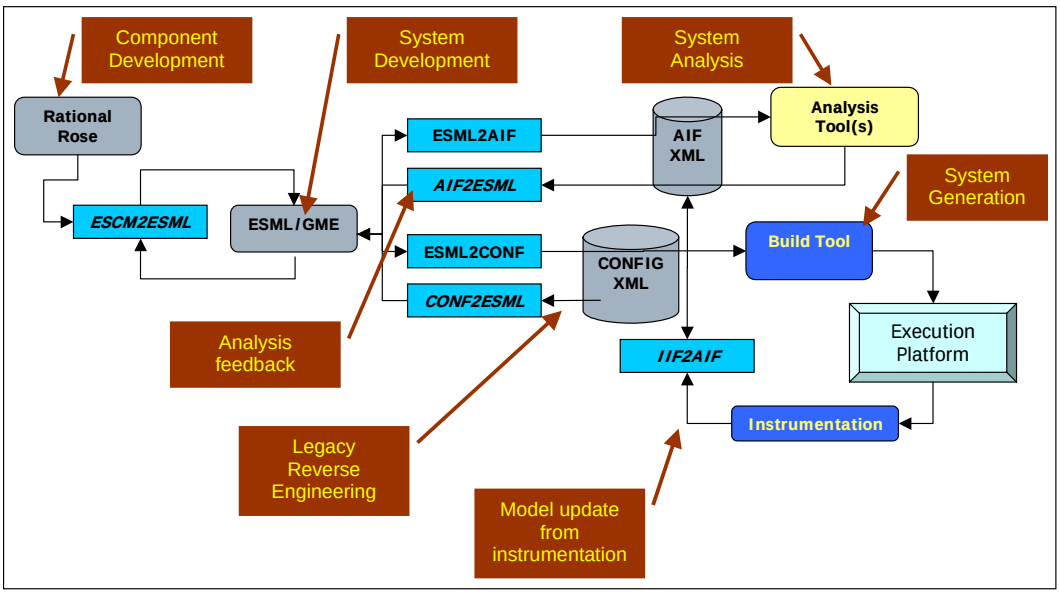

Figure 8: ESML toolchain for developing mission computing applications

on the component models.

Component models are imported into the system-level modeling tool that supports the construction of ESML models. The import service is facilitated by a translator (ESCM2ESML). The translator can also be operated in "update" mode, when ESML models are to be refreshed (but not replaced) with new models from the component modeling tool. The ESML modeling tool supports the visual construction of system configurations from components. The tool also has built-in "design checkers" that warn the developers about semantically incorrect constructs. The constructed system models can be compiled into analysis models (compliant with the Analysis Interchange Format (AIF) XML schema), and then handed over to analysis tools. A typical analysis session of the models includes event dependency analysis, component allocation analysis, and schedulability analysis. For the allocation analysis, the tool can generate new, recommended component allocation models that, in turn, can be imported back into the ESML modeling tool. From the system level model one can generate a Configuration XML file that is used to generate all the initialization code and auxiliary information (e.g. makefiles) for a build tool that compiles and links the final application. The application is executed on a computational platform (typically, an RTOS with real-time CORBA), which could be instrumented to gather run-time data. The data gathered can be incorporated into the AIF files (with the help of a translator: IIF2AIF), such that the analysis tool can take advantage of

actual, measured running times for components. One can also reverse engineer ESML models from existing Configuration XML files that are available for legacy systems.

The above tool chain has been implemented by a number of researchers participating in the project, and has been evaluated by a major aircraft manufacturer and system integrator [24]. Historical data (from two past projects) from the integrator indicated that major projects typically spend over half (51% and 59%) of their efforts in system integration, hence the motivation for building the toolchain for that purpose. The data also indicated that about 25% of defects were related to component interfaces and system configuration. On a medium size example (about 800 components), the model-based approach to integration took about 18 person/hours (as opposed to 62 hours without the tools). For finding integration errors, the model-based approach worked also better: it produced a 12-fold time saving in locating and fixing errors.

The HSIF Experience

The development of embedded computing systems often necessitates a thorough analysis of the software system in the context of its environment. In an embedded application (like an automatic flight control system) physics and computation are interlinked, and the overall dynamic system behavior is determined by both the programmed behaviors and the responses of the environment to the computer's actions. Recently, a new class of analysis techniques has been developed that integrate the continuous-time dynamics of physical systems with the more discrete behavior of computational systems. These techniques and tools are commonly called as "hybrid systems" or "hybrid automata" [25].

In spite of the underlying conceptual similarity, there are a number of hybrid system modeling approaches and corresponding analysis algorithms and tools. For the developer it is very hard to compare the various approaches and tools, especially because various tools excel at verifying various properties. For instance some tools are good at simulating a hybrid system, other tools are good at verifying reachability properties, and yet another tool is good at verifying stability properties. Clearly, there is a need for integrating the various tools and approaches.

In the context of a DARPA-sponsored research project, we have participated in a tool integration effort that aimed at producing a common interchange format for representing hybrid system models. The result of this project was a Hybrid System Interchange Format (HSIF) [26]. Note that this effort was different from typical software design tool integration projects: here, a single interchange format was to be developed and all tools were expected to communicate via this format.

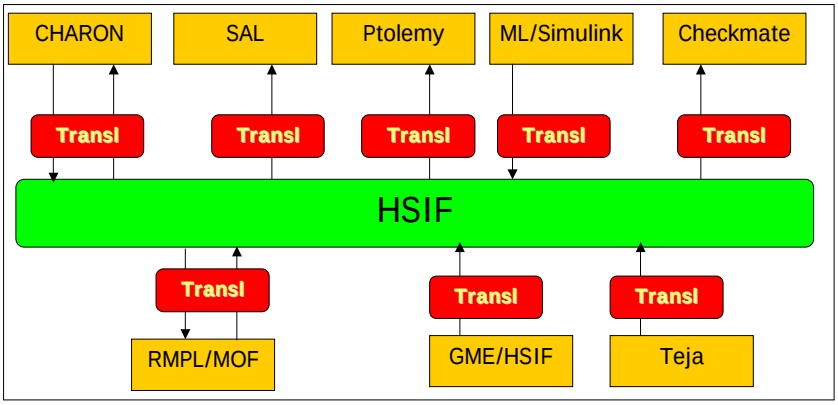

Figure 9: HSIF tool integration example

Figure 9 shows the envisioned interoperability between various hybrid system analysis tools via HSIF. Translators to and from HSIF have been built as indicated on the figure. The tools were: Charon (a hybrid system simulation and analysis toolkit from University of Pennsylvania [27]), SAL (a hybrid system analysis tool from SRI [16]), Ptolemy (a hybrid system simulation tool from UC Berkeley[28]), Matlab/Simulink (a product of Mathworks [7]), Checkmate (a hybrid system simulation and analysis tool from CMU[29]), RMPL/MOF (a hybrid system modeling language from MIT [31]), GME/HSIF (a hybrid system visual modeling environment from Vanderbilt [33], and Teja (a hybrid system modeling and simulation tool from Teja Technologies [34]).

The main benefit of a tool integration solution via HSIF is the ability of verifying controller designs. Industrial participants in the project have pointed out that current model-based tools often lack the formal verification capabilities, and the only applicable approach is testing - often not feasible because the size of the state-space of the systems.

The participants of this project have learned a number of lessons, some of which are listed below:

1. It was surprisingly difficult to arrive at a common and accepted semantics for the interchange format. While the concepts of hybrid automata have been defined many times in the past, the extended version that forms networks of these had no formal definition at the beginning of the project. As the precise semantics is essential in any kind of interchange language, significant effort was spent on defining this shared semantics.

2. The final semantics was more denotational than operational, and thus multiple operational interpretations and implementations were possible. The elegance and precision of denotational semantics is not sufficient for an operational definition. Many tools have been simulators, and when they were implemented they produced slightly different behaviors —even when the implementors worked from the same denotational semantics specification. Implementation details (like floating-point accuracy, treatment of zero-time transitions, etc.) had a significant impact and forced the researchers to re-think the precise semantics, this time from a more operational point of view.

3. In some cases the complexity of the translation was a serious issue. Some of the tools, notably Matlab/Simulink/Stateflow, are widely used in the industry, but do not have direct support for hybrid automata. The translation of models from these tools into HSIF was especially difficult as certain model elements of HSIF could not be determined automatically from the models (e.g. reset functions).

4. For realistic examples, translators often produced un-analyzable models. Straightforward translation of some constructs (e.g. arbitrary C++-style assignments on transitions) into HSIF resulted in models of enormous size that analysis tools were incapable of handling. We learned that translators may need to perform on-the-fly abstraction to simplify models such that the analysis tools could cope with the results.

In summary, the HSIF tool integration project was an extremely valuable exercise in building support for model-based development. It had partial success, but, more importantly, it allowed us to learn about the difficulties of semantics and translation.

Conclusions

Experience shows that model-driven development necessitates tool support, and the tools must work together in a seamlessly integrated manner. In this paper we reviewed the thinking behind the conceptual framework called Model-Driven Architecture (MDA) and a practical extension of it: Model-Integrated Computing. The three, large-scale architectural tool integration patterns provide a starting point from which actual tool integration solutions can be built. All three patterns have been implemented, in many, different systems, and their properties are well established. Experience with tool integration projects for model-driven development indicates the benefits but also highlights the potential problems.

In summary, tool integration is perhaps the most challenging problem facing software engineering, tool developers, and tool users today. However, the model-driven approaches (including the model-transformation technology) provide a first step towards building up a toolbox of solution patterns — with the caveat that much more research is ahead of us.

Acknowledgements

The DARPA/IXO MOBIES program and USAF/AFRL under contract F30602-00-1-0580, and the NSF ITR on "Foundations of Hybrid and Embedded Software Systems" have supported in part, the activities described in this paper. The author also would like to thank Ben Abbott, Aditya Agarwal, John Bay, Alex Egyed, Anouck Girard, Zsolt Kalmar, Bruce Krogh, Andras Lang, Eward Lee, Insup Lee, Sandeeo Neema, Wendy Roll, Mark Schulte, Dave Sharp, Feng Shi, Kang Shin, Oleg Sokolsky, Jon Sprinkle, Greg Sullivan, Janos Sztipanovits, Ashish Tiwari, Attila Vizhanyo, and Brian Williams for their support.

References

[1] Model-Driven Architecture http://www.omg.org/mda/
[2] Jack Greenfield, Keith Short, Steve Cook, Stuart Kent: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Addison-Wesley, 2004.
[3] IBM Rational Software http://www-306.ibm.com/software/rational/
[4] Borland Enterprise Studio http://www.borland.com/estudiojava/index.html
[5] Hausi A. Muller, Ronald J. Norman, Jacob Slonim: Computer Aided Software Engineering, Kluwer Academic, 1996.
[6] Matrix-X tools http://www.ni.com/matrixx/
[7] Matlab, Simulink and Stateflow tools http://www.mathworks.com
[8] Yannis Smaragdakis and Don Batory: Application Generators preprint from Encyclopedia of Electrical and Electronics Engineering, (John Wiley and Sons), available from http://www.cc.gatech.edu/%7Eyannis/generators.pdf
[9] Software Through Pictures product http://www.aonix.com/stp.html
[10] Borland's Together tools http://www.borland.com/together/
[11] VxWorks Real-time Operating System, http://www.windriver.com/products/device_technologies/os/vxworks5/
[12] OMG's CORBA http://www.corba.org/
[13] Java 2 Platform Enterprise Edition http://java.sun.com/j2ee/
[14] Microsoft .NET http://www.microsoft.com/net/
[15] Karsai, G.; Sztipanovits, J.; Ledeczi, A.; Bapty, T.: Model-integrated development of embedded software, Proceedings of the IEEE, Volume: 91, Issue:1, Jan. 2003 Pages:145 – 164
[16] Leonardo de Moura, Sam Owre, Harald Ruess, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari: SAL-2, Tool description presented at CAV 2004. Appears in Springer Verlag LNCS 3114, pp. 496-500.
[17] Karsai, G., Maroti, M., Lédeczi, A., Gray, J. and Sztipanovits, J., "Composition and Cloning in Modeling and Meta-Modeling," IEEE Transactions on Control System Technology (special issue on Computer Automated Multi-Paradigm Modeling), Dec 2003
[18] Karsai, G., Lang, A., Neema, S.: Design Patterns for Open Tool Integration, Vol 4. No1, DOI: 10.1007/s10270-004-0073-y, Journal of Software and System Modeling, 2004.
[19] Karsai G.: Design Tool Integration: An Exercise in Semantic Interoperability, Proceedings of the IEEE Engineering of Computer Based Systems, Edinburgh, UK, March, 2000.
[20] Tod Hagan, John Walker: Conceptual Data Model Evolution in Joint Strike Fighter Autonomic Logistics Information System of Systems Engineerin, White Paper, available from http://www.modusoperandi.com/Enterprise_Info_Integration/white-papers.htm
[21] Grzegorz Rozenberg, "Handbook of Graph Grammars and Computing by Graph Transformation", World Scientific Publishing Co. Pte. Ltd., 1997.
[22] Karsai, G., Agarwal, A., Shi, F., Sprinkle, J. On the Use of Graph Transformation in the Formal Specification of Model Interpreters, Journal of Universal Computer Science, Volume 9, Issue 11, 2003.
[23] Karsai, G, Sandeep Neema, Ben Abbott, David Sharp, "A Modeling Language and its supporting Toolset for Avionics Systems," Proceedings of the IEEE Digital Avionics Systems Conference, 2002.
[24] Personal communication from engineers of a major aircraft manufacturer. Evaluation results on record with USAF/AFRL and DARPA.
[25] Henzinger, T.A.: The Theory of Hybrid Automata. In Proc. of IEEE Symposium on Logic in Computer Science (LICS'96), pages 278--292. IEEE Press, 1996.
[26] The Hybrid System Interchange Format, available from: http://www.isis.vanderbilt.edu/Projects/mobies/downloads.asp
[27] Alur, R., T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky, "Hierarchical Hybrid Modeling of Embedded Systems." Proceedings of EMSOFT'01: First Workshop on Embedded Software, October 8-10, 2001
[28] Christopher Hylands, Edward A. Lee, Jiu Liu, Xiaojun Liu, Stephen Neuendorffer, Haiyang Zheng, "HyVisual: A Hybrid System Visual Modeler," Technical Memorandum UCB/ERL M03/1, University of California, Berkeley, CA 94720, January 28, 2003.

[29] E.M. Clarke, A Fehnker, Zhi Han, B. Krogh, O. Stursberg, M. Theobald. Verification of Hybrid Systems Based on Counterexample-Guided Abstraction Refinement. Proc. TACAS'2003.
[30] David Corman, Jeanna Gossett, Dennis Noll: Experiences in a Distributed, Real-Time Avionics Domain-Weapons System Open Architecture, IEEE IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, pp 307- 315, 2002.
[31] Brian C. Williams, Michel Ingham, Seung H. Chung, and Paul H. Elliott. January 2003. “Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers," Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software, vol. 9, no. 1, pp. 212-237.
[32] Agrawal A., Karsai G., Ledeczi A.: An End-to-End Domain-Driven Software Development Framework, 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Domain-Driven Development Track, Anaheim, CA, October, 2003.
[33] Ledeczi, A.; Bakay, A.; Maroti, M.; Volgyesi, P.; Nordstrom, G.; Sprinkle, J.; Karsai, G.: Composing domain-specific design environments, IEEE Computer, Nov. 2001, Page(s): 44 –51.
[34] Teja product http://www.teja.com