

Issues in Large Scale Application Development with Model-based Techniques

January 2007

Timothy A. Anderson
Basil C. Krikeles

BAE Systems
Advanced Information Technologies
6 New England Executive Park
Burlington, MA 01803

ABSTRACT

Model driven technologies for software development are being considered and used to address issues of size, complexity, adaptability, maintainability, distribution, and validation for large, enterprise-wide software systems. The Object Management Group has been promoting their Model Driven Architecture (MDA) standards, while various academic groups and commercial companies have concurrently been developing model driven techniques, versions of the executable Unified Modeling Language (xUML), and domain specific modeling languages. Over the past few years, large software projects have attempted to employ model based approaches. In this paper we assess the efficacy of MDA during the full development cycle, including: observations on the effectiveness of the model development tools; the practicality of using xUML in a large software development project in terms of managing model complexity, the expressive power of the associated 4th generation language, and the creation of links from xUML to a large conventional code base; and validation, verification, configuration management, runtime performance, and debugging issues that are unique to this approach. Based on our experience with current MDA tools, we conclude with a set of desirable properties for the next generation of tools, especially those needed to support large scale distributed development of network-centric applications.

1. Introduction

Increasingly sophisticated software systems are needed to meet complex and evolving DoD requirements. The size of software components, either measured simply in terms of number of lines of code or in terms of more sophisticated complexity metrics, has been steadily increasing. Today, sophisticated software systems, for example operating systems or airplane flight control systems, can consist of tens of millions of lines of code, with the prospect that soon hundreds of millions of lines of code may be required in order to implement certain systems. We may be close to the limit of what human beings can accomplish with current technology. There are pervasive problems with large, monolithic systems, including brittleness (fixing a defect results in

additional defects appearing elsewhere), decreasing developer productivity, and increasing maintenance costs. For the DoD this presents a worrisome prospect as costs skyrocket and the quality of the delivered software declines. A succession of software development methodologies, technologies, processes and computer languages have been used to address this problem. Humans are good with abstract concepts but less talented when dealing with the minute details necessary to communicate with computers at the machine level. To bridge this gap, computer languages have evolved to abstract the machine-level details, starting with assembly language, FORTRAN and C, and continuing with today's object oriented and dynamic languages. Similarly, development tools have progressed from functional programming, to CASE tools and UML. The goal is to bridge the abstraction gap between human and machine by allowing the human to manipulate higher level artifacts that can then be compiled into an executable. These tools frequently over-promised and under-delivered including for example a variety of CASE tools in the late eighties and early nineties. The core of the problem is that it is difficult to develop a tool that can infer from an abstract representation (a model) the details of a specific instantiation of the model. Therefore, a multi-faceted approach may be needed, and tools may need to support multiple abstraction levels in their representations. As the capabilities of hardware, networks, and software expand, the problem areas that can be addressed become more complex and more difficult; it is critical, and at least in principle feasible, to provide tools that will support development of these complex applications throughout their life cycle.

2. Model Driven Development and xUML

Universal Modeling Language (UML) was not designed to be executable, although even early versions included a limited set of actions such as sending a signal, and creating or destroying an object. In 2001 UML was extended with action semantics, a more complete set of abstract actions including for example facilities for manipulating collections of objects. Executable UML (xUML) comprises a subset of the UML standard with sufficiently precise semantics to be capable of being executable over some form of virtual machine. xUML is intended as a generic attempt to support OMG's Model Driven Architecture (MDA). xUML implementations typically include the following:

- support for a well-defined simplified model structure, a subset of full UML
- an implementation of a specific syntax (vendor-specific) for the precise action semantics of the UML standard
- a simulation infrastructure that enables the entire lifecycle of model development (creation, editing, execution/simulation, configuration management, project support)

After discussing an alternative approach to MDA that involves the use of Domain Specific Modeling Languages (DSMLs), and after describing two examples of such languages, we will discuss in detail our experiences with one large scale software project that adopted the xUML approach. Our discussion will include some the the drawbacks of the xUML approach, as well as some concrete suggestions for improving the successor to xUML (currently under standardization by the OMG).

2.1. xUML Tools

There is a wide variety of general tools that support model driven development, including many that support some form of executable UML (xUML). We focus here on what we've learned from the extensive use of a particular tool; this is a case study, not a trade study. The tool we've been using is generally compliant with existing xUML descriptions [2][10]. Obviously some of the issues we've found are specific to the tool's interpretation of xUML, but most are related to the way xUML is defined and intended to be used.

2.2. Domain Specific Languages

One approach to model driven development involves the use of languages tailored for a specific domain, where “domain” is rather broadly interpreted—our examples include the domain of access to tactical data links, as well as the domain of distributed application development. The use of such languages allows one to build an efficient solution to a particular problem in the domain on a more general platform that makes it easier to express those solutions, and that can handle many of the domain’s general problems in a consistent manner across a range of applications. It’s a step up from a well-defined API to a support library, because the language is at a higher level, and can make it easier to construct the application.

2.2.1. XML Defined Gateway (XDG) for Tactical Data Links

The XML-Defined Gateway (XDG) [11] is a router/gateway for tactical data links (TDLs), combining a conventional forwarding/filtering engine with dynamically loadable modules specialized for each supported TDL. It is intended to address a number of problems:

- Specifications for message formats and transmit/receive rules on TDLs are usually only human-readable, making it difficult to verify the correctness of any particular implementation.
- The number of data links in active use is not small (Link-11, Link-16¹, VMF, 6020 for internetworking...), and is likely to increase. Support for all of these needs to be built in a way that minimizes the opportunity for programmer error, without sacrificing performance.
- Message forwarding among a variety of TDLs can be an N² implementation problem: each pair of network types, in a naïve implementation, would require a module to handle translation.

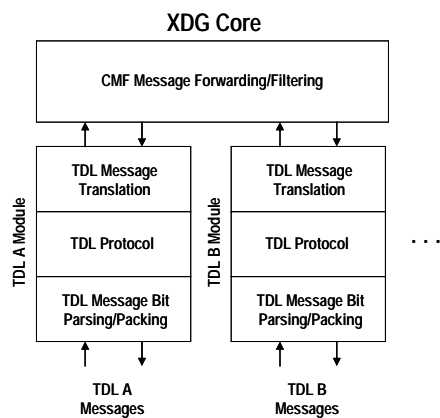


Figure 1. XDG architecture

In XDG, the forwarding engine is a fairly standard C++ implementation. It manipulates messages in a Common Message Format (CMF) that is independent of any particular TDL, and that is defined by an XML document that the forwarding engine loads at runtime—further insulating the engine from changes required to support new messages or new TDLs. See Figure 1 **Error! Reference source not found.** for the split between the core and the dynamically-loaded TDL support modules.

Each TDL for which support is required is completely specified by an XML document, the schema for which defines a domain-specific modeling language. The TDL specification describes message formats,

transmit/receive rules, and other aspects of the protocol; rather than being used at runtime, it is compiled into a C++ module that can be loaded into a running instance of XDG. The TDL-specific module handles conversion of inbound messages into CMF and of CMF into outbound messages on the link, while providing enough information for the forwarding engine to route messages to the appropriate destination.

The XML document describing a TDL can serve both as the implementation of XDG’s support module for it, and as a human-readable specification of the network. Translation of the domain-specific language into C++ provides the module; XSLT transformations of it can produce a

¹ For Link-16, the standard is 6016C, but it’s recent enough that some systems are still on 6016B.

readable document. Thus a correction in a TDL standard can be made in one place, updating both the implementation and the documentation simultaneously.

This is a solution where every aspect is tailored to the specific domain. The language, as defined in the XML schema, is intended only to describe TDL messages and their processing. The runtime implementation is a combination of conventional development with dynamic modules generated from XML documents in the domain-specific language. Although the specific runtime configuration, including the definition of the basic data structures manipulated by the system, is controlled by an XML document, XDG does not otherwise use XML at runtime: messages enter and leave the system in the format native to their specific TDL, and within the system they're represented in the binary CMF. This approach provides high performance, a high degree of flexibility, and dynamic upgrades to a running system, using off-the-shelf tools to edit the models and build the required transformations.

2.2.2. eXtensible Distributed Architecture (XDA)

XDA is a model-driven framework whose goal is to address the problem domain of distributed application construction. Its development began in 1998, based on lessons learned from large multi-vendor programs. Typically, these programs require vendors to develop components that address different aspects of the same problem domain and to produce solutions that can be easily integrated into a cohesive distributed system (or system of systems). Frequently, the difficulty involved in accomplishing this integration is not fully appreciated at program startup. In addition to the core system integration issues of managing component complexity, system configuration in a networked environment, and semantic alignment between the components, there are the additional complications of evolving system requirements and a changing definition of the problem domain, in the context of multiple development teams operating remotely. XDA was designed to provide a foundation for addressing some of these issues; specifically, XDA provides the ability to:

- Design a model of the shared domain of interest that forms the foundation for communication between components.
- Automatically generate platform neutral implementations of the domain model in Java and standard C++.
- Generate a transport infrastructure that includes configurable platform-neutral serialization strategies for the domain entities, including XML serialization with an automatically generated XML parser/writer for the domain entities.
- Hide the details of the underlying middleware by utilizing XDA provided libraries with well-defined generic APIs.
- Gracefully evolve the shared domain of interest in a constrained and disciplined way with minimal disruption across multiple development teams.
- Do all of the above consistent with good runtime performance.

One approach that can be used to address many of these issues is to partition the system into well-defined components that expose a well-defined interface using standard communications protocols (such as Web Services). The resulting system is then described as a Service Oriented Architecture. The SOA approach minimizes the coupling between components, but does not systematically address the issue of domain architecture and runtime performance. Since XDA was initially designed for the domain of distributed sensor fusion, runtime performance was an important consideration.

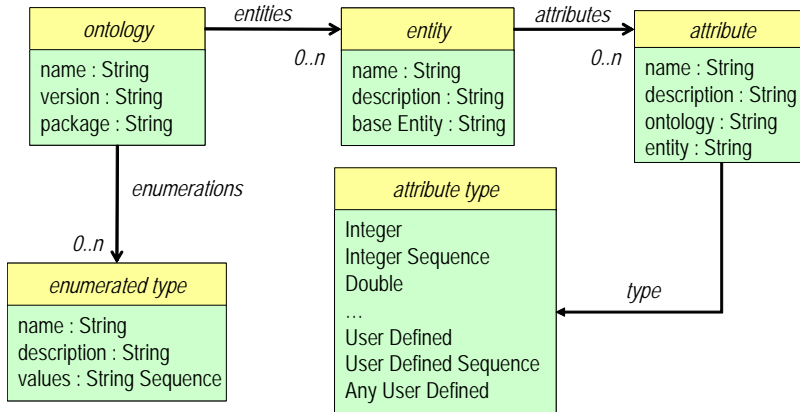


Figure 2. The XDA meta model captures inheritance and aggregation relationships

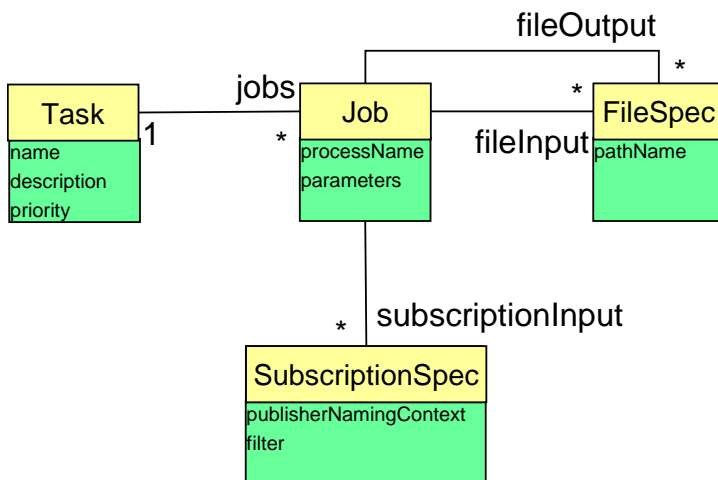


Figure 3. A partial representation of a distributed processing domain based on the XDA meta model

XDA uses a meta model that is analogous to OMG's MOF and Eclipse's Ecore but much simpler. The XDA meta model is designed to define classes (domain entities) with single inheritance and aggregation relationships as shown in Figure 2. Code generators can easily traverse the models that are

created based on the XDA meta model and can be configured to generate

efficient code. A very simple example of such a model which is part of a distributed processing domain appears in Figure 3.

Automated (model-driven) code generation for the shared domain structures provides many benefits, especially for projects involving remote collaboration among multiple teams. First, the shared domain model provides strong semantic hints for proper exchange of information. Second, the shared code is less likely to be error-prone and can be fixed by adjusting the generator at a central location. Finally, the generated

code supports substitutability of base classes by derived classes, thus permitting domain model evolution as new requirements are uncovered. The model-drive approach allows for easy extensions to the shared domain to support unforeseen requirements. For example, a new requirement that might emerge might involve the need to identify the processing path as well as a causal trace (antecedent information) that resulted in a certain piece of information appearing as an output of the system. Information Pedigree capability can be modeled using this approach and can be integrated into the shared domain of interest with minimal disruption to the overall system.

XDA is designed to be middleware and computing platform agnostic, enabling the construction of Platform Independent Models (PIMs). It can be bound to a specific platform, for example Linux x86 with CORBA middleware using an XML serialization protocol, thus producing a Platform Specific instance of the Model or a PSM.

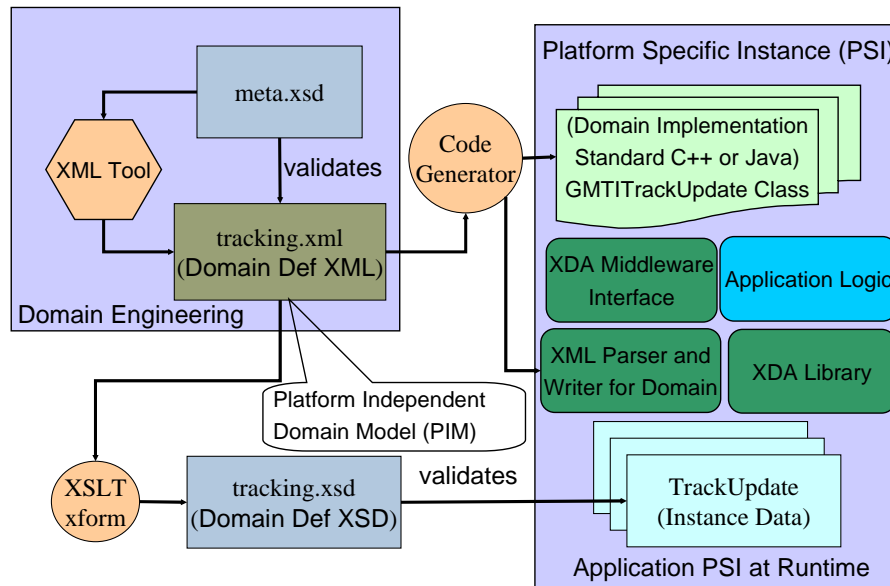


Figure 4. XDA enables automated generation of the shared domain definition as a PSM

As shown in Figure 4, the XDA framework can be used (in this instantiation of it in terms of XML-based transport) at design time for Domain Engineering. The XDA meta ontology of Figure 2 is realized in terms of an XML schema, allowing subject matter experts to use a standard XML tool to develop a definition of the problem domain in XML. The XML version of the ontology serves two purposes; it allows the user to operate in conformance with an XSD so as to guarantee the validity of the result, and it is structured with additional information that is helpful to the code generator. An XSLT transform is used to convert the domain XML to a domain XSD that can be used to validate instance data both offline and online. The Code Generator generates the domain implementation in C++ or Java which, along with the appropriate XDA core and middleware libraries is used to assemble a PSI for each component.

2.2.3. Other Languages

One of the claimed advantages of the MDA approach to software development is that it makes it easier for subject matter experts who are not software engineers to contribute to the design and verification of the software solution. The use of a modeling language permits the domain experts' knowledge to be represented in a way that they can understand, that a software engineer can understand, and that software can understand.

XDG and XDA contain examples of domain-specific modeling languages (DSMLs) defined without using UML. However, there is a considerable amount of effort being devoted to the development of DSMLs using UML. As with XDG, which can take advantage of the considerable toolset that has developed around XML, UML-based languages can use the emerging UML toolset to provide much of the support that developers need.

As France *et al.* [5] discuss, UML 2.0 provides three distinct mechanisms for tailoring: semantic variation points, profiles, and use of the Meta-Object Facility (MOF). These provide increasing flexibility, but at the cost of increasing complexity in the underlying toolset, and increasing difficulty in maintaining a consistent definition of the modeling language being used.

Semantic variation points are locations in the UML standard where a precise semantics is deliberately not specified. These include some of the semantics associated with state machines, aggregation, and so on. The system developer can, by providing a full definition, tailor UML to be better suited for a particular application domain. However, it can be difficult to maintain

consistency, both with other parts of the UML specification and with other variation points that the domain may need to have tailored; in addition, pre-existing tools by definition will require modification if they are to analyze models built using these variations, or generate code from them.

UML profiles provide another extension mechanism, allowing the creation of new constraints and additional attributes on classes in the meta-model. However, the semantics of these new elements cannot be defined in the model, so once again the tools have to be built or tailored to match the extension.

Finally, UML provides the Meta-Object Facility, which allows customization of the meta-model. This is essentially unlimited in what it allows; the difficulty, as France *et al.* point out, is that it places the burden of language development on users.

The difficulties associated with these mechanisms for creating DSMLs are not specific to UML—rather, they are part of the cost associated with any new language. Analyzing the costs and benefits of creating a DSML is particularly difficult. For example, one would expect that the productivity improvements associated with a domain specific language would outweigh the implementation, training, hiring, and opportunity costs associated with the effort, but it's easy to optimistically ignore many of those costs during the analysis. Will the engineers you want to hire be willing to work in a system that is domain-specific? If not, will you be able to find domain experts who are good enough software engineers to build the system you need? What would the costs and benefits of a well-designed class library in a conventional language be, compared to the DSML? And how much will it cost to obtain engineers with the skills needed to build a new language—in our experience, a much more difficult task to do right than the definition and construction of a library.

For many domains and applications, the cost of supporting a domain-specific language will be too great to be practical—but certainly not for all. This is a tool development problem, and as with any tool, even a very large cost can be justified for a sufficiently large productivity gain by a sufficiently large number of customers or users. As the tools for developing DSMLs improve, the costs will decrease.

One approach, described by Balsubramanian *et al.* [6], starts with a tool for building and using DSMLs, the Generic Modeling Environment (GME). This supports both the meta-modeling required to define a specific DSML, and modeling using it. It explicitly does not attempt to generate all of the code for either the entire DSML implementation or the entire target application; instead, the system provides a framework that supports the integration of modules written in general-purpose languages. For example, as part of the DSML implementation, GME will provide external model interpreters with access to the model hierarchy it has defined, allowing them to validate the models and generate platform-specific code from them. The same authors' Embedded Control Systems Language (ECSL) [6], which targets embedded automotive applications, handles deployment and integration, including the calling of “application-behavior code,” which can be generated by the tool, generated externally, or hand-coded.

This is an important step toward making DSMLs practical. Real systems under development will often have access to a body of pre-existing code that works well enough; if it's easy to integrate this, development costs will be reduced substantially. Similarly, it may be desirable to produce heavily-optimized code by hand for some parts of the system, or to integrate code produced by a subject matter expert who is only comfortable and productive working in Matlab. These are not exceptional cases; better support of these sorts of integration can reduce costs significantly.

CALM [7] focuses explicitly on system modeling down to the component level, but does not attempt to support the coding of individual components. Thus what's being modeled is the

system; obviously the interfaces and connections associated with individual components are part of this, but the components themselves are black boxes. The system also supports the creation of composite components, but beyond that appears to entirely avoid the difficulty of generating efficient code by allowing the code to be written in standard languages. As a system for integrating components using existing middleware, such as CORBA or EJB, it is similar to XDA.

Although these systems are all described as domain specific, the domain is often a platform rather than an application area: integration using CORBA, for example. Each system we've discussed will impose costs in the form of training, documentation, and system maintenance that would not apply equally to general purpose COTS tools; the existence of standards, such as XSLT for XDG, and tools, such as Eclipse for CALM, in some cases has made the cost-benefit ratio quite favorable. XDG uses relative simple, standard tools to build a specialized line of products; the relatively low cost of the tools matches the relatively small number of engineers using the system. Other systems, such as GEM or CALM, are intended to be used in more general ways, so rightly focus much more on user interactions. Of course, in every case the value of the tool is in what it produces—the model, in whatever form, is not useful unless it can be transformed into something runnable.

3. Case Study

3.1. The Promise: xUML value proposition for the DoD

The promise of the xUML/MDA approach has significant cost implications for large systems that have largely similar business logic but are deployed on multiple, heterogeneous execution environments.

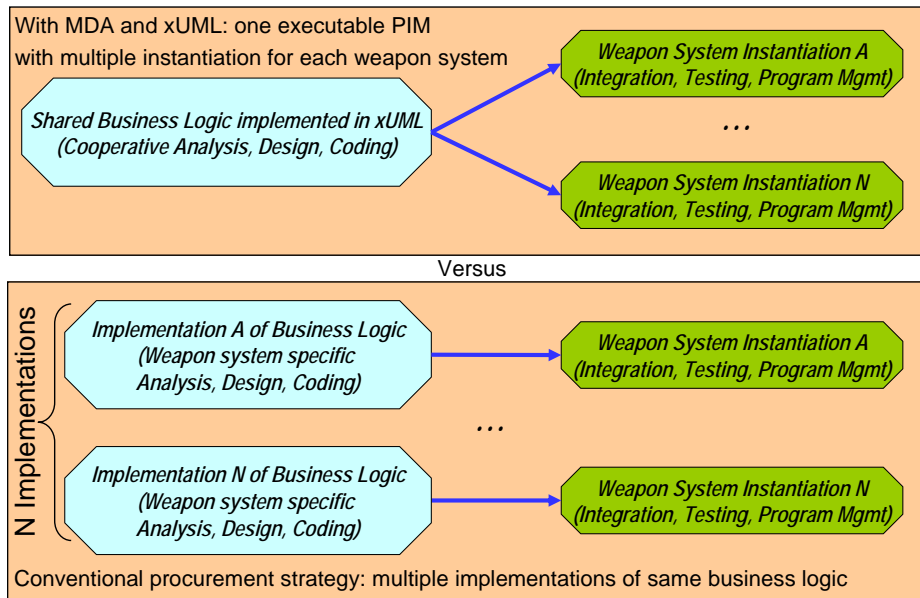
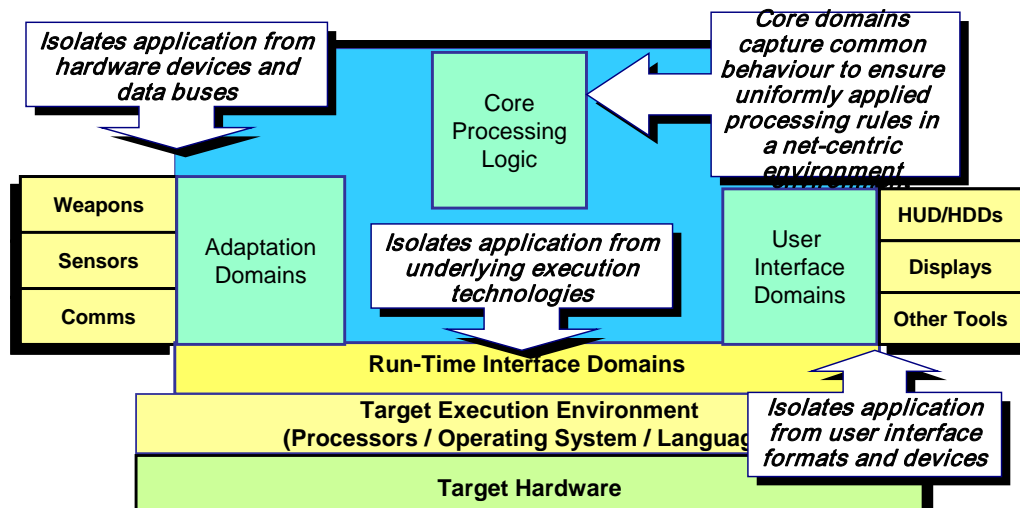


Figure 5. xUML represents a new approach to DoD procurements

As shown in Figure 5, the conventional approach is to generate a detailed specification for the system in human-readable form. Based on this specification, there is a procurement for analysis, design, implementation, integration, testing, and deployment for each execution platform. This causes significant duplication of effort, as the core logic is implemented multiple times for different execution environments. There is also a significant risk for undesirable variations in the

implementation logic because of potentially differing interpretations of the specification, especially in areas where it might be ambiguous. Finally, there are N implementations to debug and maintain. Since software maintenance is said to represent approximately 80% of the cost of software development, this approach to procuring large systems is quite expensive. By contrast, MDA/xUML promises to generate significant cost reductions by pooling resources across all the target execution environments to produce a single, executable platform-independent model. This model can be designed and implemented with the assistance of subject matter experts from all targeted execution environments, and tested and debugged prior to being released to the stakeholders to be transformed into platform specific instantiations.



The **Figure 6. Adapting a PIM to an execution platform**

shared business logic is developed once. Maintenance is easier, because defects can be fixed once. Costs are dramatically reduced, because each team implementing a PSI can concentrate on what is different about that instance, rather than worry about duplicating the core logic. Figure 6 shows the adaptation of a PIM to a particular execution environment. In addition to the Core Processing Logic, it contains domains that are designed to interact with the external execution environment. These adaptation and interface domains assume the availability of interfaces to the execution environment, including operating system facilities, sensors and displays. These interfaces are typically implemented as libraries that are linked into the final PSI.

It's expected that the upfront development costs associated with this program will be larger than they would for an effort of comparable scope developed using more traditional tools and techniques. Cost savings, if they're found, will come from two places: the amortization of the development costs for the core processing logic over many platforms, and the reduction of maintenance costs because any bug in the core need only be found and fixed once, and there is much less chance of differing interpretations of the specification.

3.2. Introduction to the Case Study Project

The authors have been associated with a Defense Department project to build a new piece of software using MDA. The product is targeted ultimately for a variety of weapons systems, covering the spectrum from large ground-based installations, through surface ships, to E-2 surveillance planes and even fighters. The value of the product is partially in the common implementation of algorithms, and partially in its intended ability to share data among systems: platforms that have access to broadband links can share data with their peers, while using TDLs to share information with legacy systems, or with other instances of the product that don't have

broadband. Data shared over the TDLs is, as expected, fully processed; other platforms receive track reports. Data shared over the peer-to-peer network is largely unprocessed: other platforms receive measurements, which they use to execute the core algorithms; thus all platforms arrive at the same result by executing common algorithms on shared data.

As discussed above, the product includes a substantial amount of conventional C++ code, both platform-specific and generic, but the core logic is being developed using MDA tools—in this case, Kennedy Carter’s iUML product [8]. The modelers on this program do their work on Windows machines, although weapons systems are typically expecting to use Linux, Solaris, or small RTOS environments such as LynxOS. The project poses major challenges in a number of areas: developing software that will allow small systems to keep up with data produced by peers with much higher capacity; ensuring that the software will in fact run on all of the target platforms; providing updates to the core software that can be integrated with adaptations made by the individual platforms; and performing adequate testing, at the unit, system, and system-of-systems levels.

In what follows, we will discuss specific issues that we’ve encountered in extensive testing and integration of the software product, performed in support of various Air Force platforms. In addition to three cases that reflect on the capabilities of the tool and of MDA principles, we will discuss separately the handling of configuration management, which is critical for a system being deployed as this one is, and the “model compiler,” which is responsible for producing conventional code from the iUML model.

This may go into a level of technical detail that not everyone will find interesting; more general observations can be found in Section 4.

3.3. Configuration Management

3.3.1. Description of CM support built into the MDA tool

The program’s choice of an MDA tool was driven in part by the desire to have core processing be coded once, then deployed essentially unchanged onto the various platforms. However, as we discussed in section 3.1, the deployment must also include code, developed either as external libraries or as additional iUML domains, that is specific to each platform—terminal interfaces, sensor interfaces, and navigation interfaces are just three examples.

The developers provide to the platforms, or in our case to a service lab set up to evaluate the product, a new drop of the core code, a “timebox,” every 6-12 weeks. This has had unit and some single-system testing in a simulated environment, but ultimately it will be deployed in a system-of-systems environment. Further, it contains placeholder implementations of some code that will almost always have to be customized by the platforms; this allows the core logic to be tested without platform-specific customizations. Our experience has shown, at least in the current rather early stages of development, that it is rarely possible for a platform to adopt a new drop without making some patches in the core code, in addition to whatever changes are required for their environment. The most obvious cause is a bug that affects only that platform, and for which the fix hasn’t yet made it into the development cycle.

The basic unit of code development and reuse in xUML tools is a “domain,” which is a collection of classes. Domains in this system range from relatively simple ones that provide basic services, such as starting all the components of the system, to a fairly complex, multi-person effort that manages the flow of data to and from tactical data links.

A given executable is produced from a “build set,” which specifies the set of domain versions to include, as well as the inter-domain bridges that allow domains to interact—they are otherwise treated as black boxes, but bridge code is allowed to make calls inside several domains. In

principle, bridge code should contain no business logic, because that eliminates the value of the encapsulation provided by domains; in addition, it is outside any domain, and so can't be moved around as a unit easily—the bridges used by a single domain cannot be treated as a unit for configuration management.

The tool stores domains and build sets in a proprietary binary format, although there is an API that allows access to the database. It's easy to export a specific version of a domain, as a binary file, and import it into another database; the export includes all of the UML diagrams as well as the code associated with operations defined in the domain. In fact, the basic code drop from the developers to the platforms is just a set of domain exports, along with an exported build set.

On this project, we have a widely distributed collection of teams trying to work on the same product, often on the same parts of the product. Admittedly, the core development happens only in a few places, but even in the core there are teams working on parts of the product, at a level where several people are (or should be) working simultaneously on a single domain. This has proved very difficult to manage effectively.

3.3.2. Shortcomings of the tool's CM support

With the tool as it stands, it is impossible to perform basic configuration management tasks. When the core developers fix a single bug in support of a demo, they cannot send a file containing only differences from the previous version; at a minimum, they can send either directions for applying the patch manually, or a new version of the domain(s) containing the bug fix. If the platform developers had to make other changes in any of the affected domains, their only choice is either to apply the patch manually, or to reapply their own changes manually. Similarly, when platform developers fix a bug in a core domain, they have no automated way to generate a file with just their changes, so the core developers can verify the fix. Finally, in a team development environment, it is impossible to resolve conflicts if two developers have modified the same domain at the same time. With conventional programming languages, this is a simple merge operation; often it requires human guidance to ensure that the conflicts are resolved correctly, but there are many tools to support this. None of these tools can be used here. Further, since domains are both the unit of code reuse and the smallest unit that can be managed separately, conflicts, or at least apparent conflicts, are more likely: a single domain will contain many classes and their methods.

Further, as we'll discuss in Section 4.2.1, the interactions of domains with each other and with external libraries are beyond even the primitive configuration management provided for individual domains. A domain defines the calls it will make to bridges, but the calls those bridges will make to other domains are generally visible only in the build set, which is managed in its entirety—a change in a single bridge is a change in the build set. Thus a change in the interactions between domains, or between a domain and some external library, is extremely difficult to localize and manage. It follows that most “patch releases” of the product contain the entire code base; the integration cost, as platforms reapply their own customizations, is therefore quite high.

3.4. Code Examples

In this section we'll discuss some specific issues that we've encountered in our testing of the system. These range from specific algorithms to examples of bugs that in effect are a consequence of the MDA system's architecture, rather than of any particular bit of bad coding.

<Authors' note: There will be material here in the final version of the paper subject to customer approval>

3.5. Model Compilation

In xUML tools, the model compiler is responsible for translating a model into a more useful form. Although conventionally one might expect first a transformation from a platform independent model (PIM) to a platform specific model (PSM), and only then a transformation to what we'll call a platform specific implementation (PSI), in the iUML tool the first step is manual: the platform developers add whatever domains, and make whatever other changes, they need to the model shipped from the core development team. The second step is handled by the model compiler, which generates C++ code for a specific build set.

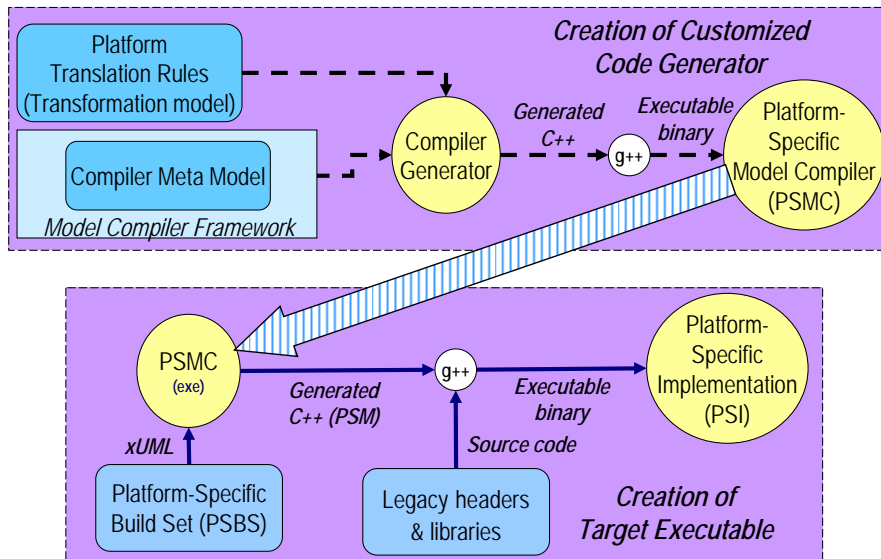


Figure 7. PIM to PSI transform through a Platform Specific Model Compiler

The model compiler generates code that will run against a set of architecture libraries that provide access to core OS services as well as to common functions such as logging—in effect, a larger version of the standard C++ libraries, which of course are an element of the architecture libraries. In addition, it is at this level that class definitions from the model are mapped into C++ classes, based on standard base classes defined in the architecture library.

The model compiler can't take an inefficient algorithm and make it better, but it can in principle produce efficient infrastructure code. Let's consider a few cases:

- The action language has been modified to include logging primitives, which compile to calls to the standard `log4cxx` [12] package. This makes it easy to add logging points that provide all the needed information, to format it in useful ways, and to filter the output as needed. The flip side is that this makes the logging mechanism part of the virtual machine; on a platform where a different logging mechanism is required, the clearest option is to change the model compiler to support it. But modifying the model compiler is to be avoided; see section 4.2.2.
- As mentioned above, the action language includes primitives that allow one to find any instance of a given class, whether or not it's referred to in any data structure. This clearly imposes some storage and performance overhead. A single naïve, but storage-efficient implementation would thread a doubly-linked list through all class instances, and use a linear search to find the matches. The architects in this case have defined a `tag2` that can be used to indicate that we expect to do a lot of finds against a particular attribute of a particular class; in

² Tags can be applied to various parts of the model, such as classes and methods; the semantics are defined within the model compiler.

that case, the compiler will generate code to store all the class instances in a Standard Template Library associative container, indexed by the value of the specific attribute. One can't declare, however, that one will *never* need to find class instances. The model compiler could, while examining a particular domain's code, observe that `find` is never used on a particular class, and give it an implementation base class that does not track all instances. This is in practice impossible, because it requires a global optimization: although classes inside a domain are not directly visible to other domains, they are visible to bridge code, which is outside the domain. Even if the compiler could do this, it would be a fragile optimization—a single ill-advised use of `find` on a particular class would eliminate the optimization for that class, in a way that would be particularly hard to find. Thus, in practice, this is something else that can only be handled by changing the model compiler, to honor a new tag that says, this class doesn't support `find`. In modern OO languages, the problem is solved by defining an interface, which a particular class can either inherit from or not.

- We mentioned other performance problems that arise directly from the use of the relational object model in xUML: there is nothing wrong with the model *per se*, but its requirements are not necessarily best handled in a conventional OO language such as C++. On the other hand, that's the language we have that can be compiled on every target machine. One could imagine optimizations based on careful analysis of the structure of class definitions. For example, a refinement class that has no sibling refinements could be implemented using normal C++ subclassing, rather than as a separate object; it might be possible to define abstract classes that sit in the C++ hierarchy above sibling refinement classes, providing a way to speed up deferred function calls by using C++ virtual method.

The list could go on, but there are three problems with solving any of these problems in the model compiler. First, the model compiler is itself written using iUML. As we discuss below, it is extremely difficult to manage different versions of the same code base; thus, we expect that only the core development team will ever change the model compiler. Should a platform go down that road, their developers will have to track all changes coming out of the core team, many of which will be critical to proper functioning of the system, in addition to managing their own changes. This is extremely difficult, as we discussed above.

Problems associated with compilation are among the most difficult to find: one does not change compiler versions lightly, nor does one generally “enhance” the functionality of a working compiler. Although the model compiler is producing C++, rather than machine language, the same principles apply: it should be relatively stable; the platforms should not require, nor should the program accept, modifications of it. The underlying principle of this development effort is that all the platforms will be running the same algorithms, but we can't be confident that that is the case, even if the platforms start with the same core models, if they're generating significantly different C++ for execution on their hardware.

Finally, it is extremely difficult to write code that reasons about other code and gets it right. I've mentioned optimizations that one might be able to make, but identifying them requires a high degree of complexity in the model compiler to find cases where these could be applied, and we've already pointed out that the language lacks the expressive power for building complex algorithms. It would be too difficult to recognize cases, for example, where an array representation might be appropriate; the alternative is to tag the model to indicate these, and modify the model compiler to honor those tags. But now we're back in the business of changing the compiler, and we're asking our engineers to think, not about the efficiency of their algorithms, but about how to make the model compiler generate code that will execute them efficiently. This second-guessing of the compiler leads to fragile, difficult-to-maintain code that ceases to be efficient when the platform changes.

4. Lessons Learned for Future xUML and MDA Tools

4.1. Use xUML only if it is appropriate

This is something that applies both to whole projects and to parts of large projects. One would not use a scripting language like Python to develop a Kalman filter, nor the domain specific language of XDG to develop anything other than the TDL logic for which it was intended. xUML is intended to be a general purpose development environment, but its quirks and limitations, as well as the ways in which it can surprise even very experienced developers, suggest that it is less generally useful than languages like C++ or Java. Transaction processing applications will often map easily into the relational model, since their underlying data store is often a relational data base; controllers will find state machines an extremely valuable facility. Other sorts of applications might find that some other approaches produce better results.

4.2. Improving xUML

Based on our experience with a tool that follows xUML as described in [2] fairly closely, we believe that the next generation of xUML tools would benefit tremendously if some adjustments were made to the xUML architecture. The OMG is currently in the process of trying to standardise a subset of UML that can be made executable (although it is not called xUML). Our opinion is that in designing new programming paradigms and languages it is beneficial to both leverage hard-won experience from previous attempts, and to avoid violating the principle of least surprise where possible. Many successful language introductions were designed as smooth transitions from existing successful languages—witness the introduction of C++ as an extension of C with support for object orientation, and the introduction of Java with a core of syntactic features that were familiar to C and C++ programmers. In this sense, as we have already indicated, the first generation xUML design has fallen short. The new version that becomes standardized by OMG will probably be different and will have its own strengths and weaknesses. In the mean time, we present some suggestions for improving the current xUML design.

4.2.1. Change the Domain Interaction Architecture

The current approach in terms of bridges and inlined code inevitably leads to strong coupling between domains, and breaks domain encapsulation because each bridge can reference an arbitrary set of domains. Consequently, code in bridges tends to rely on implementation details of each domain. This is especially harmful because it hinders domain evolution, including refactoring. When developers make changes to a domain, they must be aware of all bridge code that they might be breaking. Since domains are the smallest unit of reuse, they must be atomic. In particular, there can be no class definition re-use across domains. Classes can be used by other domains, when exposed as services by their home domain, but they cannot be re-used without breaking the atomicity of the domain concept.

Instead, each domain should provide a “domain interface” that can be used by its clients, and it must have a well-defined “required interface” that it assumes is provided for its use in the deployed environment. As long as the “required interface” is made available to a domain, the domain guarantees that it can satisfy the “domain interface”. Part of the OMG standardization effort should include selecting existing features of the UML standard that can be adapted to define domains that are fully encapsulated with domain and required interfaces that can be precisely specified. An example where OMG has done something similar is the CORBA Interface Definition Language for describing CORBA services.

Existing xUML features that break domain encapsulation, for example counterpart associations, will need to be re-designed so that they rely on domain interfaces only.

4.2.2. Modify the Role of the Model Compiler

The model compiler has a complicated role in the xUML framework. First, as shown in Figure 7, a Platform Specific Model Compiler (PSMC) is generated for each execution environment. The PSMC encapsulates the domain specific knowledge that is required to generate a PSI from a PSM. Secondly, a multitude of orthogonal modeling concerns can also be delegated to the PSMC. In particular, the adequacy of runtime performance is supposed to be guaranteed by a suitably optimizing PSMC.

In practice, it is difficult to modify a model compiler, and doing so can result in a maintenance problem. Furthermore, it is simplistic to assume that a model compiler can optimize away all modeling sins. Finally, annotating the PIM to provide hints to the model compiler in support of a variety of concerns such as multi-threading and runtime performance (which are considered not meaningful at the PIM level) only results in a cluttered and bug prone PIM.

We believe that the model compiler should be modified only when absolutely necessary, and that the PIM should not attempt to capture issues that do not properly belong there. If the recommendations of the previous section on Improving the Domain Interaction Architecture are adopted, the need for PSMCs disappears. Each domain is completely encapsulated and its method of implementation is completely hidden from its clients. Each domain by its nature will stand somewhere in a spectrum of platform specificity, and each domain will have its own way of converting to PSI. Some domains may be implemented completely in a systems programming language like C, yet they can provide their services to all domains. For such domains, the model compiler will be responsible for providing the scaffolding that implements the domain interface, while delegating platform specific details like optimization to the C code implementing the domain and an optimizing C compiler. These changes would, when combined with those from Section 4.2.1, eliminate the need for bridge code that violates domain encapsulation and increases the difficulty of configuration management.

4.2.3. Redefine Purpose of Precise Action Semantics

The value added by xUML over third generation languages is that visual representation is easier for the developers and more accessible to the subject matter experts (SMEs) who collaborate with the developers. Action Semantics needs to balance between two conflicting requirements: it must be abstract, so that it makes sense to the SMEs, but it must be expressive enough to represent the computational logic efficiently. These two goals are hard to reconcile. If complex logic can be faithfully represented in a way that results in efficient runtime performance, the language will appear complex to all except the developers. We must recognize that to be useful the Action Semantic representation can in general only capture the computational logic of a given domain at a certain (probably coarse) granularity. Since domains are fully atomic and encapsulated, in principle they could be implemented in a variety of ways. The purpose of some of these “worker” domains could be to flesh out with precise implementation details the coarser logic captured in other domains, in ways that are consistent with expected runtime quality of service. In fact, multiple such implementations should be possible, given domain encapsulation and atomicity.

4.2.4. Address the Runtime Architecture

It is clear that the RDB runtime model is too specialized to be a generic solution to an MDA runtime architecture. In addition to the performance issues, there is a fairly serious violation of the least surprise principle. The same is true of the replacement of inheritance by specialization. Considering that such important issues as runtime performance are relegated in the xUML

literature to the platform specific level (to be resolved by an optimized platform-specific model compiler), one would expect that the runtime models should at least fall in the same category. A similar criticism applies to using a threading model that is tied to the state machine design of xUML. We propose that a standard runtime model (or a reasonable abstraction of one) should be adopted. One of the lessons taken from the Java platform should be that threads and synchronization need to be addressed as first class abstractions at the domain level. If a domain is not designed to be multi-threaded, chances are fairly good that it can not run in multiple threads, no matter how smart the model compiler is. The runtime model should be designed to be compatible with a multi-threaded, event-driven architecture. The future xUML representation should be designed to eliminate memory issues. In the current version it is too easy to reference objects that have already been deleted; for example, the runtime event queue is an external data structure that can hold references to deleted objects. Java has already successfully addressed this problem, and the future xUML standard needs to be at least as effective in this area as Java is today.

4.2.5. Change the Buildset Architecture

A buildset is a set of domains that can collaborate to achieve a shared goal. Since each domain has a required interface, a buildset must be closed with respect to that relationship, i.e. each domain must be associated with a set of other domains in the buildset that together satisfy the domain's "required interface." It would be useful if a well-specified mechanism existed for wiring domains together according to their requirements. Automated tools could be built to at least verify that a buildset is consistent and that each domain's requirements are addressed. A standard buildset architecture would be beneficial because it would allow developers to reuse domains that were built using tools from different vendors. It would also allow vendors to create a market of reusable MDA tools.

4.2.6. Create an Abstraction for Distributed Processing

It should be irrelevant how a required interface is provided to a domain. A fully reconciled buildset consisting of domains completely wired with respect to their required interfaces should be deployable in a heterogeneous network centric environment. It should be equally easy to co-locate two domains in the same processor (or process) as it would be to distribute them across different processors. Although this is clearly a PSI issue, it should be addressed with an abstract specification that would permit vendors to create standards-compliant deployment tools. For instance: domain A requires domain B, domain A is implemented as a CORBA object, and domain B is deployed as a Java RMI application. A CORBA to RMI bridge could be automatically generated, consistent with the domain interfaces, that would link component A to component B.

4.2.7. Encourage Interoperability across Vendor Tools

It is important to establish standards that allow users to mix and match "best-of-breed" tools. Version control and model merging is an especially important area, where standardization may be required. If standard solutions arise in this area, the scalability of MDA efforts will be vastly improved. The MDA community needs to take a close look at the Eclipse platform and its strategy of creating an "eco-system" of tool developers and tools

5. Conclusion

In the context of ever increasing size and complexity of software projects, MDA should be considered as a possible software development strategy. MDA seems to be especially effective when it is more narrowly focused to specific domains of interest and when it is used in

conjunction with domain-specific modeling languages. We discussed two examples of such use of MDA, and many others can be found in the literature. Tools that support a generic approach to MDA and are applicable to all domains follow a tradition that started in the late eighties with CASE tools and continues today with attempts to standardize some form of “executable” UML. Standardization for such tools is in progress. The tools are not yet mature, and due to the lack of widely accepted standards, not fully interoperable. The tool vendors may have contributed to the confusion in the market place by promising much but not delivering enough, while slowing down the standardization process in order to defend narrow interests. Our case study of a large project that is being developed using the xUML methodology shows some of the promise of the xUML approach, in spite of the immature state of the tools, as well as many of its pitfalls. We believe that MDA in some form will evolve into the next generation of technology and tools for developing demanding software applications. However, this will require several years of standardization efforts as well as practical experience. While generic MDA is being defined and while it is tested in the market place, more modest efforts involving MDA and DSMLs will continue to generate success stories.

6. Acknowledgements

This work was sponsored by the Air Force Research Laboratory (Contract #F30602-01-D-0053 Order No. 0012, subcontract S06021).

7. References

- [1] Liskov, B., J. Guttag (2000). Program Development in Java – Abstraction, Specification, and Object-Oriented Design, Addison-Wesley
- [2] Mellor, S. J., M. C. Balcer (2002). *Executable UML: A Foundation for Model-driven Architecture*, Addison-Wesley
- [3] Raistrick, C., P. Francis, J. Wright (2004). C. Carter, I. Wilkie, *Model Driven Architecture with Executable UML*, Cambridge University Press.
- [4] Schmidt, Douglas C (2006). “Model-Driven Engineering,” *Computer*, February 2006, pp. 25-31.
- [5] France, Robert B., Sudipto Ghosh, Trung Dinh-Trong, and Arnor Solberg (2006). “Model-Driven Development Using UML 2.0: Promises and Pitfalls,” *Computer*, February 2006, pp. 59-66.
- [6] Balasubramanian, Krishnakumar, Aniruddha Gokhale, Gabor Karsai, Janos Sztipanovits, and Sandeep Neema (2006). “Developing Applications Using Model-Driven Design Environments,” *Computer*, February 2006, pp. 33-40.
- [7] Childs, Adam, Jesse Greenwald, Georg Jung, Matthew Hoosier, and John Hatcliff, “CALM and Cadena: Metamodeling for Component-Based Product-Line Development,” *Computer*, February 2006, pp. 42-50.
- [8] <http://www.kc.com>
- [9] Pugh, William, “Skip lists: a probabilistic alternative to balanced trees,” *Communications of the ACM*, June 1990, pp. 668-676.
- [10] Wilkie, Ian, Adrian King, Mike Clarke, Chas Weaver, Chris Raistrick and Paul Francis, *UML ASL Reference Guide*, 2003.
- [11] Some XDG reference.

[12] <http://logging.apache.org/>