Improving the Integration of a Domain-Specific Modelling Tool

Steven Kelly
MetaCase Consulting
Ylistönmäentie 31
40500 Jyväskylä, Finland
+358 14 4451 401
stevek@metacase.com

ABSTRACT

Domain-specific modelling has proved its worth for improving development speed and dependability of applications. Current tool implementations tend however to be monolithic, and to require the modelling tool to be at the root of the tree of tools used. This paper examines the requirements for integration and how this was carried out in one modelling tool.

Categories and Subject Descriptors

D 2.2 [**Design Tools and Techniques**]: *Computer-aided software engineering* (CASE)

General Terms

Design, Languages.

Keywords

Tool integration; domain-specific modelling; API; XML

1. Introduction

CASE tools have been in the mainstream market for some time now. The largest tools have been incrementally improved with ever more features for tighter integration with other software development tools. All have some kind of code generation, many support reverse engineering, most have a link to popular version control systems. Some of these links have matured to the extent that the user need not always think exactly what the tool is doing: the integration starts to appear seamless, at least in the simplest cases. Although these links in general have not followed standards such as ECMA [1] and PCTE [6], and integration is still a problem area for most projects using CASE tools, the current facilities do prove useful in practice.

However, whilst the products are now more mature, CASE tools have not actually fulfilled their promise of raising the level of abstraction upwards away from code and towards design concepts. One main reason for this is the adoption of UML as the de facto standard method. The core diagrams of UML are based around the concepts of the code world: classes, member variables, functions etc. The positive side of having a standard understood by all is also its downfall: in trying to be all things to all men, it can never be the best solution for any given situation. To put things another way: by tying itself to the terms of the solution space (object-oriented languages), UML fails to provide designers with a tool to help them think in their problem spaces.

1.1 Domain-Specific Modelling Environments

A rapidly growing school of thought addresses these shortcomings by stressing that development should be domain-specific. A domain can be considered here as the kind of applications that a given company makes – on the requirements and design levels, not the implementation level. In particular, domain-specific modelling is based on the idea of creating a modelling language for a given domain [4]. The concepts of the language are the concepts of the requirements and design – typically the kinds of things that are described in a user manual for the end-product. For instance, in a language for the domain of mobile phone user interfaces, the concepts (object types) could be the buttons of the phone, its icons, menus, ringing tones etc. [5].

The language will also contain rules and ways of linking the concepts (relationship types) such as a button being linked to an icon via a 'turn on' message. Further, and most importantly, domain-specific modelling (DSM) aims at 100% code generation: a domain-specific code generator defines how the models are turned into code. That code is of course based on the components, frameworks, architecture and platform in use in that company. Since the code generation rules are written by an expert developer in that domain, the resulting code is generally of higher quality than hand-written code. It is also far better tuned for that domain than code generated from a 'one size fits all' generator in a standard CASE tool [7].

In a DSM environment, then, development takes place at a level which roughly corresponds to normal high level design. Low level design and coding are no longer needed, and testing is distilled to finding functional problems rather than correcting syntax errors and hunting down memory leaks. Perhaps because of this concentration of functions and raising of the level of abstraction, domain-specific modelling environments have tended to form the top of the chain of tools where they are used. The models are built in the environment, and from them code is generated, documentation reports are produced, test scripts are run etc. Output from the tools is subjected to further tests and versioned, but there is often no input to the tools other than through their modelling GUI.

This situation works well initially, but in the long term it often occurs that there are in fact other tools involved upstream, whose output should be used as input for the DSM environment. One example would be tools for requirements analysis, another would be lists of component interfaces that can be used in the models.

Similarly, there is often a need for tighter integration with tools further downstream. Many of these requirements are shared with CASE tools, such as integration with version control systems or controlling the build process.

1.2 MetaEdit+: A MetaCASE Tool for DSM

MetaEdit+ [3] is a metaCASE tool – a CASE tool which can support virtually any modelling language, and includes tools for defining such languages. A metaCASE tool often forms the best way to build a DSM environment, rather than trying to build the whole CASE tool functionality from scratch, extensive CASE tool functionality is already supplied, and the domain-specific modelling language can quickly be defined to the tool.

As a basis for a DSM environment, MetaEdit+ in its current 3.0 version offered all the facilities necessary for integration with tools further downstream: code and documentation generation, linking models to external files, and calling of external tools as part of generation. It did not however offer any facilities for importing data from other tools, or for being controlled by other tools.

A major part of the enhancements for version 4.0 consist of improving the support for integration. Extensive command-line arguments allow other tools to call MetaEdit+, e.g. as part of a build process ("open this database, run this code generation, exit"). An API allows data integration: external tools can read and write model data, allowing automated creation and update of parts of models. It also allows some control integration [8] to extend that provided by the command-line arguments, e.g. for highlighting active objects during simulation. Finally, XML import and export allows creation and processing of models, for simple data integration with other tools, and for applying wholesale transformations to models. In the rest of this paper we will look at the API and XML import/export.

2. API

As MetaEdit+ runs on many platforms (Windows, Linux, HP-UX, Solaris etc.), the choice of API protocol was particularly difficult. At the start of 4.0 planning, it seemed no satisfactory solution existed. Most protocols were either too platform-specific (e.g. COM, OLE), too language-specific (e.g. Java RMI, C calls), too low-level (e.g. RPC, sockets) or too heavyweight for simply linking one calling routine with one tool (e.g. CORBA). In total, we considered 17 possible protocols, and one rose above the others in that it was standards-based, modern and growing fast, vendor, platform, and language neutral with support for many of each, high-level and relatively easy to use for new users. Unusually given these attributes, it also later was chosen as Microsoft's protocol of choice for inter-tool integration.

The protocol was of course SOAP, also known as Web Services. Usenet references to "SOAP/WebServices/.NET API" increased from 200 in 2000, to over 500 in 2001, to nearly 1000 projected for 2003. Similar searches on the Web showed an increase of a factor of nearly 5 over the same period. In contrast references to APIs based on COM, OLE, CORBA and C++ were all declining.

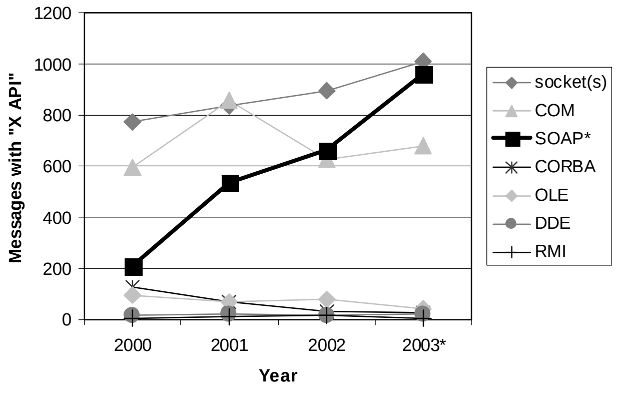

Figure 1. API Protocol Discussions on Usenet

SOAP was originally intended for use where client and server were separated over the internet, whereas most API use takes place between two processes running on the same machine. Surprisingly, this feature of SOAP has led to some benefits for use as an API protocol. Because of the difficulty of remote debugging, SOAP has strived to make the process of calling a remote function as seamless and invisible as possible, whilst still providing access to the clear-text form of the low-level communication if necessary. Making the function call seamless and invisible is achieved through a WSDL – an XML description of the functions provided by the SOAP server (in this case, MetaEdit+) – and a client framework and tools for each language that will call the API. The client tools for almost all languages include functions to read a WSDL description and generate code that wraps those calls in a language-appropriate way. Conversion of parameter value types is handled automatically by the framework, as is the low-level business of formatting the call into an HTTP request with XML content. Calling a function to open a given diagram and zoom it to 50% can thus be as easy as:

```
openDiagramAndZoom("MyDiagram", 0.50);
```

2.1 API Functions

The API functions in MetaEdit+ can be divided into control functions, read functions, write functions and utility functions. Control functions allow basic operations such as logging in to a repository containing models, opening a project, opening a window on a model, highlighting a given object in a model, and running a code generator.

Read functions and write functions allow access to the model information. MetaEdit+ includes both conceptual information (an object's name and other properties, and links to other graphs) and representational information (where an object is positioned on screen, how big it is). Currently, only conceptual information is supported. Similarly, only model information can be accessed, not in general meta-model information (DSM language definitions).

Utility functions help to bridge the gap between the real objects in models and their string representations. Every object in MetaEdit+ has a unique numerical ID, which will not change even if its name is changed. Similarly, two different objects can have the same name but will be distinguishable by their ID. The ID is thus what is normally passed via the API to

[*] Data for 2003 extrapolated from figures up to 5th August. SOAP figures include references to WebServices and .NET.

identify an object for an operation. Since users will not in general know IDs, utility functions are provided to find an object based on its name and the name of its type. Of course, in API client programs the IDs are not normally visible, as they are referred to via variable names:

```
topObject=findObject("UpArrow", "Button");
highlightObject(topObject);
setName(topObject, "CursorUp");
```

This approach differs from what is often found in SOAP APIs. There the client program usually contains data structures that mirror those in the server, e.g. both would have an Address type with street address, city and country. Whenever an address is passed, it would actually be a copy of the contents of that address: there would not by default be a way to receive an address, modify it, and update the original address on the server. Such an approach works, and indeed makes life easier, providing the data in question is not highly interlinked, and has no persistent identity of its own. A given customer could have his address replaced with a new one, and the old address would simply disappear. In MetaEdit+, all data is highly interlinked, and if a Customer object has his Address object replaced, the old Address object will still exist, and probably be referred to by other objects.

This interlinking of data also gives rise to another problem for the normal approach: if the client requests a Customer object, how can we avoid sending the contents of the Address object it contains. Requesting the top-level graph – or any object that refers to it – could involve transmitting the whole contents of the repository.

Thus in MetaEdit+, and probably any other metaCASE tool, it makes more sense to send objects through the API as unique identifiers. If the client wants to know some more details about the object – e.g. its name or type, or its 'street address' property – he can send an API call with the object ID as an argument:

```
topType = type(topObject);
```

This approach also has the benefit of significantly reducing the complexity of the data structures used by the API, which would otherwise form a large and difficult part of the learning curve for new API users.

2.2 Problems

One downside of SOAP APIs is their performance: because of the work involved in formatting (marshalling) objects to and from HTTP/XML, and the considerable freedom that is supported in that, each operation takes a significant number of milliseconds to execute. The actual time is more dependent on a machine's TCP/IP settings than its speed: with the wrong DNS and proxy settings, results can be ten times slower. This rules out API use for functions requiring many thousands of calls, and motivates the creation of larger functions that replace what would otherwise be a series of calls.

Another problem is the difficulty of integrating utility functions for user-defined data types passed via SOAP. Whilst the SOAP client tools will create correct language-specific versions of the data types based on the WSDL, they will only have simple getters and setters. Whilst other functions can be added by hand, this then means the API provider is responsible for providing a separate SDK for each client language, rather than simply one WSDL which can be used by all client languages.

2.3 Use Case

MetaEdit+ already included an example DSM for digital watches, complete with full code generation to Java for execution in the user's web browser [7]. Part of the DSM represents the behaviour of each watch application, e.g. Stopwatch, with a customised form of State Transition Diagrams. With a few minor extensions to the example, and the installation of a Java SOAP framework, this example now makes a SOAP API call to MetaEdit+ whenever a new state is entered:

```
animate(graphID, stateObjectID);
```

This call causes MetaEdit+ to open the graph (if it is not already open), and highlight the specified state object. In this way the user can run the resulting watch applet in a web browser, and follow at each stage as MetaEdit+ traces the execution from one state to the next.

This kind of visual tracing or simulation functionality is particularly important in DSM, where the idea is to insulate the developer from the details of the code as much as possible. Now debugging can be carried out conceptually on the model level, rather than trying to 'reverse engineer' from a given problematic line of code to the object from which that line of code was generated. One large customer is already working to take advantage of this approach in their use of MetaEdit+, and estimates a 50% reduction in testing time because of it.

3. XML Import/Export

If the choice of API protocol had been difficult, the choice of a textual format for import/export also presented its own challenges. The obsolescent CDIF and PCTE DDL could be ruled out: neither were ever much used, and CDIF only supported a fixed set of modelling languages. The basic choice of XML seemed certain, but the schema / DTD was harder to find. Previous experience with XMI had shown it to be irretrievably entangled with UML: whilst in theory it should have been able to support other modelling languages, in practice it was barely able to support UML, and could never be extended to support the wider range of modelling languages possible with MetaEdit+'s GOPRR.

For a metaCASE tool, import and export can cover both model data, and metamodel data (DSM language definitions). Initially it appeared that the best approach would be to have the model data as XML, and the metamodel as a DTD or schema. This however would require each DSM language to have its own DTD or schema. As most programs using XML seem to embed details of the DTD or schema throughout their code, on reflection this seemed a poor idea. Better would be to keep one DTD or schema, raised one level higher: instead of tag names being type names, e.g. <Button>, they would be meta-type names, e.g. <Object>, corresponding to the five meta-types of GOPRR.

Comparatively late in the development of 4.0, the authors happened across an XML language that offered precisely that: GXL [2]. In addition, GXL was grounded firmly in the world of metaCASE, as some of its authors had been involved in making a research metaCASE tool.

3.1 Issues

GXL is in a state of flux, with the whole representation of objects' types changing significantly. Fortunately – if unsurprisingly – this change and several others actually make it closer to fitting well with MetaEdit+. As with the API, we made a decision to only support conceptual data, not representational or types' internal data at this stage. Somewhat surprisingly, even our most advanced customers agreed that the extra power of being able to manipulate representations or types would be more than offset by the extra complexity of working with the language.

Whilst GXL has as its apparently primary relationship construct only a binary relationship, it also offers an n-ary relationship. In MetaEdit+, all relationships are treated as a binding, which contains a relationship, and two or more roles, each connected to an object. In 4.0, roles connect to objects via ports. Relationships, roles and ports are all first-class members of models, with their own types, properties etc. GXL, however, only has relationships as first-class members: roles have no type, and ports do not exist. GXL does however include mechanisms for adding extra information to the DTD, offering a solution to both omissions.

An unexpected problem occurred when GXL revealed problems in the XML library used in MetaEdit+, in particular in the way unique objects are dealt with. Even after corrections to the library, it seems apparent that there are actually several different use cases for GXL use in MetaEdit+, and each will need a slightly different way to map XML to model data. For instance, if a user wants to export a model, manipulate it, and re-import it, it is important to maintain links to the unique IDs of objects, and to maintain each unique object as a single object, even if it is referred to by several other objects. In contrast, when importing or exporting simple objects with Excel, only the barest information need be supplied. If communicating with another GXL tool, the resultant XML should be as close to the GXL DTD as possible. To some extent these differences can be effected with optional elements, but not in all cases.

3.2 GXL Postscript

After writing this paper, it became increasingly apparent that ever larger amounts of effort would have to be spent on reconciling the differences between the GXL data model and that of MetaEdit+. For some major areas, no satisfactory solution could be found. As all of the XML import/export users would be familiar with the MetaEdit+ data model, and virtually none with GXL, there was little choice but to abandon GXL and construct our own XML format.

This format uses the same structures as MetaEdit+, making learning it simple for users, and of course making it far easier to develop. Additionally, as MetaEdit+ evolves, the XML format can evolve in the same way: with GXL, the two ends of the import/export would have evolved separately, producing an ever greater strain on the import/export code between them.

A sad outcome from the point of view of GXL, which is as good an attempt at a generic CASE data exchange format as I have seen, but hopefully the right decision for the actual users of the import/export functionality in MetaEdit+. Perhaps if GXL continues to evolve towards MetaEdit+, it could be offered as an alternative XML format in a future release.

4. Conclusion

Perhaps the hardest part of adding integration functionality to a tool is trying to guess how users would want to use it. By its very nature, it opens up a much wider world of possibilities than the addition of a normal feature in the tool's GUI. These problems are compounded when working with a meta-tool, and further so when working on multiple platforms.

Until something concrete is available to test, and the users have a concrete problem they want to solve, most are unable to offer any practical requirements. Integration is necessary, but because of these problems it is wise to remember the rules of low coupling and high cohesion: anything that can be accomplished without integration, should be.

The adoption of XML as a standard has had wider repercussions for integration than all previous integration work. By raising the level of abstraction from a character stream to a tree of elements, both APIs and export formats can be accomplished with significantly less effort, both for the tool provider and the client. As most users will not have previously made API calls or import/export links to any tool, having a simple format with good, free frameworks for all languages is a significant help in making integration work.

5. References

[1] ECMA: Computer Systems Laboratory, National Institute of Standards and Technology (NIST), and European Computer Manufacturers Association (ECMA). Reference Model for Frameworks of Software Engineering Environments, 3rd ed. Technical Report NIST 500-211, ECMA TR/55, 1993.

[2] GXL, www.gupro.de/gxl (April 2003)

[3] Kelly, S., Lyytinen, K., and Rossi, M., "MetaEdit+: A Fully configurable Multi-User and Multi-Tool CASE Environment," In Proceedings of CAiSE'96, Lecture Notes in Computer Science 1080, Springer-Verlag, Heraklion, Crete, Greece, May 1996, pp. 1-21

[4] Kelly, S., Tolvanen, J-P, Visual domain-specific modelling: Benefits and experiences of using metaCASE tools, In: International workshop on Model Engineering, ECOOP 2000, Ed. J. Bezivin, J. Ernst (2000)

[5] MetaCase, Nokia case study, http://www.metacase.com/papers/ (1999)

[6] PCTE: ECMA, Portable Common Tool Environment - Abstract Specification, Standard ECMA-149, 4th Edition, December 1997

[7] Pohjonen, R., Kelly, S., Domain-Specific Modeling, Dr. Dobbs Journal, August 2002

[8] Wasserman, A.: Tool integration in software engineering environments. In F.Long (ed.), Software Engineering Environments, Springer-Verlag Berlin, pp. 138-150, 1990