

# Executable/Translatable UML and Systems Engineering

**Shayne R. Flint**

Department of Computer Science  
Australian National University  
Tel: 02-6125-8183  
Email: shayne.flint@anu.edu.au

**Clive V. Boughton**

Department of Computer Science  
Australian National University  
Tel: 02-6125-5689  
Email: clive.boughton@anu.edu.au

## Abstract

In March 2003 the *Object Management Group* (OMG) released an RFP for customisation of the *Unified Modeling Language* (UML) to support the modeling of a wide range of systems including software, hardware, people, procedures and facilities within the framework of *OMG's Model Driven Architecture* (MDA). Much of the response to this RFP has focused on how the UML notation might be extended rather than how UML and MDA could be used during the conduct of systems engineering lifecycle process activities. In this paper we describe important concepts underlying the Executable/Translatable UML and how they and more traditional modeling concepts can be used to support the objectives of MDA in the Systems Engineering context.

## 1 Background

Efforts are currently underway within the *Object Management Group* (OMG) and associated organisations to enhance the *Unified Modeling Language* (UML) and *Model Driven Architecture* (MDA) for use in systems engineering. Much of this effort has been directed towards the UML notation rather than how UML and MDA can be used in support of systems engineering lifecycle processes.

In this paper we provide an overview of UML and MDA followed by a description of the Executable/Translatable UML ( $X_T UML$ ) approach to software development and how it supports the objectives of MDA. It is then shown how concepts that underpin  $X_T UML$  might support the use of UML and MDA in the systems engineering context. We conclude with a summary of the benefits that  $X_T UML$  may bring to systems engineering, and plans for future work.

### 1.1 The Unified Modeling Language

The UML (OMG, 2003c) is a notation for specifying, visualising, and documenting models of software systems. It comprises a set of diagrams that can be used to represent models of software structure and behavior. The UML does not prescribe a method for developing software. It is only a notation for representing the models produced by many of the software development methods in use today.

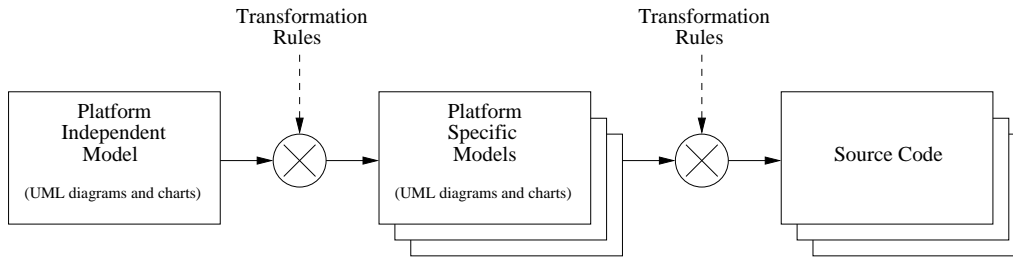


Figure 1: Primary MDA models and transformations

## 1.2 The Model Driven Architecture (MDA)

The *Model Driven Architecture* (MDA) is an *OMG* initiative that aims to improve the productivity of software development, as well as the portability and maintainability of software systems, by exploiting the well established principle of separating the specification of required software operation from its design and implementation. MDA makes use of *Platform Independent Models* (PIM), which capture software requirements completely free of any design or implementation concerns, and *Platform Specific Models* (PSM) which represent software design and implementation. PIMs are transformed into one or more PSMs according to well defined transformation rules. The PSMs produced in this way may then undergo further transformations to more specific PSMs and eventually to source code. Figure 1 depicts these primary MDA models and associated transformations.

Productivity is expected to improve by automating the transformation of PIMs to PSMs and then to final code. It is anticipated that portability will be enhanced because PIMs remain unchanged in the face of changing technology. As new platform technologies emerge they will be modeled as new PSMs and associated transformation rules. Existing, technology neutral, PIMs can then be transformed into these new PSMs leading to the possibility of large scale reuse of corporate intellectual property captured in the PIMs. Maintenance could also be improved because of the clear separation of functional requirements modeled in the PIMs and non-functional requirements modeled in the PSMs, and because PIMs may have extended lifecycles through a series of emerging technologies.

The MDA Guide (*OMG*, 2003a) and (*Frankel*, 2003) provide more complete descriptions of MDA.

## 1.3 UML, MDA and Systems Engineering

In March 2003 the *OMG* released a Request For Proposal (*OMG*, 2003b) for customisation of UML to support the modeling of a wide range of systems including software, hardware, people, procedures and facilities within the framework of *OMG's* MDA. This effort to bridge the software-hardware-people gap is being conducted by the *Systems Engineering Domain Special Interest Group* (*SEDSIG*, 2003) of the *OMG* and is supported by a several organisations including the *International Council on Systems Engineering* (*INCOSE*, 2003).

While a number of organisations are working on the UML specification itself, few are working on how the UML and MDA should be used to support the disciplined conduct of systems engineering lifecycle process activities. As a result, there is a risk of repeating events in the software industry which have led to the

widely and falsely held view that UML is much more than just a notation and that its use alone somehow constitutes a disciplined approach to software specification and design.

In order to maximise the benefits of using UML and MDA in the systems engineering context, we believe that methods for using the technology need to be developed along side development of the UML and MDA standards. The Executable/Translatable UML ( $\frac{X}{T}UML$ ), described in the next section, is a well established method for the development of software and while it facilitates an MDA approach to software engineering, it is based on principles more advanced than those underpinning MDA. It is these more advanced principles that may support the practical application of MDA in a systems engineering context.

## 2 Executable and Translatable UML: Key Concepts

In the following subsections we describe key concepts that set  $\frac{X}{T}UML$  apart from other approaches to implementing MDA. In particular,  $\frac{X}{T}UML$  is a translative method of software development which supports the separation of concerns beyond those represented by MDA (separation of PIMs from PSMs). We also show how  $\frac{X}{T}UML$  provides a framework for the systematic development and verification of requirements specifications and design descriptions, and their subsequent translation into deployable software.

### 2.1 $\frac{X}{T}UML$ is a translative method of software development

Object oriented software development methods can generally be classified as either *elaborative* or *translative*.

#### 2.1.1 Elaborative methods

*Elaborative* methods of software development are most popular and are based on the belief that object orientation can be used to smooth the transition from analysis to design and implementation. An *elaborative* approach begins during analysis by creating object oriented models of software at a high level of abstraction, omitting design and implementation details. During design phases these models are *elaborated* (or refined) to include design and implementation information thus blurring the distinction between requirements specification and design descriptions. Eventually the models become specifications for code.

Elaborative techniques, by definition, do not support complete translation limiting the capabilities and promise of MDA. Most of the object oriented methods and tools available support elaboration but not translation.

#### 2.1.2 Translative methods

Translative methods of software development are based on the belief that maintaining a clear separation of concerns throughout the entire software lifecycle improves the understandability, verifiability, portability, large scale reuse and productivity associated with software engineering artifacts. These separated concerns

include aspects associated with requirements specifications, software architecture, and implementation.

Models produced during analysis activities specify the required data, state and behavior of separate aspects of a software system independent of implementation. Information from these models is then *translated* and *woven* together with details from separately developed software architecture and implementation models to form code and other software engineering artifacts. These concepts of separating concerns and weaving constitute an important generalisation of the more specific MDA concept of developing a PIM and transforming it into one or more PSMs.

The translative approach to software development was pioneered by Sally Shlaer and Stephen Mellor in the 80's leading to the development of the *Shlaer/Mellor method* (Shlaer and Mellor, 1992). During recent years and with the adoption of some translative ideas by the *OMG* in MDA, the Shlaer/Mellor method has been refined and updated to make use of a well defined subset of UML for representing models which can be executed for verification and simulation purposes. This method is now called Executable/Translatable UML ( $X_T UML$ ) and is described in a number of recent books (Mellor and Balcer, 2002; Starr, 2002; Starr, 2001). The method is also supported by at least two commercial software engineering tools (Project Technology, 2003; Kennedy Carter, 2003).

## 2.2 Domains

$X_T UML$  extends the simple concept of PIMs and PSMs by supporting the separate modeling of the various subject matters that comprise a software system. An  $X_T UML$  model comprises a set of *domain* models that each capture the concepts of an autonomous subject or aspect of a system such as the application itself, user and other interfaces, databases, security and networking. While some domains may appear to be subsystems, most are not. Most domains specify a particular aspect of a system which may touch all parts of the final software. An example of such a domain is security. A security domain may specify security related concepts such as users, roles, passwords, and privileges, how the concepts relate to each other, and how they respond to various stimuli. Security may affect many parts of an operational software system but nonetheless is modeled and verified independently of any other subject matter. During model translation, subject matter of the application domain, security domain and other domains will be systematically *translated* and *woven* together to form a working software system.

Domains can be categorised as *application*, *intermediate abstractions*, or *implementation*. Application domains deal with software requirements and contain no design or implementation concerns. Implementation domains, on the other hand, deal with design and technology concerns such as software architecture, programming languages and communications. A working software system will be constructed from the concepts defined in implementation domains. Intermediate abstractions are domains used to decouple an application requiring generic services, such as a database, from the specific technology that provides the service. Some of these domains may be well understood or they may already exist. These *realised* domains are not modelled using  $X_T UML$  and usually represent implementation technologies and external systems with which the subject software must interact, including off-the-shelf or legacy software.

Figure 2 depicts a domain chart showing the software aspects of a simple sys-

tem. The implementation domains represent various aspects of the popular open source LAMP architecture - Linux (Operating system), Apache (web server), PHP (programming language) and MySQL (database). The Persistence Domain is an intermediate abstraction that decouples application domains from implementation technology (MySQL). The Warehouse and Security domains model application requirements.

### 2.3 Executable domain models

An important feature of  $\frac{X}{T}UML$  is that domain models are complete executable models of a given subject matter. This means that domains dealing with requirements are executable specifications which allow for verification of required functionality early in the software lifecycle before any design or implementation technologies are even considered. This emphasis on a set of independently executable domain models is a cornerstone of the  $\frac{X}{T}UML$  approach and sets it well apart from the other approaches to MDA.

To facilitate the construction of executable models,  $\frac{X}{T}UML$  domains are represented using a well defined and integrated subset of UML comprising class diagrams (no operations), state charts, collaboration diagrams and sequence diagrams together with well defined semantics. Such a subset is necessary because UML, as it is defined in the current UML specification (OMG, 2003c), is unnecessarily large and devoid of the precisely defined semantics necessary to produce executable models.

Note that while  $\frac{X}{T}UML$  domain models are represented using a small subset of UML and include an asynchronous view of behavior, they place no restrictions on how domain subject matter is translated. For example, domain models can be translated into object-oriented or structured designs, or directly into object oriented, structured or unstructured programming languages. An asynchronous specification can be implemented as a synchronous design.

### 2.4 Bridges

Application domains are usually modeled independently of issues such as security, user interaction and persistence. They rely on *bridges* to other domains to deal with such issues. These bridges are shown as dashed arrows on the domain chart and represent the assumptions made by a client domain and a set of corresponding requirements that are placed on some other server domain.

For example, the arrow between the Warehouse and Graphical User Interface (GUI) domains depicted in Figure 2 indicates that the Warehouse domain assumes that some other (anonymous) domain will provide a user interface and that in this case the bridge has placed corresponding requirements on the GUI domain.

The consequences of this bridging concept include the ability to replace or supplement server domains without changing any client domains. For example, the GUI domain depicted in Figure 2 could be replaced, at the level of specification rather than design or implementation, with a domain that models another form of user interface such as could be provided by a mobile phone. Replacing a domain in this way would require changes to bridges from client domains, but not the domains themselves.

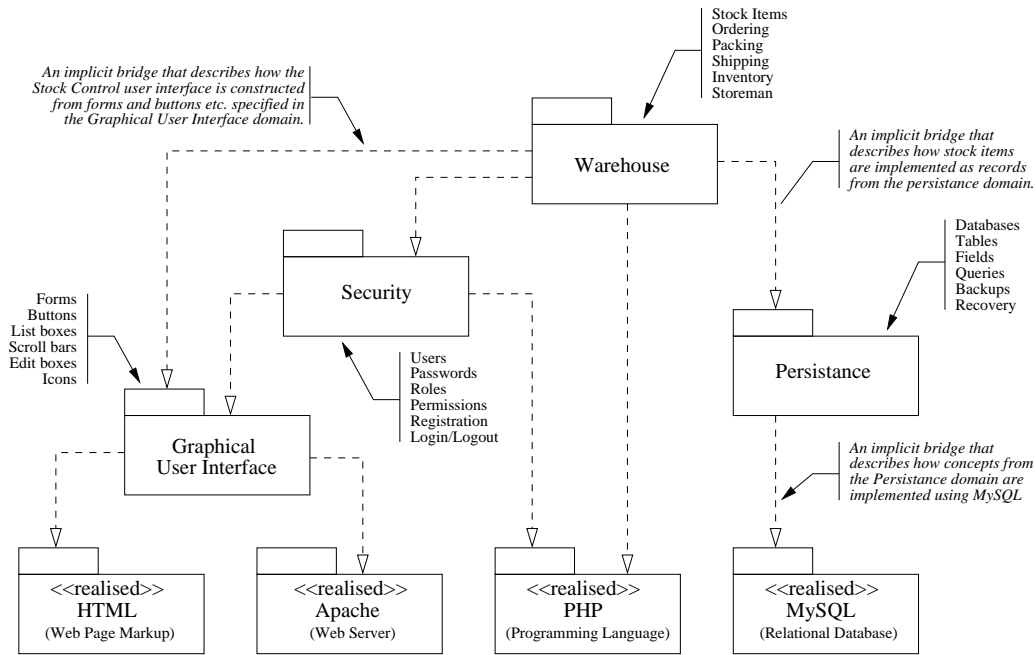


Figure 2: An Annotated Software Domain Chart

### 2.4.1 Explicit Bridges

*Explicit Bridges* represent assumptions within the model of one domain concerning the existence of another domain that are explicitly represented as signals to and from external entities and the invocation of operations on such entities. In effect, the model of one domain assumes that some other anonymous domain, linked via an explicit bridge, will generate required signals, consume and act appropriately on signals it is sent and implement correctly any operations invoked on it.

Like domain models, the implementation of links between domains is not constrained by modeling them in terms of signals and operations. Signals, for example, could be implemented as asynchronous events or method calls within a single processor, by passing messages between computers connected via the internet or by any other means that satisfy requirements placed on the domains involved. The choice of implementation approach will often depend on design and architectural constraints.

Explicit bridges are usually used to link application and intermediate abstraction domains to realised domains that represent external systems and legacy software.

### 2.4.2 Implicit Bridges

By far the most common and important bridges in an  $X_T UML$  model are *implicit bridges*. Unfortunately they are also the most difficult to understand for those new to  $X_T UML$ .

In contrast to explicit bridges, *implicit bridges* represent assumptions within the model of one domain concerning the existence of another domain as a set of rules that direct the use of subject matter in the bridged domains to form requirements specifications, design descriptions, implementation and other software engineer-

ing artefacts.

Implicit bridging rules can be categorised as *mapping*, *transformation*, or *weaving*. *Mapping* rules are used to state the correspondence between the subject matter of one domain and that of another domain. For example, stock items in the Warehouse domain depicted in Figure 2 may correspond to text items in a scrollable list box in the Graphical User Interface (GUI) domain. In such a case, the implicit bridge may model a complete user interface for warehouse operations based on concepts specified in the GUI domain. These models may take the form of screen shots or some other diagrammatic representation of the required user interface. The implication of this is that the GUI domain, which only defines the concepts of a GUI and how the concepts relate and behave, is reusable in other applications that require a GUI. It is the implicit bridge to the GUI domain that specifies how the concepts will be used to satisfy the needs of the Warehouse domain.

*Transformation* rules involve the *transforming* of subject matter of one domain into subject matter of another domain. For example, stock items in the Warehouse domain may be transformed into records in a relational database table specified in the Persistence domain. Other data from the Warehouse domain could be implemented in other database tables. Relationships between warehouse data would be implemented as foreign keys in the appropriate relational database tables. Note that we have described two different translations of stock items; one as a *mapping* to items in a GUI scrollable list box and another as a *transformation* into records of a database table. In effect the two implicit bridges require the concept of a stock item to end up as a record in a database table, and that information from this database table record be displayed in a GUI list box.

*Weaving* rules are more complex and involve the principles of aspect orientation (Elrad et al., 2002) where the subject matter of one domain is woven throughout the subject matter of another domain. For example, concepts such as registered users, access rights, roles, passwords and encryption specified in the Security domain depicted in Figure 2 may be woven throughout the Warehouse domain. The implicit bridge between the two domains is used to specify the actual access rights and roles required by the warehouse and how such details impact its operation.

### 2.4.3 Modeling Implicit Bridges

There are no concrete rules in  $X_T UML$  regarding the modeling of implicit bridge rules. In fact, it is often assumed that the operation of implicit bridges is embedded within automated translation tools such as those included with commercial  $X_T UML$  modeling environments (Project Technology, 2003; Kennedy Carter, 2003). However, in practice, automated translation tools are limited to a small range of target platforms and are not particularly good at dealing with implicit bridging rules other than the transformational kind. It is therefore necessary to consider approaches to modeling implicit bridges so that translations can be accurately performed by software engineers.

As a general rule, it has been found that implicit bridges should be modeled using approaches appropriate to the nature of the bridge involved. For example, bridges from application domains to GUI domains might be modeled using pictures of the required user interface screens. Bridges from an application domain to a database domain may use a table to describe the mapping of data types in an application domain to field types supported by an SQL database. Bridges from

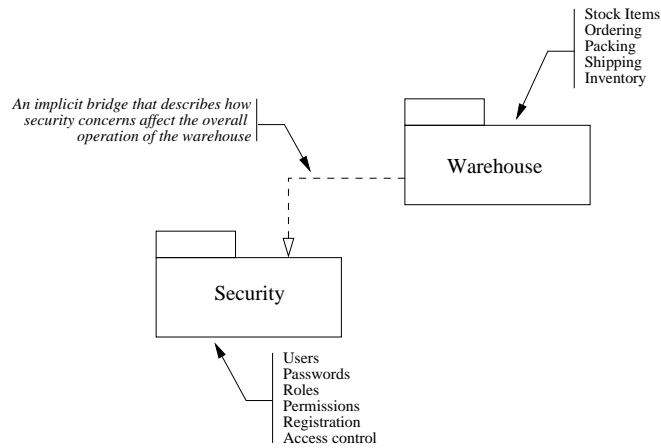


Figure 3: An Annotated Application Domain Chart

an application domain to a programming language domain are likely to make use of design patterns (Gamma et al., 1995) to describe the translation of application subject matter into source code.

## 2.5 $\frac{X}{T}UML$ and the software development lifecycle

$\frac{X}{T}UML$  domains are identified, modeled and used throughout the software development lifecycle. During analysis, application domains, applicable intermediate abstraction domains and associated bridges are identified and modeled. During design, any other required intermediate abstraction domains are modeled along with all chosen implementation domains and associated bridges. During implementation, domains are translated into deployable software in accordance with rules associated with implicit bridges connecting the domains.

During maintenance, domains and bridges may be added, removed and/or modified to correct defects, add new functionality and to take advantage of new technology.

Implicit bridges should be created and maintained separately from models of the domains involved. That is, domain models capture intellectual property, while the creation and maintenance of bridges puts intellectual property to work.

## 3 $\frac{X}{T}UML$ and Systems Engineering

While  $\frac{X}{T}UML$  is primarily used to develop software systems, the concepts it employs may have wider applicability in areas such as systems engineering and the development of intelligent enterprises (IEWG, 2003). So, how can  $\frac{X}{T}UML$  concepts be used to model systems comprising hardware, software and people? We believe the answer may lie in the systematic development of autonomous domain models and the use of implicit bridging between them.

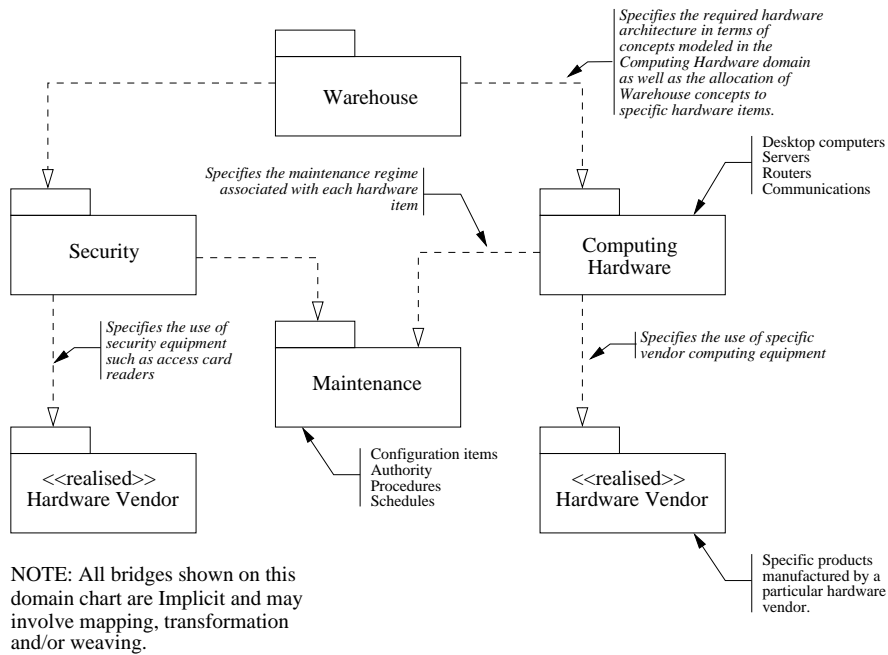


Figure 4: An Annotated Hardware Domain Chart

### 3.1 Systems Engineering Domain Models

The purpose of  $X_T UML$  domain modeling is to separate specific autonomous subject matter. While this approach has been mainly used in the software development context, we believe it may have applicability within the wider context of systems engineering. For example, domains associated with hardware, maintenance, process and training and even project management, safety, risk management, test and evaluation could be used.

Because of the large number of domains likely to be involved in a systems engineering effort, we have found that a series of domain chart views, each showing those domains associated with a particular viewpoint, leads to improved clarity and scalability over the single domain chart used by the  $X_T UML$  method. A number of such domain views are depicted in Figures 2, 3, 4 and 5. Annotations on these figures indicate the nature of each domain and the implicit bridges between them.

The kind of models developed for each domain need to be considered carefully. In order to ensure the executability of domain models, the  $X_T UML$  subset of UML could be used. In some cases, however, UML may not be appropriate. For example domains dealing with mechanical and electronic design should be modeled using traditional approaches. In fact, software may sometimes be more effectively modeled using techniques such as Data Flow Diagrams, Entity Relationship Diagrams and Process models.

Obviously, the basic concepts underpinning  $X_T UML$  and MDA do not require the use of UML, and so the universal use of UML as proposed by the *OMG* may not be required to implement an MDA approach to systems engineering or indeed software engineering.

The key to understanding how this might work lies in the systematic use of implicit bridges between domains.

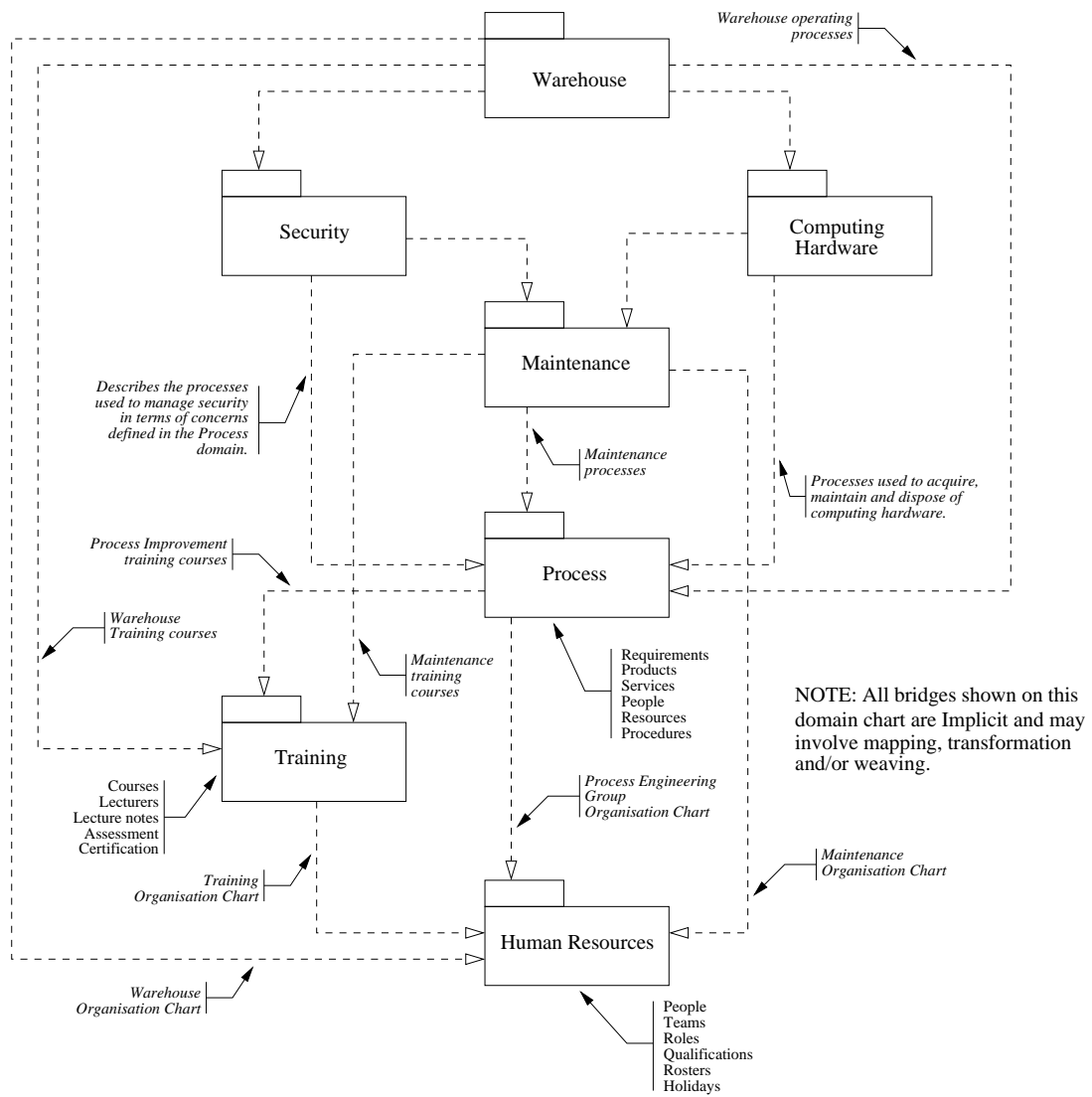


Figure 5: An Annotated Process, Training and Human Resources Domain Chart

## 3.2 Bridges between system domains

Bridges between domains in a systems engineering context serve the same purpose as those in a software only context. That is, they comprise assumptions made by a client domain and a set of corresponding requirements that are placed on some other server domain.

In the software context *explicit* bridges are specified in terms of signals and operations which can be implemented in a variety of ways. In the systems engineering context, some explicit bridges will need to be specified in new ways including mechanical interfaces and procedures that can be followed by humans.

Similarly, implicit bridges may need to be specified in new ways including those depicted in Figures 4 and 5. For example, the bridge from the Warehouse domain to the Computing Hardware domain depicted in Figure 4 might use a network model to describe the generic equipment and communications required to host software produced to support warehouse operations. The bridge from Computing Hardware to the Hardware Vendors domain might use a table to identify specific pieces of vendor supplied equipment required to populate the above network of equipment and communications. The bridge from Maintenance to Human Resources depicted in Figure 5 might use organisation charts to describe the organisation of people, teams and roles required to maintain equipment described by the above mentioned bridges. The bridge from Maintenance to Process might use work flow diagrams and the like to describe maintenance processes.

## 4 The benefits of $\frac{X}{T}UML$

The real value of the  $\frac{X}{T}UML$  approach is not so much its support for MDA, but rather it's more general applicability in supporting the integration of disciplines practised in systems engineering, the systematic creation and profitable exploitation of intellectual property as well as the simplified adoption of new technology.

### 4.1 Integration of disciplines

The modeling of autonomous subject matter in separate domains, together with the use of implicit bridges between them, supports the conduct of all systems engineering lifecycle activities within a single framework. Such a framework facilitates the independent work of domain experts and supports the systematic weaving of their output to construct, maintain and operate a system or indeed a large system of systems. This approach directly supports the *OMG's Systems Engineering Domain Special Interest Group* (SEDSIG, 2003) objectives to bridge the gap between software and hardware engineers as well as integration of the work products of soft sciences such as human resource management, psychology and sociology.

### 4.2 Intellectual Property Management

$\frac{X}{T}UML$  provides direct support for the disciplined and systematic creation, maintenance and exploitation of intellectual property. Domain modeling can be viewed as the creation, verification and maintenance of intellectual property related to a particular subject matter. Domain models will often represent significant corporate

knowledge and investment that is independent of changing technology. Intellectual property created in this way can be exploited by assembling a set of domains and associated bridges to form a complete system model which can be transformed into an operational system.

The intellectual property captured in domain models, can be reused to specify and form other systems. For example, a company could develop domain models that deal with the subjects of military situation display, tracking, mission planning, messaging, security and graphical user interfaces. These separately developed domain models could then be bridged implicitly in various configurations and with various implementation domain models (platforms) to form a range (product line) of military command and control systems.

### 4.3 The adoption of new technology

The concept of domains and implicit bridging simplifies the adoption of new technology in existing and new projects. As new technologies emerge, the value of intellectual property captured in many existing domain models is maintained. To use new technology, these existing domains are simply bridged to domains that model such technology.

For example, the *realised* domains depicted in Figure 2 could be replaced with a completely different software implementation without any need to change the application and intermediate abstraction domains. The MySQL database could be replaced with PostgreSQL and the PHP domain could be replaced with Java without affecting the content of other domains. Similarly, the Computer and Vendor Hardware domains depicted in Figure 4 could be replaced with new technology without needing to change the warehouse domain.

Of course, any change in technology is likely to have an impact on requirements in that more or less may be achievable with new technology. In such cases, application domain models may need to be changed to reflect changing requirements, but would still remain independent of any technology, new or otherwise.

## 5 Further Work

### 5.1 Translative approaches to enterprise architecture

The  $X_T UML$  principles described in this paper and the way in which they could be applied to systems engineering, may also be applicable to the design and maintenance of enterprise architectures. This will be the subject of further work by the authors.

### 5.2 Capability Dynamics

(Flint, 2001) describes research which has led to the development of *Capability Dynamics*, a novel approach to modeling the dynamics of requirements and capabilities developed and used throughout complex systems and organisations. This ongoing research is now exploring the use of domains and implicit bridging to systematically translate *Capability Dynamics* models into operational systems and

organisations that easily adapt to changing environments, objectives and measures of effectiveness.

## 6 Conclusions

In this paper we have outlined moves by the *OMG* to extend UML and MDA for use in the systems engineering context. We noted that this work is mainly concerned with development of the UML and MDA standards rather than how these standards could be applied to systems engineering. Concepts that underpin  $X_T UML$  were described and considered in the context of systems engineering. We concluded that the  $X_T UML$  concepts of domain modeling and implicit bridging could support an MDA approach to systems engineering without further development of the UML and MDA specifications.

## References

- Elrad, T., Filman, R., and Bader, A. (2002). Aspect-oriented programming: introduction. *Communications of the ACM*, 44(10):29–32.
- Flint, S. (2001). Capability dynamics - an approach to capability planning in large organisations. In *Proceedings of the 2001 International Council on Systems Engineering Symposium*, Melbourne Australia.
- Frankel, D. (2003). *Model Driven Architecture, Applying MDA to Enterprise Computing*. Wiley Publishing, Indianapolis, IN.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.
- IEWG (2003). *INCOSE Intelligent Enterprise Working Group*. International Council on Systems Engineering [<http://www.incose.org/iewg/>], Seattle, WA.
- INCOSE (2003). *International Council on Systems Engineering*. INCOSE web site [<http://www.incose.org/>], Seattle, WA.
- Kennedy Carter (2003). *iUML executable UML modeling tool*. Kennedy Carter Ltd. [<http://www.kc.com/>], Surrey, UK.
- Mellor, S. and Balcer, M. (2002). *Executable UML, A foundation for Model-Driven Architecture*. Addison-Wesley, Indianapolis, IN.
- OMG (2003a). *Model Driven Architecture Guide, version 1.0*. Object Management Group [<http://www.omg.org>], Needham, MA.
- OMG (2003b). *UML for Systems Engineering, Request for Proposal*. Object Management Group [<http://www.omg.org>], Needham, MA.
- OMG (2003c). *Unified Modeling Language Specification, version 1.5*. Object Management Group [<http://www.omg.org>], Needham, MA.
- Project Technology (2003). *BridgePoint Development Suite*. Project Technology Inc. [<http://www.projtech.com/>], Tucson, AZ.
- SEDSIG (2003). *OMG Systems Engineering Domain Special Interest Group*. Object Management Group [<http://syseng.omg.org/>], Needham, MA.

- Shlaer, S. and Mellor, S. (1992). *Object Lifecycles, modeling the world in states*. Prentice-Hall, Upper Saddle River, NJ.
- Starr, L. (2001). *Executable UML, A case study*. Model Integration, LLC. [<http://www.modelint.com>], San Francisco, CA.
- Starr, L. (2002). *Executable UML, How to build class models*. Prentice-Hall, Upper Saddle River, NJ.