

Component-Based Development Extensions to HLA

Alex Radeski

Shawn Parr

Russell Keith-Magee

Calytrix Technologies Pty Ltd

PO Box 1173, Bentley, WA 6982, Australia

+61 8 9362 5300

{alex.radeski, shawn.parr, russell.magee}@calytrix.com

Dr John Wharington

Defence Science and Technology Organisation

Marine Platform Division

Aeronautical & Maritime Research Laboratory

Po Box 4331, Melbourne, Victoria, 3001, Australia

john.wharington@dsto.defence.gov.au

Keywords:

Component Based Development (CBD), High Level Architecture (HLA),

Reuse, Federate Aggregation, CORBA Component Model (CCM)

ABSTRACT: *Traditionally, High Level Architecture (HLA) development has been seen as a FOM-centric activity, where all inter-Federate interactions (Object and Interaction Classes) are described and then manually coded in the Federate using a publish and subscribe pattern. This development approach tends to lead to FOM lock-in, poor Federate reuse and an unwieldy code-base.*

Significant improvements in the reuse and portability of Federates and federations can be achieved by applying a Component-Based Development (CBD) methodology to the HLA environment. These improvements are realized through the use of abstraction to insulate from boilerplate RTI API code, the improvement of translation and transformation services, and the improvement of component aggregation in both RTI and Non-RTI based component integration infrastructures. These improvements lead to better simulation granularity and fidelity, and improved simulation performance by enabling Non-RTI integration between aggregated Federates using the same unchanged code-base.

This paper looks at the difficulties currently inherent in the HLA environment, how CBD principles can be applied to HLA and the advantages that CBD delivers to the simulation developer.

1. Introduction

Component Based Development [1, 2] describes a software development process where the business logic of an application can be clearly separated, via an interface, from its underlying integration and service requirements. This separation allows the developer to decompose the business logic for the application into software components, where each component is a self-contained and self-describing piece of code. This component-based approach to design and development leads to better reuse of code, and the concept of 'composable' applications through component assembly and aggregation.

The application of CBD to the High-Level Architecture (HLA) promises to deliver a number of significant benefits to the development of simulations. The CBD

methodology promotes the separation of the simulation logic from the rigors and complexities of the Run-Time Infrastructure (RTI) integration API. This clear separation removes any explicit code dependencies from the simulation logic on a known Federation Object Model (FOM) and its related Object and Interaction Classes, making it possible to realize a far greater degree of reuse and flexibility in the development of distributed HLA simulations.

In this paper we will introduce the motivations behind developing a component based methodology and framework for HLA. We will then introduce CBD and examine how it can be applied to HLA development via an example, as well as discussing the potential benefits CBD has to offer the simulation community.

2. Motivation

The High-Level Architecture was developed by the US Defense Modelling and Simulation Office (DMSO) to facilitate the development of reusable and interoperable distributed simulations. HLA describes a set of simulation services, accessible through the RTI's API, which can be utilized to create time-managed HLA compliant simulation objects. The compliant simulation objects, called Federates, can then be combined in an interchangeable fashion to develop complex simulations of real-world phenomena.

While the goals of reuse and interoperability are clearly stated in HLA, in many implementations these characteristics have been difficult to realize. This can be partially attributed to the FOM-Centric nature of HLA, which mandates that all Federates in a given simulation must comply with a predefined set of Object and Interaction Classes as described in a shared FOM. These dependencies often find their way into the developed code, as the developer must explicitly reference by name the data items described in the FOM. This tight coupling, in code, of simulation logic, FOM dependencies and RTI middleware integration requirements dramatically restricts reuse and portability, as even the smallest name change or FOM extension can cause multiple points of failure within the code of a Federate.

While the HLA specification encourages the reuse and interchange of Federates between simulations, it does not address the reuse of simulation logic code within Federates. At the simulation level, each Federate can be regarded as describing an atomic simulation object. The HLA specification does not address the interchange of these objects, as a result there is limited incentive for the sharing of code between HLA Federates. The failure to reuse these simulation code elements results in both duplicated engineering effort and an increase in testing overhead.

The potential for Federate reuse is also impeded by the absence of a mechanism for mapping semantically equivalent, but syntactically different, Federate interfaces. This problem is often referred to as FOM Agility [3]. For example, one Federate may publish velocity in meters per second using rectangular coordinates while a second Federate may expect to receive velocity descriptions in knots using polar coordinates. Although the two interfaces are semantically identical, HLA provides no native mechanism for transforming the inputs and outputs of these interfaces in such a way as to permit data interchange and transformation to occur.

At an implementation level it is a laborious and often complex exercise to write HLA Federates, particularly as

coding for the RTI in popular languages is not code-space efficient, which in turn leads to higher development costs. The complexity and steep learning curve associated with the RTI is further hindered by the lack of appropriate tools support, and the absence of pre-packaged solutions and templates for HLA development. Furthermore, the difficulties of developing and debugging a distributed simulation is made worse by the required consensus in wire data formats, such as big-endian/little-endian incompatibility.

This section has outlined a number of the core difficulties currently experienced in HLA development, much of which can be attributed to the complexity of simulation development and the lack of a standardized abstraction layer to simplify the HLA model.

In the following sections we will illustrate how component based development techniques can be applied to HLA to address these issues. In particular we are going to examine how the application of a component model to HLA leads to significantly reduced development effort.

3. The Component-Based Development Methodology

The application of Component Based Development represents a growing movement in the enterprise software industry [4]. The goals of CBD include reduced development effort and increased software quality through the realization of reusable software components, while delivering improved software quality and reliability.

In this section we will provide an abstract overview of component based design and an introduction to the basic building blocks of a generic component model, its implementation and a description of its runtime services. From this grounding we will be able to see how CBD can be naturally applied to HLA.

3.1 An Overview of CBD

Component Based Development is a process that affects both the design and implementation of an application, where in this paper we use the term "application" in its broadest sense to include any piece of executable software, including simulation programs. CBD encourages the logical partitioning of the application into a number of reusable software components, where each software component conforms to a component model. The component model defines a standard for component composition and deployment, as well as providing runtime services to assist in the development of components.

The primary goals of Component Based Development are [1]:

- to develop or assemble software from pre-existing parts;
- to reuse these parts in other applications; and
- to easily maintain and customize these parts to produce new functions and features.

By taking a more holistic view of an application we can divide the implementation into two parts: the *business logic implementation* that solves a specific domain problem, and the *integration implementation* that glues the business logic to a service framework, such as an Operating System or Middleware (see Figure 3.1). CBD places a strong emphasis on decoupling the business logic from the integration implementation within an application.

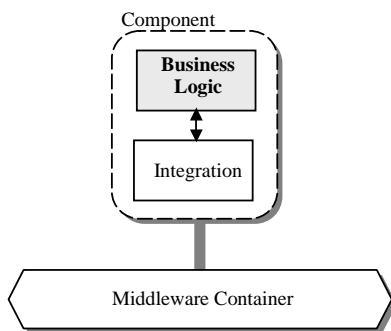


Figure 3.1 – Component Architecture

This approach is in contrast to more traditional coding practices, where both the business logic and integration requirements tend to be tightly coupled into the same piece of application code. This tight coupling restricts reuse, as a developer cannot simply ‘drop-in’ a piece of existing code into an application without first considering and often reworking its intrinsic integration code. The rigid nature of this approach also has an adverse affect on the ease and cost of maintenance and software extension, as again the business logic cannot be considered or changed in isolation.

The decoupling of business logic from integration code yields three significant benefits. Firstly, software reuse is more achievable as the business logic is no longer tightly coupled to its integration implementation. Secondly, CBD code generally matures faster than traditionally developed code, as the separation of business and integration concerns inherent to component based design enables easier unit testing, which in turn allows code to mature faster. Thirdly, shorter development cycles can be

achieved, as blocks of mature business logic are more easily reused through multiple applications.

In many respects CBD is analogous to the motivations of the manufacturing industry with regards to product assembly, for example in the automotive sector where consumer products (cars) are made from a number of prefabricated parts (engines, gearboxes, taillight assemblies). While this approach does slightly limit the overall flexibility of the design, the finished product costs less and is generally more reliable, as each individual component can be designed, tested, and manufactured in isolation.

Furthermore, CBD is not restricted to the development of new applications. CBD methodologies can also be applied to legacy systems through the use of the Façade Pattern [5]. Under this model the Façade is used to reflect the non-CBD interface of the legacy application as one or more components. These components can then be used within a greater CBD application. This approach opens up two avenues for future development. Firstly, it allows the legacy application to remain in production, with all new extensions being built using the Façade interface. Secondly, it provides an interim staging implementation for rewriting the legacy system from scratch. Once complete, the legacy system can then be retired and seamlessly replaced by the new system.

3.2 The CBD Core: A Generic Component Model

Based broadly on the CORBA Component Model (CCM) [6] we can describe a component as consisting of two parts, the *specification* and the *implementation*. The specification can be further broken down into the *component descriptor* and the *component interface*, which are both published in a standardized format. The component descriptor contains metadata associated with packaging and deployment, such as the version information, the vendor of the component, or any applicable copyright notices or licenses.

Services provided between Components are defined in the *component interface*. The component interface supports the declaration of event based messaging and method invocation. The event model is a publish/subscribe model where one component produces an event and all subscribed components consume the event. [7]

The method invocation model has two parts: attributes and operations. Attributes are member variables of a Component that can be accessed by other components. In practice an attribute is associated with accessors (“getters”) and mutators (“setters”) to allow other components to query and modify the value.

The middleware services required by the component are collectively referred to as the component container [8]. These services govern the runtime management and integration of the component, and can include, amongst others:

- Naming services, which provide an index of other available components by name (similar to the White Pages);
- Security management services;
- Services to ensure Quality of Service (QoS);
- Transactional services to ensure data integrity;
- Services that provide component persistence.

The specification defines the services that a given component requires of its container, as well as any qualifying restrictions on those services (e.g., minimum security or QoS levels). The specification and implementation of these services have been developed primarily for the enterprise application integration and electronic business software industries; however, many of the services are generic in nature and are applicable to many other fields, including simulation.

The generic component model development process can be summarized into five stages:

- Design, where the component descriptor and interface are specified;
- Generate, where skeleton implementation is generated from the component interface;
- Package, where the component specification and simulation logic are assembled into a single unit;
- Deploy, where the packaged component is distributed to the site where it is to be executed; and
- Execute, where the component is activated and run in the component container.

The design stage requires manual effort, however the later stages benefit from automation through appropriate software tools.

Design and Code Generation

The implementation of a component is the realization of the Component Interface described in the specification. This realization contains only the business logic of the component. Integration code can be automatically generated for the target infrastructure [8] by tools that read the specification and provide the generated skeletal implementation.

The process of specifying a generalized interface is also applied to the services provided by the component container. In this way, the component business logic is dependent on abstractions of a service rather than an explicit implementation, for example the Java Transaction

Service (JTS) defines a standard interface independent to the actual implementation. This loose coupling leads to a more portable component that can be easily migrated between container versions or vendor implementations.

It is important to note that while the interface of a component is somewhat similar to an OO class, a component is not necessarily a class. A component specification represents an interface to an underlying system; the system can be implemented as a single class, multiple classes, or without classes at all. Nor is the implementation restricted to OO languages (such as Java or C++); CBD frameworks are also available for procedural languages (such as C and Fortran) and functional languages (such as LISP). In much the same way, HLA is defined as an OO language but is not limited to OO languages.

Additionally, new components can be constructed by composing two or more components into a single component specification. This allows complex systems to be developed and deployed by aggregating a number of well-proven, independent components.

Packaging, Deployment and Execution

The clear specification of an interface helps to ensure that the implementation of the component only contains the business logic. Inter-component dependencies are limited to the public interfaces, as no other mechanism is provided for influencing the behaviour of a component's internal functionality or state. In addition, changes in the underlying component implementation cannot affect the overall system, as the contract specified by the interface does not change.

At the completion of development, the component specification and implementation are packaged into a single deployable unit. This package is self-describing and contains enough information to allow the component to be deployed in any compatible container (i.e., any container that satisfies the service dependencies of the component).

The deployable unit allows for additional information to be stored with the component, such as:

- Authorization – the deployable unit may contain organization specific documentation that states that the component implementation has been audited. This helps to guarantee that any stipulated performance and security requirements have been met.
- Verification – a digital signature can be embedded in the deployable unit that guarantees the integrity of the component implementation. This can be of most benefit when developing with third-party components.

- Usability – the component can also be bundled with additional developer documentation to increase usability.

When packaging is complete, the deployable unit can be executed on the required host. At execution time the lifecycle of a component is handled by the Component Home. The Component Home provides a means to create, delete and find component instances. A developer can also extend the Component Home to include helper methods. These methods are analogous to static methods in C++ or Java, as they do not require an object instance in order to be invoked.

The component container can also provide additional runtime services, such as:

- Load balancing – a running component can be deactivated and reactivated at a different location if the local computer system is overloaded.
- Fail-over redundancy – the state of a running simulation is placed in persistence storage that can be restored in the event of a failure.
- Common data encoding scheme – provides a configurable encoding scheme for data passed through the container. This includes implicit big-endian versus little-endian conversion or encoding complex types into a more human readable form such as XML for easier interoperability.

In this section we have described the core concepts of component based development and its implementation, as well as some of the benefits that arise from this approach. Building on this introduction the following section will examine how the CBD methodology can be applied to HLA, both at an abstract and detailed level.

4. Applying CBD Techniques to HLA

While HLA provides a strong foundation for building robust and accurate simulations there are a number of compelling reasons for creating a component-based framework to support the HLA development process.

In this respect the difficulties already outlined in HLA do not differ greatly from other software industries. For example, a comparison can be drawn between the problems of developing in HLA and those of developing in CORBA. In both environments the development effort tends to be ad-hoc in nature with no standard software framework to facilitate the design, generation, packaging, deployment and execution of a system in a repeatable manner [1]. The CORBA Component Model (CCM) was developed to address the same framework issues in CORBA. Similarly, the goals of this section are to

provide an overview of one possible component model for the development of HLA simulations that is appropriate for use by most simulation developers.

4.1 Creating an HLA Component Model

Following from the previous section there are a number of immediately recognizable similarities between the HLA simulation model and CCM. The most obvious being that between a Federate and a component, as both aim to provide an atomic service or object that publishes sufficient descriptive information to enable composable development of an application or simulation. In a Federate, this descriptive information is held in the Simulation Object Model (SOM).

Extending this comparison, the RTI is analogous to the component container, as it provides specialized simulation services to the Federates not necessarily provided by a generic component container. At a high level these services can be grouped into four areas: time management, federation management, ownership management and data distribution. Time management regulates the execution of the simulation and facilitates repeatability. Federation management provides an insight into the execution of a simulation. Ownership management provides a means of controlling the ownership value updates and data distribution provides a means of communicating data between Federates.

The implementation of these simulation services requires bi-directional communication between the RTI and the Federates. This is performed from the RTI to the Federate via the `FederateAmbassador` interface, and from the Federate to the RTI via the `RTIAmbassador` interface. This relationship is summarized in Figure 4.1.

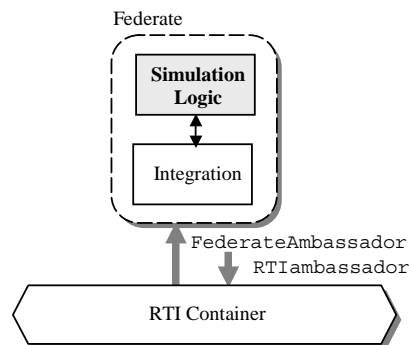


Figure 4.1 – HLA Component Architecture Overview

4.2 An HLA Component Model

From this initial comparison we elaborate on the more concrete definitions for creating standard component based development patterns for HLA. These definitions

take into account the key elements of an HLA simulation, namely the Federates, and their related Object and Interaction Classes. Once these elements have been defined we can explore how they work together.

The HLA component model is defined through the use of stereotypes [11]. The use of stereotypes provides a means of associating additional semantic meaning to the elements in a component model. Stereotypes also allow any constraints required by HLA to be applied to the generic CBD elements. We can apply three stereotypes to the base elements of HLA:

- a **Simulation Process** Stereotype is a component that relates to an HLA Federate;
- a **Simulation Entity Stereotype** is a light-weight component that relates to an Object Class; and
- a **Simulation Event Stereotype** is a short lived entity which relates to an Interaction Class.

The following section details the stereotypes as they apply to the DMSO “Hello World” implementation [9].

Simulation Process Stereotype

The Simulation Process stereotype is associated with the component and represents a Federate. A Simulation Process typically does not have a state that is visible to other components and can only communicate with each other via Interaction and Object Classes which are passed via the RTI.

Figure 4.2.1 illustrates a UML Class Diagram [10] of a “Hello World” Simulation Process component implementation. Figure 4.2.1 illustrates four groups of classes that make up the Simulation Process component: the simulation logic classes (HelloWorld, HelloWorldHome and SimProcess), the RTI service abstraction interfaces (TimeService and EventService), the auto-generated integration implementation (HelloWorldRTI and HelloWorldFederate) and the standard HLA interfaces (RTIAmbassador and FederateAmbassador [14]).

The Simulation Process implementation is comprised of the component proper (HelloWorld) and the component home (HelloWorldHome). The component contains the simulation logic and uses the RTI service abstraction interfaces, such as TimeService. The component home provides a Factory Method [5] for creating instance of a Simulation Process component. However, a Simulation Process instance cannot be removed by the component home, as the instance must resign itself from the simulation.

The overall implementation size of the Simulation Process is greatly reduced, as the code required to integrate with the RTI can now be automatically generated [11]. This results in a number of additional benefits. Firstly, the simulation logic (HelloWorld) is separated from the integration code (HelloWorldRTI and HelloWorldFederate), insulating it from any possible changes in the RTIAmbassador and FederateAmbassador interfaces, as well as any changes to the semantics of HLA itself. Secondly, the simplification of the interfaces and the implementation realised through abstraction and ‘separation of concerns’ also helps to reduce future maintenance costs and improves the chance of reuse in other simulations.

Requests from the RTI are translated into requests upon the component via the SimProcess interface. The design goals for the SimProcess interface are to provide only the essential methods that the majority of Federates require. Additional interfaces can be introduced for additional services as required. These interfaces help describe the extended services a component requires.

The TimeService and EventService interfaces are example abstractions of the standard services provided by the RTI. The RTI provides two groups of services; those that are used to statically configure the RTI and those that change the behaviour of the simulation when modified.

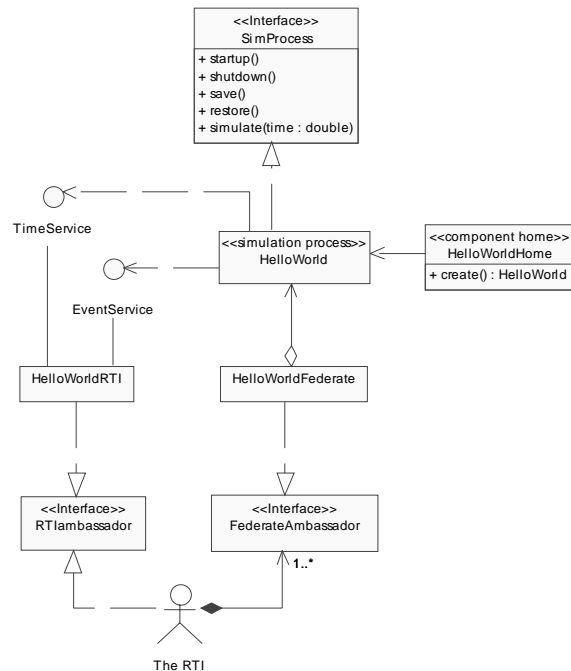


Figure 4.2.1—Simulation Process

The static configuration services include initialisation and cleanup of the RTI, the Federation, Federates and the FOM. The initialisation required by a component is specified in the component descriptors; as a result the services can be automatically generated.

The remainder of services change the behaviour of an executing simulation. These include time management, synchronization points, transferral of data (Interaction and Object Classes), and notification of changes in the simulation state. It is these services that can be logically grouped into the service abstractions to be used by the Simulation Processes.

The `HelloWorldFederate` provides the glue between the RTI and the Federate. This interface is used to configure the runtime settings of the Federate such as the timing settings (regulating and constrained), and to translate requests from the RTI that affect the simulation logic (such as time advancement or receiving Interactions).

The runtime settings reside in the component descriptor and are decided at design time. After the component is packaged, deployed and executed the generated integration implementation queries the component descriptor to configure the runtime behaviour of the Simulation Process.

The `HelloWorldRTI` provides the glue from the Federate to the RTI; its role is to provide the implementation of the abstract service interfaces used by the Simulation Process. The dependency of a Simulation Process on the service abstraction interfaces removes the need to directly configure the RTI, resulting in a saving of development time.

The use of service abstract interfaces provides a means of altering the runtime behaviour without modification to the simulation code. For example, consider an interface

`TimeService` with a single method `requestTimeAdvance` for advancing time. This method is used by `HelloWorld` to request that the simulation time be advanced to a specified time in the future, instead of using the equivalent methods (`timeAdvanceRequest` or `nextEventRequest`) provided by the RTI. The property that describes the time request policy is stored in the *component descriptor*, which is set at design-time. It is now the component descriptor that is used at runtime to determine which of the time advance protocols is applied.

The Simulation Entity Stereotype

The Simulation Entity stereotype is associated with the component that represents an Object Class. Figure 4.2.2 illustrates a UML Class Diagram [10] for the Country Simulation Entity component in the “Hello World” example. A Simulation Entity is a lightweight component; however it cannot be used without an associated Simulation Process component, as all Object Class calls from the RTI are first handled by the Simulation Process and then despatched to the appropriate Simulation Entity.

The implementation of a Simulation Entity can be automatically generated from the component interface, which is derived directly from the Object Class definition. The generated implementation has three parts: the component (`Country`), the component home (`CountryHome`) and a callback interface (`CountryCallback`).

The component stores the attribute values internally; these can be modified via the generated accessors (`getPopulation`) and mutators (`setPopulation`). These accessors and mutators can be extended to provide a Design By Contract [12] mechanism to enforce strong typing based on the semantics defined for each attribute in the component interface.

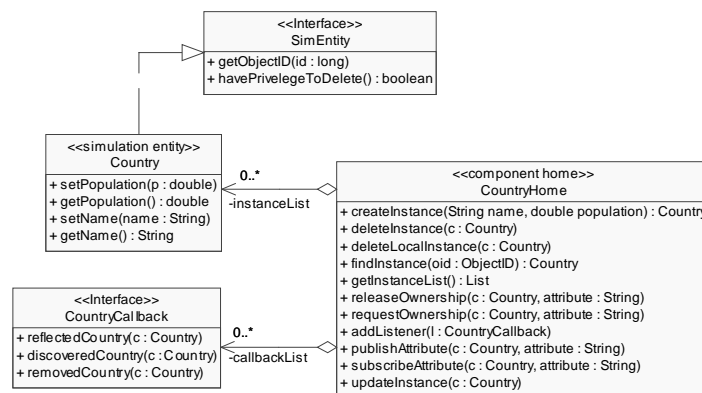


Figure 4.2.2 – Simulation Entity

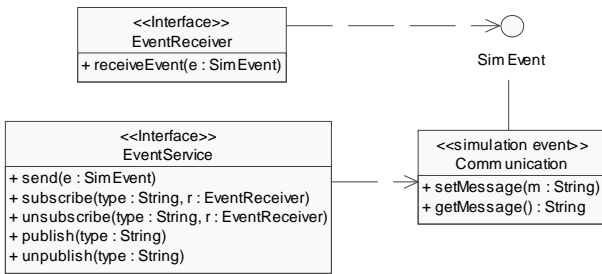


Figure 4.2.3 – Simulation Event

The component home provides a toolkit for managing the full lifecycle of a Simulation Entity component. When a component is created, it is registered with the RTI. As a result, other instances of the component home discover the new instance, and create a local representation. The component home also handles reflecting values and the removal of instances to maintain a synchronized collection of components that can be retrieved via `findInstance` and `getInstanceList` methods.

The callback interface is used to pass on changes in the state of a Simulation Entity back to an interested party. A class that implements the callback interface must first be registered with the appropriate component home before any feedback can be received. For example, when a `reflectAttributeValues` message is issued by the RTI, the `CountryHome` updates the corresponding `Country` instance values, and the new value is passed on to the registered listeners via `reflectCountry`. Similar steps are followed for discovery and removal of Simulation Entity instances.

The Simulation Event Stereotype

The Simulation Event stereotype is associated with the entity that represents an Interaction Class. A Simulation Event can only be associated with a Simulation Process, which handles the RTI Interaction Class calls. As shown in Figure 4.2.3 a Simulation Event is comprised of three parts: the data container (Communication), a callback interface (EventReceiver) and the common event service (EventService).

The data container stores the values that represent an Interaction. Like a Simulation Entity, a Simulation Event can be entirely generated from the component interface, which is derived from an Interaction Class definition. This results in the automatic generation of accessors and mutators for each parameter, including the code required to marshal and unmarshal the parameter values.

The event service is used to publish and subscribe to Simulation Event instances. To subscribe to a particular event a Simulation Process must specify the event

identifier and a class that implements the callback interface. The callback interface is invoked when an external Simulation Process publishes an instance of a Simulation Event.

Pulling It Together

This section assembles the design patterns presented in the previous sections and illustrates the remaining development stages of the “Hello World” component.

Given these design patterns for the HLA component model, it is possible to generate a skeleton implementation. The skeleton implementation is generated from the component interface description and provides essential hooks that allow developers to implement the simulation logic.

Figure 4.2.4 illustrates the implementation of the “Hello World” Simulation Process component. The `HelloWorld` class implements `SimProcess`, `CountryListener` and `EventReceiver`. The `SimProcess` interface provides the base callback interface that allows the RTI to interact with the component. The Simulation Process receives notification of the “Country” Object Class instances via the `CountryListener` and of “Communication” Interaction Classes via `EventReceiver`. The interfaces provided and used by a Simulation Process component are similar to a Federate SOM, as they define how to communicate with the component.

The packaging process requires assembling all the required resources into a single unit. Simulation Entities and Events cannot be packaged independently of a Simulation Process. The Simulation Process package

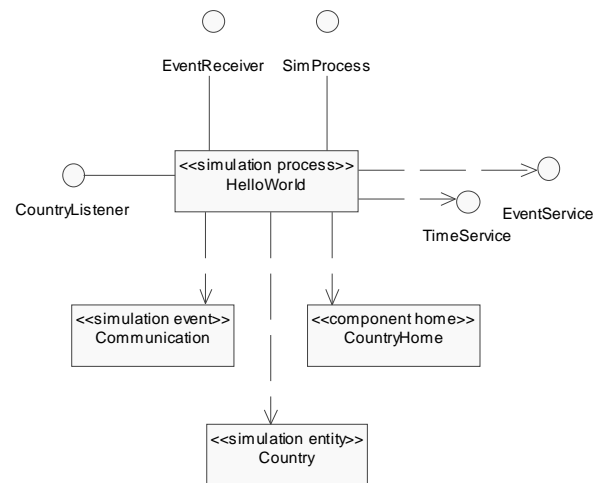


Figure 4.2.4 –Simulation Component with dependencies

contains all the resources for dependencies upon Simulation Entities and Events, including all the required implementation and specifications required by the Simulation Process to ensure it is a self-contained deployable unit. The packaging process also involves generation of the integration implementation for the RTI container; this process is transparent to the developer.

The assembled packages can then be deployed and executed on a desired host. The deployment and execution process can be assisted by an activation service, which provide the ability for the remote deployment and maintenance of Federates over a network; this removes the need for developers to manually start a simulation.

4.3 Addressing FOM Agility

In dealing with real world simulations it is not uncommon for the FOM to be changed after a simulation component has been implemented, or if a third party simulation component is not compatible with the local FOM. This problem is a result of low FOM agility. The HLA component model addresses this problem through transformations performed on the FOM data.

The mappings from FOM elements (Object and Interaction Classes) to HLA component model elements (Simulation Entities and Events) are specified in the component descriptor. If the FOM elements are not compatible with the component model elements, the generated integration implementation (`HelloWorldRTI` and `HelloWorldFederate`) can handle the transformation of data. For example, if the Simulation Entity `Country` were modified so that the “population” attribute of type `double` was changed to “popCount” of type `long`, the integration implementation would make the appropriate name mapping and type conversion between the component and the FOM.

The integration implementation that performs the transformations also provides the ability for complex data transformations through the application of Interceptors [13]. The Interceptor allows the developer to provide the implementation for a custom transformation that is triggered by the integration when data is transmitted through the RTI.

4.4 Benefits

Standard Framework

There are a number of clear advantages that arise from the introduction of a component model for the development of HLA simulations. Firstly, the framework significantly reduces the amount of development effort required to integrate Federates with the HLA infrastructure, thus insulating the developer from much of the “boilerplate” coding associated with integrating to the RTI API. In

addition, it promotes better simulation decomposition, ensuring that the business logic of a simulation is separated from integration concerns, freeing the developer to focus on the simulation logic rather than integration requirements.

Component Reuse

Improved reuse also follows from this approach, as the simulation logic code is no longer incorporated or intertwined with any integration code. In addition, any FOM-Agility and data transformations issues can be managed externally in the integration components. For example, in the previous example, the `HelloWorld` component contains all business logic, while the `HelloWorldRTI` and `HelloWorldFederate` components manage the integration and any FOM-Agility issues between the now isolated simulation logic and the RTI.

These reuse characteristics also introduce the possibility for the development of a ‘commercially of the shelf’ market place for HLA development, as simulations and ancillary services (such as logging and visualisation) can more easily be assembled from pre-existing parts.

It is worth emphasizing at this point that the component model outlined in this paper is not introducing a new or proprietary API to the HLA space; rather it uses design patterns and abstractions to better utilize the services provided by the RTI. This avoids any vendor lock-in issues. Whilst code generation tools could greatly simplify the design and coding process, there is nothing preventing a developer from creating all the required elements for a component model by hand; this would incur additional development costs and may not be done correctly or fully.

Tools Support

The component model encourages the use of automation tools for performing repetitive development tasks, increasing productivity and overall development consistency and quality. There are three key areas where tools can be applied to the CBD development process; simulation design, code generation, and component packaging /deployment [11].

The simulation design process can be improved through the introduction of GUI tools. These design tools provide a means of visualizing the discrete components in a simulation, the interactions between these components and the simulation as a whole. Such design tools provide a means to visualize the often-complex relationships that exist between components in an accessible and manageable way, greatly increasing overall understanding.

One of the most obvious and compelling applications of tools support is in the area of code generation, both in respect to the actual simulation code and its integration requirements. By providing design tools which allows the developer to describe the interfaces of the components - which is conceptually the same in HLA as designating the Object and Interaction Classes a Federate is going to publish and subscribe to in the FOM - it is possible to automatically generate the integration code and the simulation logic skeleton implementation. The generated skeleton implementation remains indifferent to the eventual middleware infrastructure in which it will be deployed, although specialized container services can be utilized through the generation process (such as region support in HLA). Once generated, all that remains is for the developer to provide the necessary functionality into the skeleton implementation.

Code generation alleviates one of the main criticisms made of HLA, namely the complexity and repetitive nature of developing RTI integration code for the Federates, as all but the simulation logic can now be defined and generated through the tools. Through the consistent use of templates and design patterns the code generation process also improves the resulting component's stability, repeatability and overall quality.

Component packaging tools allow the consolidation of the various component elements into a single deployable unit or file. Once packaged a component's physical location can be assigned through the use of deployment diagrams [10], which in turn can be executed across a network by a deployment management service. This deployment management service can also provide simulation management, for example synchronizing, starting and stopping a simulation. Additionally, real-time performance metrics, such as system state and load, and services, such as load-balancing and fail-over recovery, can be presented back to the developer in a visual tool.

5. Conclusion

This paper describes one approach for the delivery of a component-based framework for the development of HLA based simulations. This approach borrows heavily from many of the ideas and concepts encapsulated in the CORBA Component Model. It was shown that HLA fits comfortably into a component-based architecture through the use of three design stereotypes. A number of the core advantages of this approach have been discussed, particularly in the areas of component reuse and simulation composition.

Component-based development, in conjunction with the support of development tools, promises to significantly reduce the complexity and costs currently inherent to

HLA development, while encouraging and supporting component reuse and FOM-agility. This has the potential to ultimately lead to cheaper, more reliable and finer grained composable HLA simulation development.

6. References

- [1] Heineman, G., Councill, W: "Component-Based Software Engineering", Addison Wesley, 2001
- [2] Syzperski, C: "Component Software – Beyond Object Oriented Programming", Addison-Wesley, 1998
- [3] Macannuco, D., Coffin, D., Dufault, B., Civinskas, W: "Experiences with a FOM Agile Federate", Spring Simulation Interoperability Workshop, 1999
- [4] ComponentSource: "Enterprise Reuse", <http://www.componentsource.com>
- [5] Gamma, E., et al: "Elements of Reusable Object-Oriented Software", Addison Wesley, 2000.
- [6] Object Management Group: "Update CCM Specification", <http://www.omg.org>, November 2001
- [7] Object Management Group: "Event Service Specification", <http://www.omg.org>, March 2001
- [8] Kassem, N., et al: "Designing Enterprise Applications with the Java™ 2 Platform, Enterprise Edition", Addison Wesley, 2000
- [9] Centre for Object Technology: "How to become an HLA guru in a short(er) time", 1998
- [10] Booch, G., Rumbaugh, J., Jacobson, I.: "The Unified Modeling Language User Guide", Addison Wesley, 1999
- [11] Calytrix Technologies Pty Ltd: "A Visual Tool to Simplify the Building of Distributed Simulations using HLA", www.calytrix.com, November 2001.
- [12] Meyer, B: "Object-Oriented Software Construction, Second Edition", Prentice Hall, 1997
- [13] Schmidt, D., et al: "Pattern-Oriented Software Architecture, Volume 2", Wiley, 2000.
- [14] Defense Modelling and Simulation Office: "IDL application programmer's interface" <http://www.dmsomil>, April 1998

Author Biographies

SHAWN PARR is the co-founder and Chief Technology Office at Calytrix Technologies, an Australian based research and Development Company specializing in component-based solutions and HLA simulation. Shawn has been working in the IT industry for over 10 years and holds a Bachelor of Science degree and a research based Masters degree.

ALEX RADESKI is a Senior Software Engineer at Calytrix Technologies and has a keen interest in Design Patterns, Component Based Development and real-time visual simulation. Alex has been working as a software engineer for over 7 years and holds a Bachelor of Science in computing.

RUSSELL KEITH-MAGEE is a Software Engineer at Calytrix Technologies with a research background in both Physics and Computing. He holds a Bachelor of Science in Physics, and a Bachelor of Science with Honours in Computer Science; his Ph.D. thesis is currently undergoing assessment.

DR JOHN WHARINGTON is a research scientist with the Maritime Platforms Division (Melbourne) of the Australian Department of Defense. He has worked on various HLA development tools since 1999, and is also working on developing distributed simulation components to support autonomous underwater vehicles. His research background in Aerospace Engineering and Artificial Intelligence has given him extensive experience in both applied simulation and modelling and in the development of simulation and visualisation codes. John has a PhD and a BEng (Hons 1) in Aerospace Engineering from the Royal Melbourne Institute of Technology.