A VISUAL TOOL TO SIMPLIFY THE BUILDING OF DISTRIBUTED SIMULATIONS USING HLA

Shawn PARR

Introduction

The High Level Architecture (HLA)[1] is intended to promote the reuse and interoperability of distributed simulations. While in many respects HLA achieves these goals, it unfortunately also adds additional cost and complexity to the development task, resulting in the need for specialist HLA skills.

This paper outlines the problems currently faced by simulation developers wanting to use HLA, and the way they are addressed by the Calytrix SIMplicity product.[2]

About HLA

Why Use HLA?

In an ideal world, a developer could write a component once and then reuse it in any simulation in which it played a part. This would have a variety of benefits:

- Simulations could be more quickly and easily constructed at a reduced cost.
- It would become easier to construct larger and more sophisticated simulations assembled from existing components.
- Component quality would increase, as more work would be focused on improving existing component functionality rather than rewriting components from scratch.
- Components from different developers and different projects (potentially in different parts of the world) could be combined in new simulations.

The High Level Architecture was introduced to facilitate simulation reuse and interoperability in order to realise the above benefits. HLA addresses a number of the limitations imposed by the data protocol approach associated with the earlier Distributed Interactive Simulation (DIS) standard. HLA has been mandated by the

U.S. Department of Defence, has been published as a standard by the Institute of Electrical and Electronics Engineers (IEEE)[3] and the Object Management Group (OMG),[4] and is being adopted by creators of simulation software worldwide.

The Problems with HLA

While there are good reasons to use HLA to develop simulations, there are also drawbacks. The learning curve for HLA is steep, and a lot of extra work and code is needed to build the necessary software infrastructure needed for HLA compliance. Specific problems that simulation developers encounter include:

- The HLA "glue" code required to bind a simulation component to the RTI[5] is often tightly coupled or intertwined with the simulation code. This makes the code unnecessarily complex and difficult to change and reuse.
- Due to the complexity of the RTI interface, specialist-programming skills are needed to write HLA compliant components.
- A number of cross platform issues introduce unnecessary portability and interoperability issues in HLA development (one example of this is the handling of "big-endian/ little-endian" conversion between hardware architectures).
- In a single simulation, all HLA components (known as "federates"[6]) must use the same data specifications as defined in the simulation's Federation Object Model (FOM). For example a location cannot be sent as 'latitude and longitude' in one component and received as 'eastings and northings' in another. This means that a component cannot be easily taken out of one simulation and reused easily in another unless they use exactly the same data types and format conventions. This problem is often referred to as "FOM Agility."
- Due to the complexity of HLA there is a tendency to maintain a relatively coarse granularity at the federate level in order to minimise the number of federates to develop (hence minimising the pain of RTI integration). However, it is often more desirable to build finer grained components in order to maximise the potential for re-use and extension.
- There are two incompatible HLA standards: DMSO 1.3 and IEEE 1516. Federates written for one standard cannot easily interoperate with those written for the other, thus undermining the key goals of interoperability and reuse.

SIMplicity solves the above problems, thereby making it easer for the simulation community to create large-scale, high fidelity simulations constructed from reusable and exchangeable simulation components.

Addressing the Problems with HLA

In order to address the problems described in the previous section, Calytrix has developed SIMplicity, which delivers an IDE for HLA development with the following attributes:

- Simple to use. Much of the work can be done through a visual interface so that specialist HLA skills are not needed.[7]
- Introduces a Simulation Component-Model (SCM) to HLA development.[8,9] This allows SIMplicity to decouple a component's simulation logic from its HLA "glue" or integration code, thus simplifying simulation development and making component simulation logic more reusable.
- Automatically handles the binding of the simulation code to the HLA infrastructure (by generating the FOM and federates infrastructure code), thus removing much of the "grunt work" associated with developing the RTI API.
- Handles transformations between simulation components. This addresses inter-platform issues (like "big endian – little endian"), and data translations between components created for different FOMs (FOM Agility).
- Allows developers to decompose a federate's functionality into a collection of finer-grained reusable components.
- Utilizes a Model Driven Architecture™ approach[10] to development that enables developers to easily transition and reuse their existing component's simulation logic with different RTI versions (including reuse between 1.3 and 1516 standards) and future simulation middleware.

An Introduction to SIMplicity

A simulation developer should be concerned with ***what*** *the simulation does, not* ***how*** *it integrates with the HLA infrastructure.*

Calytrix SIMplicity is an integrated development environment (IDE) that enables software developers and scientists to rapidly create component-based simulations from new and pre-existing components in a visual environment.

In this section we will introduce the underlying concepts and architecture of the SIMplicity development environment, as well as providing an overview of the component-model adopted.

Adopting an MDA Approach

Design and development within the SIMplicity IDE is based on the OMG's Model Driven Architecture (MDA) approach.[11] In summary, MDA provides a common approach for designing and building a system that remains decoupled from the eventual languages, platforms and middleware environments they will be used in. The key advantage to MDA is *future proofing*, as it provides a mechanism for an organization to design their systems once and then transition them over time when *the next best thing* comes along.

Following the MDA approach, developing simulations within SIMplicity is made up of the following phases:

Phase 1: The developer creates a *platform independent model* (PIM) for their simulation using UML[12] and specialized notation. The PIM remains independent of the eventual middleware infrastructure or RTI implementation that the simulation will be deployed into.

Phase 2: From the PIM the developer further refines the model to create a *platform specific model* (PSM). For example, in a simulation context a PIM can be refined for either an HLA 1.3 or IEEE 1516 PSM. It is important to note that the PIM and PSM remain separate, allowing a single PIM to be refined to a number of PSMs without having to re-implement the simulation logic.

In combination, the PIM and PSM provide a complete description of the simulation components and the infrastructure and services required to execute the system.

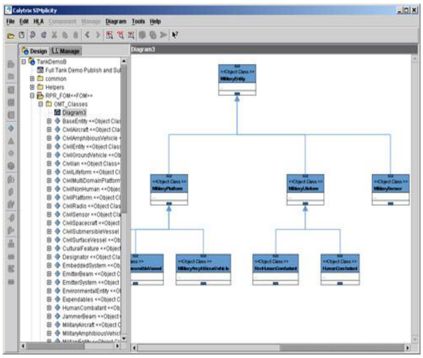
Phase 3: Based on the PIM and PSM meta models, a template based code generation engine can be used to generate the simulation's code, resulting in compilable federates that will execute on the targeted platform; all that remains is to insert the required simulation logic or behavior into the place holders created during the generation process (see the Simulation Component Model section below).

Lets now examine each of these phases, in relation to HLA and simulation, in more detail:

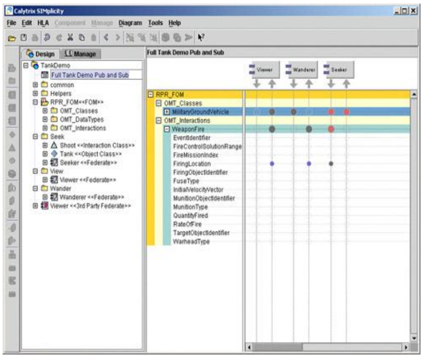
Phase 1 - Design your simulation

Central to the development process are visualizations to assist and simplify the design and specification of the simulation and its participant components. Starting from a blank canvas it is easy to model a simulation, from the base data elements and FOM to the federates and their relationship with each other.

SIMplicity employs a number of UML and specialized diagrams to allow the developer to rapidly construct a simulation model (see Figure 1).



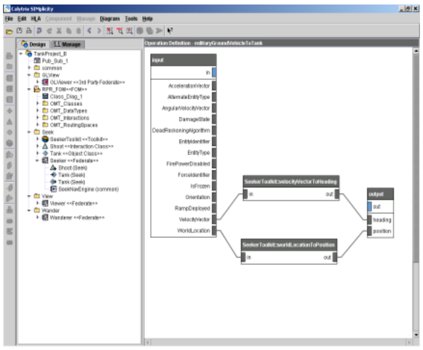
A UML Class diagram is used to model the simulation's base data elements and FOM.



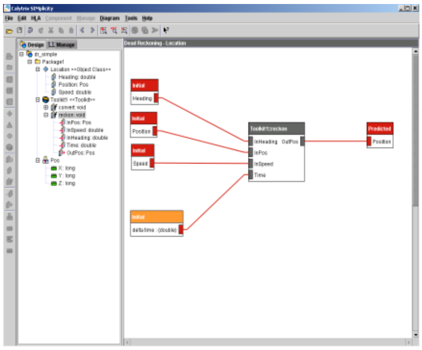
SIMplicity's Publish and Subscribe Diagram, relationships in the model.

Figure 1: Simplicity's Diagrams.

As part of the design process the developer will also define the relationships between the individual federates. This includes modeling transformations between semantically equivalent but syntactically different data items, allowing you to incorporate federates that use different SOM elements into your simulation's FOM (FOM-Agility). Similarly, dead reckoning and threshold values can be applied to published data objects through the IDE, reducing the amount of data traffic exchanged at execution time (see Figure 2).

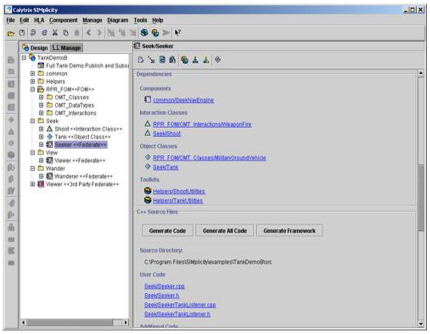


Connection diagrams allows the user to visual define complex data transformations between interfaces.

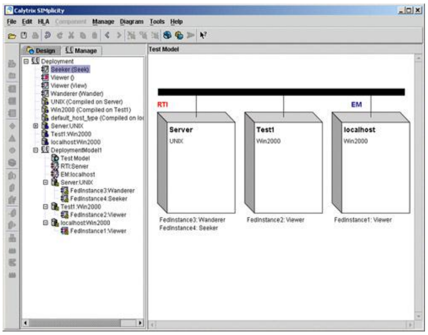


Link graphs are employed to easily capture complex dead-reckoning requirements.

Figure 2: Simplicity's Connection Diagrams and Link Graphs.



Property pages provide a way for developers to specialize their model.



UML deployment diagrams are used to specify how the system is to be deployed and execute across the distributed network.

Figure 3: Illustration of Property Pages and UML Deployment Diagrams.

By adopting a component model (see below) these types of integration refinements can be changed and regenerated seamlessly into a federate's integration code without having to revisit the existing simulation logic.

At the completion of this design phase the developer has specified their simulation's *platform independent model*.

Phase 2 - Refining your simulation

Following the high-level design phase the developer specializes or refines the simulation's PIM to the target environment to create a *platform specific model*. The PSM identifies key platform specifics such as the code generation language (C++, Java, VB) and target simulation architecture, including HLA vendor and version information.

As part of the PSM process the developer may need to model the physical deployment, via a UML diagram, of their simulation. Physical deployment will have an impact on issues such as byte ordering and host type, all of which needs to be taken into consideration during the code generation and compilation process. Figure 3 illustrates property pages and UML deployment diagrams.

Phase 3 - Generation and execution

Once the PIM and PSM are complete a template-driven code generation engine can be employed to create all the components and configuration files for the simulation. At the end of this process the developer has a compilable simulation that will execute on the targeted platform; all that remains is to insert the simulation logic or behavior into the generated components.

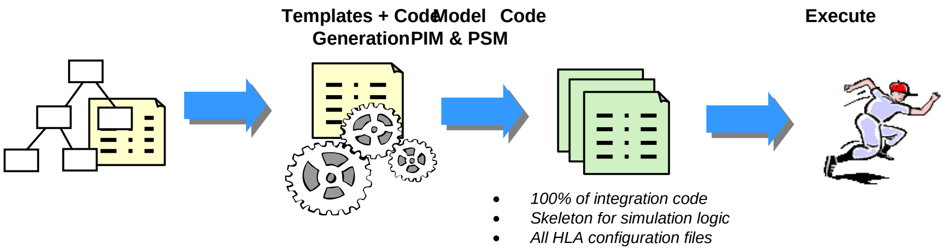

Templates + Code Generation
Model PIM & PSM
Code
Execute
100% of integration code
Skeleton for simulation logic
All HLA configuration files

Figure 4: Simulation Life Cycle.

Local and remote compilations can also be managed through the same IDE, thus ensuring that the simulation is ready to be executed on the modeled network. Once the simulation has been built, tools can be used to package, deploy and execute the distributed simulation directly from the unifying IDE.

The diagram shown in Figure 4 summarizes the design, code generation and execution process used to manage the simulation life cycle.

Under the Hood

The Simulation Component Model

One of the core objectives of our work has been to insulate the federate developer from as much of the RTI infrastructure as possible, therefore lowering the barrier to HLA entry. Driving this objective is the ability to enable scientist and non-middleware programmers to develop simulation logic in their preferred component-based development language (C++, Java, Visual Basic .NET etc) with little knowledge of HLA and that these components can then be rapidly reused in any HLA simulation.

In order to achieve this objective we have created the Simulation Component Model (SCM),[13] which describes a programming pattern for developing federates based on the CORBA Component Model (CCM).[14] To help explain the SCM the diagram in Figure 5 shows the current programming responsibilities using just the RTI compared to that with the SCM.

As the above diagram shows, the SCM separates the HLA 'glue' code, which resides in the automatically generated integration code, from the simulation logic. In contrast, without a component model managing the developer would have to construct the main execution loop and the simulation ambassador from scratch, while using the RTI API to integrate the component into the HLA environment, as well as managing all the underlying plumbing issues like marshalling and un-marshalling of data to and from the RTI.

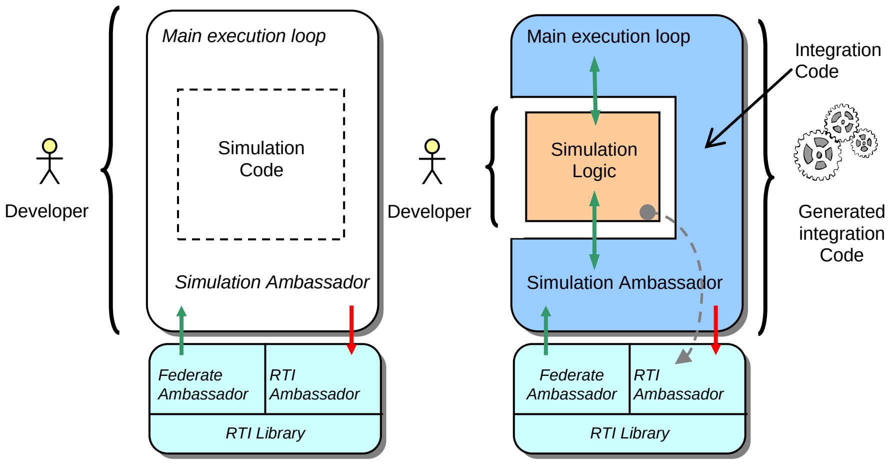


Figure 5: Developer Responsibilities with Just RTI (left) and with the SCM (right).

When creating new federates, the developer defines their interfaces and relationships within the visual environment, which in turn can be interpreted and used by a code generation engine. By using customizable templates it is possible to ensure that the generated code exploits good OO techniques and design patterns, providing the developer with a well engineered and consistent code base (a simple class to provide the simulation logic in), as well as employing abstractions to insulate the developer from most of the generated HLA boilerplate code. Most common HLA functions, such as publishing and subscribing to data objects and interactions, and basic timing models, are seamlessly handled in the generated code, insulating the developer from writing any RTI calls.

In addition, the separation of simulation logic and integration code provides a mechanism to modify and transition a component between different HLA implementations without having to revisit or update a federate's tested simulation logic. For example, you can regenerate the integration code for different RTI implementations without impacting the simulation logic code (see Figure 6).

Architectural Overview

In order to create an extensible MDA based architecture that can support a range of varied and changing infrastructures, it is important to build a "pluggable" architecture that can accommodate change. To this end, the SIMplicity architecture can be broken into four key components:

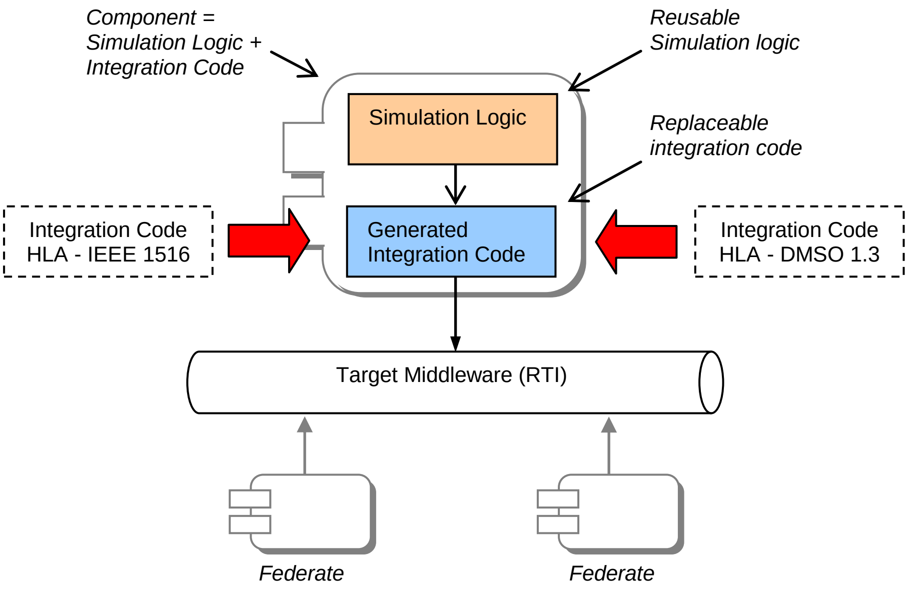

Component = Simulation Logic + Integration Code
Reusable Simulation logic
Simulation Logic
Replaceable integration code
Integration Code HLA - IEEE 1516
Generated Integration Code
Integration Code HLA - DMSO 1.3
Target Middleware (RTI)
Federate
Federate

Figure 6: Separation of Simulation Logic and Integration Code.

GUI Layer:	The GUI or presentation layer provides the user interface and manages the modeling and visual aspects of the system.
Meta Object Model:	The Meta Object Model layer maintains the design internally, which is a highly customizable data structure for describing and storing all the PIM and PSM characteristics of a model.
Platform Engine:	The Platform Engine is responsible for generating all the code for the simulation. This is achieved by mapping the characteristics held in the Meta Object Model to the corresponding code templates.
Plug-ins:	SIMplicity supports a pluggable architecture for incorporating and updating PIMs, PSMs and code templates. An *MDA Plug-in Developers Kit* will allow organizations to customize the design and code generation process to suit their particular requirements.

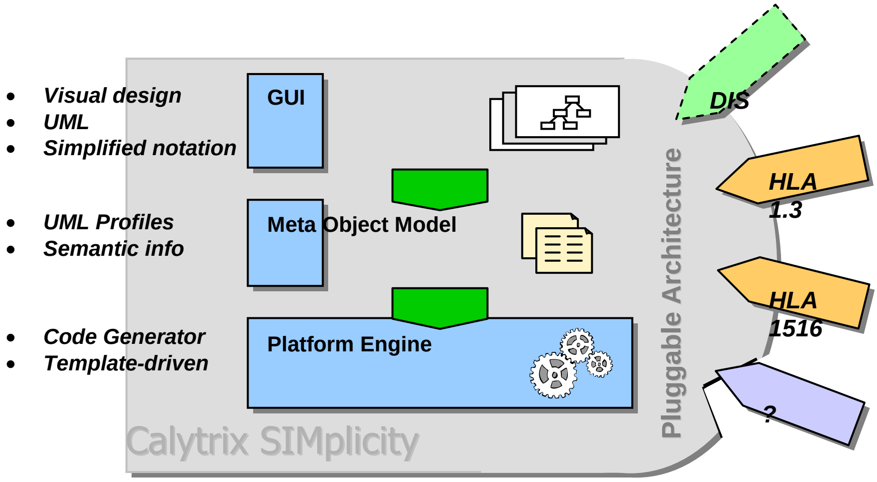


Figure 7: SIMplicity's Architecture.

The diagram shown in Figure 7 provides an overview of the SIMplicity architecture.

SIMplicity is working towards supporting multiple plug-ins, allowing the developer to define the following platform specific characteristics through the IDE:

- Simulation architectures: HLA (HLA 1.3 and IEEE1516 standards) and DIS;
- HLA platform transitions: from NG4 → NG5 → NG6; and
- Component languages: C++, Java and .NET.

Addressing Potential User Concerns

This section outlines concerns users may have and discusses how these have been addressed.

Can You Really Abstract the Developer from HLA?

The HLA integration code that SIMplicity generates contains a layer of abstraction that sits between the developer's code and the RTI interfaces, thereby directly shielding (not replacing) the developer from the RTI API. This results in a much smaller set of code to be maintained by the developer. In the end, the developer is only required to know about the simulation in general and not HLA specifically.

Using this method the developer is still able to directly access the RTI from their simulation logic code if required.

What about Timing?

SIMplicity provides the most common timing models used by simulation engineers. Should any advanced time management be required, such as optimistic timing, the developer is able to extend the generated code to support this.

Avoiding Vendor Lock-in

Users have expressed a concern about being too dependent on a single vendor. This is addressed by SIMplicity in a number of ways:

- SIMplicity is non-intrusive at the federate level, allowing SIMplicity created federates to be deployed and used in non-SIMplicity environments without requiring any additional or third party run-time services.
- SIMplicity provides the developer with all generated code. There are no proprietary APIs or runtimes required to use a SIMplicity HLA component in a running simulation.
- SIMplicity supports multiple RTI implementations and middleware infra-structure.
- Wherever possible SIMplicity utilises both existing standards (like HLA, XML, UML, MDA, etc) and component standards (like the CORBA Component Model (CCM)).

Increasing Federate Fidelity

One of the major concerns with increasing the fidelity of a simulation by decomposing federates into many smaller components is that of performance, as replacing one high-level federate with a composite of smaller federates which communicate via the RTI may adversely affect the simulation's performance due to an increase in RTI and network traffic.

SIMplicity overcomes this issue by providing a component-based solution within a federate. Here, entities are represented as independent reusable components that communicate through interfaces. These interfaces are subject to the same transformation facility as regular federate interfaces.

Limiting Power Users

With any visual or 'ease of use' tool there is the concern that it imposes limitations on power users. SIMplicity addresses this in several ways:

- Significant flexibility is built into the visual environment to accommodate a wide spectrum of users. This includes access to different timing schemes, data exchange, transformations, etc.
- SIMplicity works alongside existing technologies and methodologies. The use of SIMplicity will not prevent interoperability with components or

simulations created by hand or with other tools.

- The use of SIMplicity does not preclude the use of the original RTI API. Should any specific HLA behavior be required then the developer is free to provide this implementation.

Performance

Reasonable performance is recognized as a key requirement and SIMplicity is built to ensure it meets acceptable performance criteria.

The code generated by SIMplicity represents the code that the simulation developer would normally have to write. That is, there is no ‘additional’ code being executed – the developer is merely responsible for less of it.

Cross-Platform Support

Portability is seen as a key requirement. The follow platforms are currently supported with more to follow:

- Windows 2000/NT/XP
- Linux Redhat 6.2 and 7.2

Is Re-use Really Achievable?

Reuse is a core goal in moving to HLA based simulations, however there are some basic logistical issues that can prevent re-use goals being achieved. SIMplicity addresses:

- Storing and cataloguing components (pigeon hole problems)
- Finding components in a large repository (cataloguing problems)
- Configuration management of components (version control problems)
- FOM-Agility (incompatible component interfaces)

Sharing Binary Components (Protecting IP)

SIMplicity through the component repository provides the ability to distribute and share simulation components without releasing source code, thus protecting valuable intellectual property and meeting security requirements.

Conclusion

The adoption of HLA will provide an opportunity to realize the benefits of reuse and interoperability for those involved in developing simulations. However the complexity associated with HLA is hindering its adoption. SIMplicity solves many of the problems associated with HLA development, making it feasible for developers to create HLA simulations without specialist HLA or middleware knowledge.

Notes:

1 *IDL Application Programmer's Interface* (Defense Modeling and Simulation Office, 1998), <http://www.dmso.mil> (5 September 2003).

2 Shawn Parr, Alex Radeski, and Robert Whitney, "The Application of Tools Support in HLA," in *Proceedings of the Simulation Technology and Training Conference 2002 (SimTecT 2002)*, Paper ID 26 (May 2002).

3 *IEEE Std 1516-2000 - IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) — Framework and Rules* (Institute of Electrical and Electronics Engineers Inc, approved 21 September 2000), <http://www.ieee.org> (20 September 2003).

4 *Distributed Simulation Systems Facility v1.1 specification* (Open Management Group), <http://simsig.omg.org/> (20 September 2003).

5 RTI is an HLA term for "Run Time Infrastructure". The RTI provides the run time services needed to allow HLA components (federates) to interoperate.

6 A "federate" is an HLA component. Any number of federates can make up a "federation" which simulates a specific scenario.

7 Russell Keith-Magee and Shawn Parr, "Visualising Distributed Simulation Design And Deployment," in *Proceedings of the Interservice/ Interindustry, Simulation and Education Conference* (Paper ID 258, 2002) <http://www.iitsec.org> (29 October 2003); Parr, Radeski, and Whitney, "The Application of Tools Support in HLA."

8 Alex Radeski, Shawn Parr, Russell Keith-Magee, and John Wharington, "Component-Based Development Extensions to HLA," in *Proceedings of the 2002 Spring Simulation Interoperability Workshop*, Paper ID 02S-SIW-046 (SISO Spring 2002).

9 Alex Radeski and Shawn Parr, "Towards a Simulation Component Model for HLA," in *Proceedings of the 2002 Fall Simulation Interoperability Workshop*, paper ID 02S-SIW-079 (SISO Fall 2002).

10 *Model Driven Architecture* (Object Management Group), <http://www.omg.org/mda/ > (29 October 2003).

11 "Calytrix SIMplicity -- An MDA Approach to Distributed Simulation" (Calytrix Technologies Pty Ltd, August 2002), <http://www.omg.org/mda/mda_files/ Calytrix_SIMplicity_OMG_MDA_Release.pdf> (29 October 2003).

12 Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide* (Addison Wesley, 1999).

13 Radeski, Parr, Keith-Magee, and Wharington "Component-Based Development Extensions to HLA;" Radeski and Parr, "Towards a Simulation Component Model for HLA."

14 *CORBA Component Model Tutorial*, ccm/2002-0401 (OMG CCM Implementation Group, April 2002).

SHAWN PARR is the co-founder and Chief Technical Officer at Calytrix Technologies, an Australian based research and development company specializing in HLA simulations, the Model Driven Architecture and component-based design. He has been working in the IT industry for over 10 years and holds a Bachelor of Science degree and a research based Masters Degree. Address for correspondence: EIR Building, 1 Sarich Way, Technology Park, Bentley, WA 6102, Australia. Phones: Australia: +61 8 9362 5300; USA: +1 717 560-7874.