How To Apply MDA To Simulation

Shawn Parr, Chief Technical Officer; Dr Russell Keith-Magee, Software Engineer

Calytrix Technologies Pty Ltd
info@calytrix.com

Abstract. Over the last few years the Open Management Group (OMG) has been consolidating various open standards, such as UML, XMI and MOF, into a unified approach to systems development called the Model Driven Architecture (MDA). MDA delivers a codified approach for designing and building component-based systems that remain decoupled from the languages, platforms and middleware environments that will be eventually used to implement the system. Central to the MDA philosophy is the concept that an organization should be able to model and build their systems once, and then transition these systems over time as standards and infrastructure technologies evolve and change.

While MDA has been for the most part focused on the enterprise development space, the advantages that MDA delivers are equally applicable to the simulation community. This paper will examine both the technical and business motivations for adopting the Model Driven Architecture in the simulation community. As part of this discussion it will be shown that MDA, and its related industry standards, can deliver significant improvements and project savings in the areas of simulation design, tools support, automated code generation, and reuse. In addition, a case study will be presented that demonstrates the application of MDA in simulation.

1. INTRODUCTION

At present, simulation development tends to be performed in a substantially ad hoc manner, without any particular architectural concern for medium to long-term maintainability and reuse. While the general software engineering community has made many advances in this area, the simulation development community has been slow to adopt these improvements. For example, the Unified Modelling Language (UML), which is common in other software domains, has not been widely adopted by the simulation community. Similarly, the movement towards component models (such as Enterprise Java Beans (EJB) and the CORBA Component Model (CCM)) in the enterprise-computing space has not been observed in the development methodologies used in simulation.

One of the more recent developments in software engineering has been the development of the Model Driven Architecture (MDA) from the Open Management Group (OMG). The purpose of the MDA is to ensure, at an architectural level, that behavior is abstracted from infrastructure and platform concerns, allowing behaviors to be reused and transitioned as infrastructure standards change. The MDA development approach, supported by tools that can automatically generate the required infrastructure code, has the potential to deliver higher quality code, with reduced testing requirements, and improved reusability. By contrast, minor interoperability issues currently plague the simulation community, including issues between High Level Architecture (HLA) standards (e.g., HLA 1.3, IEEE 1516) and between Run-Time Infrastructure (RTI) implementations (DMSO, Pitch, MäK). Furthermore, development efforts require extensive debugging and the reuse of simulation behavior between different infrastructures (e.g., reusing logic from a Distributed Interactive Simulation (DIS)) project in an HLA simulation) is still difficult to realize.

This paper will introduce the Model Driven Architecture, and demonstrate the advantages of adopting this approach for simulation development. Section 2 of this paper presents an overview of the MDA, including the motivations for its adoption and its current state of maturity. Section 3 then describes the development process for building an MDA based simulation. In Section 4, the benefits of adopting an MDA approach to simulation development will be discussed. These benefits include a reduction in the required coding effort, increased code quality, improved documentation and reduced exposure to changing standards. Finally, Section 5 will demonstrate these benefits using a simple case study drawn from DMSO’s own “HelloWorld” HLA example.

2. AN OVERVIEW OF MDA

The OMG’s MDA provides an approach for designing and building component-based systems that remain decoupled from the languages, platforms and middleware environments that are eventually used to implement the system. Central to the MDA philosophy is the concept that an organization should be able to model their systems once and then transition them over time as standards and infrastructure technologies evolve and change.
Figure 1 shows the OMG’s vision of the Model Driven Architecture (OMG, 2001a). As this figure shows, the MDA has a very broad scope. As such, it is not a single standard or even a single concept, but rather an amalgam of various industry domains, open standards and modeling techniques (UML, Meta-Object Facility (MOF), XML Metadata Interchange (XMI) and Common Warehouse Model (CWM)) that have been consolidated over the last few years to support a common direction and methodology. Nor would it be accurate to describe the MDA as a revolution, but rather it is evolution that encompasses existing key technologies in order to address a number of pressing issues in the software industry, including interoperability, reuse, design and change management.

At the center of the MDA approach is the model. As the acronym implies, MDA development is driven from and maintained through a platform independent model. While design models are already used extensively in the initial design phases of projects (often utilizing UML notation), in a pre-MDA environment these are usually treated as a starting point for the final implementation. In contrast, the MDA advocates that the model is the system, and that all changes must therefore be performed on the model.

In order to understand the emphasis on modeling, this section will examine the motivations behind adopting the MDA, followed by an overview of the design process and its underlying technologies. This will be followed by a discussion on how MDA can be applied to simulation.

2.1 Motivation for Adopting The MDA

To appreciate the strong focus placed by the MDA on modeling and business process, it is important to understand the two motivations behind this initiative.

2.2 Motivation 1 - Managing Change:

One of the continual challenges facing the IT industry is the pace and frequency of change, whether that be evolutionary changes in standards or major infrastructure upgrades, such as those imposed by new platform and infrastructure technologies like J2EE, CCM and Web Services. Figure 2 shows the adoption of platform technologies over the last decade. From this figure we can make three observations. Firstly, platform technologies tend to evolve and change. Experience would suggest that this occurs over a period of three to five years and it would be reasonable to assume that the technologies shown in this figure will continue to evolve. Secondly, as a successful technology becomes mainstream, the directly competing technologies disappear. This presents a real challenge; while on the one hand there is a need to stay current in order to maintain competitive advantage, there is also a risk of adopting a dead-end or vendor specific technology that does not allow a system to move forward. Thirdly, as a technology becomes ubiquitous it will often form part of the core infrastructure used by the next generation of products, reaffirming the need to stay technologically current.

This constant change presents a real problem to an organization, which needs to both use and integrate their existing software while employing industry standards to provide future proofing for tomorrow’s next best thing.
Unfortunately, while developers appreciate that change will inevitably occur (and that this is a good thing), current development methodologies and tool sets do not adequately support this kind of change. In part, this failing can be attributed to the way current design and development tools tend to be targeted at a single platform. If that platform changes or needs to be replaced then it is usually a manual exercise to repurpose the software. For example, in an enterprise scenario this might mean migrating a system from CORBA 2.3 to CORBA 3.0 (and the CORBA Component Model) (OMG, 2001b) or from a Java platform moving from RMI to a J2EE container based model. A similar situation exists in the simulation arena, with groups having to support DIS simulations (using various PDU standards), the OMG’s HLA 1.3 specification and the IEEE 1516 HLA standard, with new platform initiatives like the eXtensible Modeling and Simulation Framework (XMSF) (XSMF, 2002) and the Test and Training ENabling Architecture looming.

2.3 Motivation 2 - Promoting the Business Model:
Software systems tend to serve an organization best when they focus on the business needs first and the technology second (Siegel, 2001). However, as we have already seen, our current development environments tend to be very platform specific; so while a system design may start at a business level it will invariably end up as a programming exercise. At the end of the development process, only a loose coupling between the initial business constraints and the final platform-oriented design remains, making traceability between the now disconnected designs difficult. In practice this makes it difficult to see how a business change will impact the physical system. Conversely, because the business model has been altered to fit the target platform, it is difficult to transition or reuse the business aspects of the design with a new platform. Under these conditions it is the technical needs and not the business concerns that drive the development of systems.

2.4 Addressing these Problems
The MDA directly addresses these problems by codifying and standardizing the steps required to take a model from design through to final implementation. Using the MDA approach, a system is defined by a platform independent model (PIM) that captures all the business and non-business (e.g., security, performance) attributes of a system. The PIM becomes the core development artifact of the system. From the PIM, the MDA defines a process of refinement and transition that allows the model to be realized through a platform specific model (PSM) for a given platform. However, unlike current modeling environments that separate the design phase from the subsequent development and implementation tasks, the MDA process clearly defines how the PIM is realized via a PSM for any given platform without polluting or compromising the core design integrity with implementation issues.

This approach has two effects. Firstly, the design remains the central artifact of the system, ensuring that the business model captured in the PIM always reflects the actual system. Secondly, by not tying the design to any given platform, the PIM and its associated business logic can be transitioned over time as standards evolve and when the next big thing comes along.

The following section will describe in more detail the development process using the MDA.

2.5 Technical Overview
The first step when developing an MDA model is to describe the system using a PIM. The PIM provides a complete description of the system’s business and non-business attributes, without reference to any implementation or technical concerns. The PIM is captured using UML and associated metadata. Industry specific models are emerging in the form of UML Profiles, which provide a mechanism to capture additional business-oriented information in the model.

While the ultimate goal of the MDA is to allow a system to move directly from the PIM specification to the completed system, in reality this ambition is still some way off. As a result there are a number of intermediate steps required, through the definition of platform specific models (PSM), to refine the PIM for the targeted platform. As the design focus moves from the PIM to PSM, there is a shift in emphasis from the business aspects at one end of the spectrum to the technical aspects at the other end. The PSM captures additional metadata, in the form of UML diagrams and transformation rules, that describe how the services requested at the PIM level can be realized in the targeted environment. A PIM may have multiple PSMs for multiple platforms.

The MDA also describes a number of pervasive services, such as security, transactions, and naming, that should be supported by all target platforms. These services are provided though the platform’s container or component model. It is therefore the PSM’s role to describe how a requested service at the PIM level is to be realized in the targeted platform’s component model. It should be noted that if a platform does not support, or provides only partial support for a required service, then it may not be possible to meet the complete requirements of the PIM.

Figure 3 shows the relationship between the PIM and PSM and how some of the key technology standards, such as the Meta Object Facilities (MOF) and XML Metadata Interchange (XMI), are utilized by the MDA.
As this figure demonstrates, a continual refinement process occurs at both the PIM and PSM levels, which are in turn mapped or refactored back to the PSM or PIM respectively to ensure that both models remain in step with each other.

Once the PIM and PSM have been completed, automated code-generation tools can be used to create the system’s components. The PSM should contain the same level of detail as the coded application. This detail is stored in UML, using Object Constraint Languages (OCL) and Action Semantic Languages (ASL) to more accurately specify the internal behavior of a component. However, while it is the ambition of the MDA to completely model and automate the generation practice it is still some way to being fully realized. Currently, the most advanced MDA tools model only the behavioral aspects of a system for a given industry (arguably leaving the more complex considerations to the programmer); other tools take a more component-oriented approach and model just the container services (OMG, 2001c). Regardless of the degree of automated code generation, the use of MDA based tools has the potential to deliver a significant saving in development time by allowing the developer to concentrate on business logic concerns rather than integration logic. This ultimately leads to better quality code with reduced development risk. The entire MDA development process is shown in Figure 4.

2.6 Current State of MDA
While many of the concepts being advocated are similar to those of CASE tools from the early to mid 1980s, the software industry as a whole has matured. As a result, unlike CASE tools that were proprietary in nature, the MDA is based on standard architectures, employing well-understood open standards like UML, MOF and XML. Beyond the technical issues, the MDA is generating strong support from the business sector, which recognizes the need for improved software process and tools in order to improve their return on software investment while managing technological change.

While the MDA is still a relatively new entrant with limited tools support, the future for this approach to software modeling and development looks promising. Not only is the MDA now the key technology focus from the Open Management Group (OMG), arguably the world’s most influential standards body, but it has also received strong industry support from internationally recognized companies such as IBM Software, IONA Technologies, Lockheed Martin, Rational Software and Thales Research and Technology.

In the following sections, we will explore how the use of MDA affects the simulation development process, and the development benefits that come as a result of applying the MDA approach.

3.  EFFECT OF MDA ON THE SIMULATION DEVELOPMENT PROCESS
The previous section provided an overview of the MDA and described how by focusing on the model as the central artifact of the system it is possible to create platform independent models that can be transitioned and reused on different target platforms. Building on this foundation, the following section will focus on how the MDA design philosophy can be applied to the simulation domain.

3.1 Simulation Design
Regardless of the target domain, the design process for an MDA application involves developing a PIM. In the simulation domain, this PIM design process consists of describing the data model for the simulation (all the
entities and interactions that are allowed to participate in the simulation, the simulation components (i.e., the executable units) that utilize this data model, and the interaction between the simulation components and the data model. The definition of the PIM also requires identification of the pervasive services required for the simulation. This could include such services as ownership management, timing, data distribution management (DDM), and persistence.

The PIM design is described using UML. Simulation specific features are captured using a notation defined by a UML Profile for simulations. Additional metadata, such as performance requirements, may also be associated with UML elements. As an optional feature of the design, the PIM might include definitions of simulation logic, using UML notations such as state diagrams and action semantic diagrams. However, these logic elements are not absolutely required; the next section will describe how it is possible to integrate hand written behaviour code into the MDA process.

It should be noted that the descriptions provided during this design phase are independent of the final choice of infrastructure. For example, when transitioned to a DIS environment the entities and interactions in the design will map to PDUs, which the automatically generated component code captures and marshals for the user. Alternatively, when converted into a HLA simulation, entities and interactions in the design will map to FOM object classes and interactions, simulation components map to federates, and pervasive services map to specific API calls on the RTI. However, the PIM design process should not be performed in terms of a specific infrastructure. The mapping to DIS, HLA or any other infrastructure is performed as part of the process of defining a PSM.

3.2 Automated Code Generation

Given a UML based representation for the PIM, it is possible to automatically generate code for the simulation. The developer selects a target PSM; the definition of this PSM includes transformation rules for converting the abstract model of the PIM into a platform specific model, plus a set of templates for generating code from this PSM. A code generation engine then takes these templates, and drawing from the PIM, generates code for the simulation. This code could be in any language, targeted at any hardware platform, and for any simulation infrastructure that provides the required pervasive services. A PIM can be rapidly re-tasked for an alternate infrastructure by simply regenerating the code using a different PSM definition.

It should be noted that the choice of pervasive services in the PIM may constrain the choice of PSM; if a given infrastructure does not provide a given service, it may not be possible to generate code for the infrastructure. For example, a PIM that defines the need for a timing service may not be able to be deployed onto a DIS PSM due to the restricted timing controls available on that platform. A user defined timing service within the DIS environment could be used to overcome this problem.

In the ideal MDA process, all code is generated from the PIM, including behavior logic. Behavior code is generated from the definitions of state diagrams, action semantic diagrams, and other logic defining diagrams in the PIM. Behavior can then be described through the use of Action Semantic Languages (ASL) – an abstract programming language that can be transitioned to a specific language (such as Java or C++) through the code generation process. The exact nature of the ASL is the subject of an ongoing standardization process within the OMG.

However, this vision of MDA is not the only possible mechanism for managing behavior implementation. A less extreme version involves the generation of code skeletons. These skeletons provide empty function stubs where behavior logic can be inserted using conventional code development tools. The empty stubs represent an implementation of an abstract interface; the interface itself is generated as part of the code generation process.

The behavior code that is inserted into these empty stubs should not make any direct calls to any platform specific API. Rather, the behavior code should make calls on generic interfaces provided by the generated code. These generic interfaces wrap the platform specific API’s of the pervasive services. This approach to code development ensures that the infrastructure for a simulation can be changed without the need to alter the behavior logic. User code can be migrated to other platforms by simply regenerating the platform specific implementations of the generic interfaces. The user code will continue to compile and operate as designed, as the interfaces to the pervasive services have not changed – only the implementations have been altered. However, it should be noted that this approach does not allow the developer to transition behavior logic to a different implementation language.

3.3 The Role of Tool Support

Central to the MDA process is the role of tools in the design, development and implementation of a system. The use of a common UML based notation encourages the use of visual tools in the design of the PIM. Metadata can also be assigned to PIM elements using visual tools. Automated code generation tools are used to generate wrappers for the simulation elements, and the pervasive services required to service them. Furthermore, tools can
be used to coordinate the compilation, deployment, and execution process for the simulation.

Tools can also be used to provide other features not normally associated with the code development process, such as requirements gathering and the development of conceptual models. The MDA process encourages the use of a single model for all aspects of the application design, with viewpoints into the model that show different aspects of the system for different stakeholders (e.g., Business Analysts, Systems Architects and Programmers). Rather than a project consisting of several independent documents, the MDA approach encourages the use of a single document, with tools customized to the needs of various stakeholders. For example, a requirements gathering engineer might only see use-case diagrams for an application, whereas the viewpoint of a platform engineer may be restricted to deployment aspects of the project. However, all developers operate on a single, internally consistent model. In this way, it is possible to enforce consistency between requirements, implementation and documentation.

Using a UML notation for the PIM has the additional advantage of limiting vendor lock-in for the tool set. Associated with the UML is the XML Metadata Interchange (XMI) file format; this format provides a common method for the serialization of UML documents. This common format helps encourage interoperability between tool sets, limiting the potential for vendor-lock in.

4. DEVELOPMENT BENEFITS OF MDA

Although the MDA process involves the use of a layer of abstraction from native platform implementations (implemented through a standard platform component model such as the CORBA Component Model (CCM) or Simulation Component Model (SCM) (Radeski, et al., 2002), there are a number of significant advantages to this abstraction.

4.1 Improved Reuse and Interoperability

A key benefit of the MDA approach is the increased ease with which behavior logic can be reused between simulations. This infrastructure-agnostic model ensures that the design of a simulation component can be rapidly retasked for any infrastructure, simply by choosing a new PSM and regenerating the platform specific parts of the code base. In this way, simulation components can be rapidly deployed into new simulation infrastructures with virtually no effort.

The use of the abstraction layer in behavior code protects any user-developed code from changes in the infrastructure. As a result, this user code – the simulation logic that forms the bulk of the development effort – can be easily and rapidly redeployed to new simulation infrastructures. Therefore, simulation logic need not be repeatedly developed for each new simulation, or each new target infrastructure. Figure 4 shows how the SCM separates the behavior and integration logic in the component (Radeski, et al., 2002).

Interoperability between simulation components is also improved. For example, the PIM for an individual simulation component contains no details concerning “on-the-wire” representations of data for the component; as a result, all interoperability concerns are transferred to the PSM and the code generation process. In addition, the abstraction API that is exposed to the user does not contain any function points that could affect interoperability (e.g., byte ordering is not exposed to user code). Changes in platform requirements that affect interoperability (e.g., deployment over a network which has a range of byte-endian representations and operating systems) are transparently resolved in automatically generated code.

![Figure 4: Simulation Component Model.](image)

4.2 Lower Complexity and Project Costs

Another key benefit of the separated design process is a reduction in project complexity, and therefore the costs associated with development. These improvements come via a number of avenues.

4.3 Less User Code to Write

The most fundamental difference between conventional simulation development and MDA development is the quantity of code that must be written by developers. Using traditional techniques, the development task consists of designing the simulation, implementing integration code, and implementing simulation logic. Under this development approach the integration and simulation code are often intertwined in the same code block, affecting reuse, readability and maintainability. In contrast, using the MDA approach, the task of implementing integration code is automated, removing
this concern from the developer. If behavior logic is to be
hand written, stub code is generated. This reduction in
coding effort reduces both the complexity of a given
project, and the costs (time and money) associated with
development.

The increased potential for code reuse further decreases
development costs. For example, rather than having to re-
implement the logic behind a requirement each time it is
needed (e.g., a basic physics engine), in an MDA
environment it becomes possible to extract this
functionality from a repository and reuse it in the current
application. Common behaviors need not be redeveloped
and retested for each new project or infrastructure.

4.4 Reductions in required VV&A
A key feature of the separation of simulation logic from
integration code is the ability to perform VV&A on the
simulation logic. Using a test harness PSM, it is possible
to test the internal behavior of simulation logic
independent of the integration logic. Once the simulation
logic has been tested, it is possible to accredit, package
and archive the simulation logic ready for reuse. When
reused, this simulation logic requires no further unit
testing as the accreditation seal does not need to be
broken in order to redeploy it into a new simulation, even
if this new simulation utilizes a different infrastructure.

In addition to the testing of the simulation logic, the code
templates used to generate PSM level features can also be
validated; once validated the templates can be repeatedly
used without concern that the generated code will contain
errors. All that remains is to unit test any new simulation
logic and to perform a systems test on the entire
simulation. This represents a significant reduction in the
quantity and time required to test a system, without any
reduction in confidence in the robustness of the final
system.

4.5 Rapid Prototyping
The MDA approach to development also allows for rapid
prototyping of designs. For example, an initial cut at a
design may call for a highly segmented object model.
This design can be rapidly modelled, generated, and
tested within the MDA tool set. As a result of this testing,
the highly segmented model may prove unfeasible (e.g.,
due to poor performance). If this is the case, the model
can be reorganized to overcome this limitation. After
reorganization the code can again be generated and tested.
This process can be repeated until a feasible system
model is found.

This speed of prototyping cannot be achieved using
conventional development techniques, as each new design
would require a complete re-implementation, or at least a
port from a similar implementation. The ability to rapidly
prototype a simulation reduces the risks associated with
development, as unfeasible designs can be eliminated
early in the development process, rather than presenting
an insurmountable performance barrier at the end of the
development cycle.

4.6 Improved Code Quality
The application of the MDA process and code generation
improves the quality of code produced by a project,
which in turn improves reuse and maintainability.

Skilled platform experts can be used to develop templates
for a PSM. These platform experts understand the
capabilities of an infrastructure, and are familiar with
industry best practice. The resulting templates they
develop will therefore utilize a wide range of tested
design patterns and established industry experience. The
code subsequently generated from these templates will
reflect this best practice. The templates can also be
verified, ensuring that generated code will be as bug-free
as possible.

Code generation also allows engineers who are not
familiar within a given infrastructure to take advantage of
a well-structured framework in which to develop their
behaviors. The MDA approach ensures that the path of
least resistance for developers is to utilize the rich, best
practice environment provided by the code generation
process, rather than adopting their own proprietary and ad
hoc approaches.

4.7 Improved Utilization of Infrastructure Features
The MDA approach encourages the use of the more
complex facilities of an infrastructure. Consider the use of
dead reckoning. Dead reckoning is an ‘on the wire’
optimization technique for minimizing network traffic.
Using traditional development techniques, the developer
would need to be aware of the need for dead reckoning on
certain attributes up front, and develop code accordingly.

However, in MDA the use of dead reckoning would be
treated as an implementation concern, which has no effect
on the underlying behavior of a simulation. The developer
can be provided with a simple get/set interface for
attributes. Behavior logic can then be developed using
this interface. A simple PSM can satisfy this get/set
interface using an implementation that writes/reads data
directly to/from the simulation infrastructure. However,
by describing the need for dead reckoning in the PIM, this
simple implementation could be seamlessly replaced by
an alternative that performs caching and forward
prediction of values. This new implementation requires
no changes to the behavior code, as the get/set interface is
unchanged - the only effect will be observed during runtime as a reduction in network traffic.

The MDA approach to development decreases the barrier to entry for complex features of an infrastructure (such as Dead Reckoning), increasing the likelihood that benefits of these features can and will be exploited for a given project. Furthermore, the ease with which complex features can be introduced into a project encourages exploration of the capabilities of an infrastructure, as it becomes possible to perform rapid ‘what if?’ testing of features.

4.8 Reduced Exposure to Evolving Standards

Finally, the use of MDA reduces the developer’s exposure to changes in standards. When the “next best thing” becomes available, the MDA process makes migration to this new platform a minor task. A one off development effort is required to create the PSM for the new infrastructure, after which all pre-existing simulation logic can be redeployed by regenerating the integration code from the new templates.

Using traditional development methods, the process of migrating to a new infrastructure would involve porting or redeveloping simulation logic. The ability to deploy to new infrastructures by simply regenerating integration code represents an almost 100% reduction in coding effort, making exploratory testing of new infrastructures a viable development alternative. Similarly, changes in policy that require migration to new infrastructures (such as a recent mandate requiring a migration from DIS to HLA) impose almost no development effort beyond the initial development of a PSM. This reduces the exposure of the developer to the effect of changing standards.

5. CASE STUDY

The development benefits outlined in the previous sections are not merely theoretical. In this section we will provide a case study of two implementations of the DMSO “HelloWorld” HLA example. The first implementation is the raw HLA implementation provided by DMSO as part of the RTI installation. This version has been written to directly use the RTI’s API. The second implementation uses an MDA approach using the Calytrix SIMplicity™ MDA tool suite (Calytrix, 2002). This comparison will demonstrate the significant advantages of adopting an MDA approach to simulation development.

5.1 Overview

The “HelloWorld” example is a simple demonstration of the basic capabilities of HLA. It consists of a single federate, which creates an object class called Country, and an interaction called Communication. The Country object has two attributes: the name of the country and its population. The Communication interaction has a single parameter used to communicate messages between countries.

Each federate instance in the federation publishes and subscribes to both the Country object class and the Communication interaction. As the population of a country changes it is published to the RTI; these changes are received by each federate. Communications are occasionally sent between countries. The code for these examples is written in C++ using the DMSO RTI1.3ng6.

5.2 An Ad hoc HLA Implementation

The sample implementation of HelloWorld provided with the DMSO RTI consists of:

- 556 lines of code in 2 header (.hpp) files,
- 2213 lines of code in 3 source (.cpp) files,
- 602 lines of configuration (FED, RID) files.

This is a total of 3,371 lines of hand written code. This code is well documented; much of the line count represents documentation of the system logic. The testing effort associated with this project is a combined test of the logic for interacting with the RTI (i.e., are the correct calls to the RTI made in the correct order), and a test of the behavior logic (i.e., is the formula for population growth correct). These test regimens are difficult to separate, as the behavior logic is tightly coupled to the integration code.

In addition to this coding effort, there may have been a design phase during which requirements for the simulation, such as the definition of the object model, start-up sequence, and timing requirements were defined. These design artifacts do not form part of the delivered product, so if changes are to be made to the HelloWorld simulation, the design documents must either be redeveloped, starting the project from scratch, or the design must be reverse engineered from the available code.

5.3 MDA Implementation

Unlike the native HLA implementation, the MDA development process explicitly requires a number of design steps to be performed before any code can be generated or written by the developer. The simulation is modelled in an MDA tool and the resulting design files become a core part of the delivered project.

5.4 PIM Definition:

The first step in an MDA implementation of the HelloWorld example is to build the PIM. The PIM definition is performed in two parts. Firstly, the developer must define the entities that will exist in the simulation;
the object classes, interactions, and federates. In this simulation, these definitions describe the Country object class, the Communications interaction, and a single federate – the HelloWorld federate. These definitions can be expressed using a UML based notation; alternatively, they can be defined using a tree notation. In addition, documentation can be associated with the UML elements.

Secondly, the interactions between the federates and the object model must be defined. In this simulation, the HelloWorld federate publishes and subscribes both the Country object class and the Communication interaction. Figure 5 shows one mechanism for defining the relationships between the federate and data model; other formats for defining these interactions have been proposed [Keith-Magee, et al., 2002].

This is a total of 3400 generated lines of code. This code is documented to the same standard as the *ad hoc* project, however no developer effort was required. In addition, *makefiles* for this project were also produced.

While the generated project does have a slightly higher line count than the *ad hoc* HLA implementation, this can be attributed to the heavy use of design patterns within the generated code. The differences in line count in the FED/RID files can be attributed to a combination of differences in formatting: there are few functional differences between the FED/RID files in the two examples.

It should also be noted that up to this point the developer has not written any lines of code, rather they have been automatically generated from templates and the PIM/PSM expressed in the project’s design.

### 5.7 Behavior Logic:

The stub code produced by the tool suite contains no behavior logic – it contains only the code necessary to produce a federate participating in a federation as described by the PIM and PSM. The user must then add behavior code to the stubs. In this project, it is necessary to add:

- 7 lines of code to the header files (local variable definitions, method prototypes), and
- 67 lines of code to the source files (behavior to calculate and publish the changing population and occasional communications).

The total user coding effort for this project therefore amounts to 74 lines of code (see Appendix A). This code is pure behavior code – the code required to instantiate a Country object, calculate changes in population and reflect these changes, and occasionally send a message to other federates.

This user code is not written using native HLA function calls – rather, it uses generic interfaces that wrap the underlying HLA function calls. This has two benefits; firstly, it provides a vastly simplified interface to the developer. For example, the HLA interface to advance time is asynchronous and requires 12 lines of code. This call sequence can be abstracted behind a single synchronous call. The use of this synchronous interface is also more intuitive than the native HLA interface.

Secondly, the resulting behavior code is independent of infrastructure. By replacing the implementation of the wrapped function call, it is possible to transparently change the infrastructure in which behavior is deployed. For example, the HLA implementation of the advance

![Visual Design Model](Calytrix, 2002).
time interface could equally be satisfied by a series of analogous calls from any other simulation infrastructure. Alternatively, a dummy testing harness could be used. This would allow the behavior code to be tested independent of any infrastructure.

5.8 Comparison

5.8.1 Lines of Code Written

The ad hoc implementation of this project requires 3371 lines of code to be written by hand. The MDA implementation requires only 74 plus some visual modeling. The user code required for this project can be found in Appendix A. These line counts demonstrate a 45-fold reduction in coding effort for this project. Although this is a small HLA project, anecdotal evidence suggests that the reduction in coding effort is representative for larger projects, as HLA code overhead increases linearly with the number of object classes, interactions and federates in the simulation.

5.8.2 Testing

The MDA project also has a significantly reduced testing requirement. The ad hoc project requires testing of all 3371 lines of code, which is difficult to unit test as it is tightly coupled to the use of a specific RTI. However, in the MDA solution only the 74 lines of behavior logic need to be tested, as the templates guarantee the quality of any automatically generated code. The behavior code is also relatively easy to unit test as it is decoupled from the RTI.

5.8.3 Extra Features

The MDA approach affords the exploratory use of extra RTI features. For example, using the MDA approach, Dead Reckoning can be transparently enabled for the population attribute of Country by adding the feature to the PIM and regenerating the code. Adding dead reckoning to the ad hoc implementation would be a difficult task, as there is no separation between the logic that updates values from the logic that communicates these changes to the RTI.

The MDA approach allows an easy migration path to new infrastructure platforms. By simply changing the template backend and regenerating the integration code, the behavior logic is ready for deployment into the new infrastructure. This is possible because the behavior logic is written using generic interfaces rather than specific RTI calls.

The quality of the code produced by the MDA project is also superior. The extensive use of design patterns in the generated code makes it easy to identify where the key logic for the population update is contained. No such guidelines exist in the ad hoc project. While this is not a significant feature for a small project such as HelloWorld, on larger projects the existence of a consistent logical structure between projects significantly simplifies the project development and management task.

5.9 Summary

This case study has gone some way to demonstrate the advantages of using an MDA approach for simulation development. Using MDA it has been possible to reduce the coding requirement by a factor of 45 in this example, while substantially reducing the testing requirements. The code that must be written by hand is entirely behavior code, as all the infrastructure code is automatically generated from the design. The API used by developers has been significantly simplified, while the barrier to accessing the more complex features of the RTI has been reduced. Lastly, the behavior logic has been insulated from changes in infrastructure, so when a new simulation standard arrives on the scene it becomes a relatively minor task to redeploy the existing simulation logic into the new environment.

6. CONCLUSION

The theory and case study presented in this paper demonstrates the wealth of opportunity that is available to those in the simulation community that choose to adopt the MDA development model. The MDA approach promises to deliver higher quality of code, with less coding and testing effort. In addition, MDA increases the ease with which simulation behavior can be exposed to new infrastructures and new features within existing infrastructures. Finally, MDA maximizes the integrity of documentation and design artifacts of a system by ensuring these documents form a key part of the system.

In order for the simulation community to realize the benefits of MDA, several things are required. Firstly, the use of MDA toolsets must become a widespread and accepted part of the simulation development landscape. Secondly, the simulation community must agree upon standardized notations (such as a UML profile for simulation), and standards for free data interchange.

However, most important of all is the acknowledgement that simulation development is a branch of software engineering. Although simulation development has very specific and unique requirements, the lessons learned by the software engineering community can and should be adopted by the simulation development community. Unless the lessons learned by the software engineering community are accepted and integrated into simulation best practice, the simulation community is doomed to repeat the “green elephant” (Tolk, 2002) development efforts of the past.
REFERENCES
APPENDIX A - HELLO WORLD FEDERATE C++ SOURCE CODE

The bold text indicates code that was added to the original generated simulation class. Also the #includes and namespaces have been trimmed for brevity.

HelloWorld.h

class HelloWorld : public sim::Federate {
protected:
    CountryHome* m_countryHome;
    Country* m_country;
    sim::String m_message;
    sim::Long m_loop;
public:
    HelloWorld();
    virtual ~HelloWorld();
    virtual void startup();
    virtual void configure();
    virtual void initialize(sim::Reader& reader);
    virtual void simulate(sim::Time time);
    virtual void cleanup();
    virtual void save(sim::Writer& writer) const;
private:
    sim::Double calculatePopulation(const sim::Double pop);
};

HelloWorld.cpp

void HelloWorld::startup() { }
void HelloWorld::configure() { }
void HelloWorld::initialize(sim::Reader& reader) {
    m_countryHome = createCountryHome();
    m_country = m_countryHome->createInstance();
    reader.readString("Message", m_message);
    reader.readObjectClass("Country1", *m_country);
}

void HelloWorld::simulate(sim::Time time) {
    auto newPop = calculatePopulation(m_country->getPopulation());
    m_country->setPopulation(newPop);
    m_countryHome->updateInstance(m_country);
    if((m_loop % 20) == 0) {
        Communication* comm = createCommunication();
        comm->setMessage(m_message);
        m_interSvc->send(comm);
        delete comm;
    }
    CountryList l = m_countryHome->getCountryList();
    for(CountryList::iterator it = l.begin(); it != l.end(); it++) {
        std::cout << "Population for " << (*it)->getName() << " is "
                  << (*it)->getPopulation() << std::endl;
    }
    m_timeSvc->requestTimeAdvance(time+5.0);
    m_loop++;
}

void HelloWorld::cleanup() { }

void HelloWorld::save(sim::Writer& writer) const {
    sim::Double HelloWorld::calculatePopulation(const sim::Double pop) {
        sim::Double p = pop * 1.2;
        return p;
    }