

VISUALISING DISTRIBUTED SIMULATION DESIGN AND DEPLOYMENT

**Dr Russell Keith-Magee, Shawn Parr, Alex Radeski
Calytrix Technologies Pty Ltd
Perth, Western Australia**

ABSTRACT

The “Hello World” code provided as a sample application for the High Level Architecture (HLA) framework comprises over 2500 lines of code, of which less than 50 provide any real simulation logic. This coding and integration overhead represents a considerable development effort. This overhead, and the lack of a standardized design pattern for separating simulation logic from integration code, often results in high levels of code coupling. This in turn leads to poor code structure and design, poor reuse, and increased maintenance costs.

Significant improvements can be made in the design, development and deployment of HLA based simulations through the application of a Component-Based Development (CBD) approach and the use of visual design metaphors. These improvements are realized through the use of CBD patterns, which provide a mechanism to clearly separate the simulation logic from the integration requirements of the federate. Using this model, simulation logic remains separated from the integration code. Generated integration logic can then be used to manage the physical integration with the RTI, and any other required services, such as data transformations. The simulation logic and the integration logic collectively form the HLA federate.

This separation also allows the developer to visually model the relationships between federates, allowing the creation of a simulation without the need to consider integration requirements. This visual design approach can be used to encapsulate the simulation workflow, the asynchronous ‘publish and subscribe’ relationships between the components and the FOM, the synchronous inter-component relationships (method calls between components outside of the RTI), any data transformation required to insure interoperability, and the deployment of the simulation.

This paper examines a number of visualizations that can be used to significantly aid in the development and deployment of distributed simulations. It will also examine how a simulation can be directly generated, deployed and executed from the visual model.

AUTHORS’ BIOGRAPHIES

Dr Russell Keith-Magee is a Software Engineer at Calytrix Technologies with a research background in both Physics and Computing. His doctoral thesis was on biologically motivated models of machine learning and development. He also holds a Bachelor of Science with Honours in Computer Science and a Bachelor of Science in Physics.

Shawn Parr is the co-founder and Chief Technology Officer 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. He holds a research-based Masters degree, and a Bachelor of Science.

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.

VISUALISING DISTRIBUTED SIMULATION DESIGN AND DEPLOYMENT

Dr Russell Keith-Magee, Shawn Parr, Alex Radeski
Calytrix Technologies Pty Ltd
Perth, Western Australia

INTRODUCTION

Despite the significant development effort associated with the creation of a HLA compliant federate, there is currently no standardised development approach to federate design. Consequently, federates tend to be developed in an *ad hoc* fashion, resulting in code that is difficult to reuse between federates, and federates that are difficult to reuse between federations. This conflicts with the stated design goal of HLA: to encourage and support the reuse and interoperability of simulation federates.

It has been shown in other industries that by adopting a common development framework, it is possible to dramatically improve the reuse and maintainability of software. It has also been shown that given the existence of a common development framework, it is possible to use tools to assist in the development of complex systems (Parr, *et al.*, 2002; Calytrix Technologies, 2001).

The HLA Component Architecture (Radeski, *et al.* 2002) is one candidate for a standardised development framework for HLA. This development framework provides a clear separation between integration logic and simulation logic. Furthermore, it provides a structure for the integration logic that maximizes the flexibility and potential for reuse of the encapsulated simulation logic.

This paper provides an overview of the HLA Component Architecture, and discusses the ways in which this architecture can be used to facilitate the use of tools for federate design, code generation and federation management. This use of tools is dependent upon the ability to cleanly characterize the design of a federate and federation. This paper will discuss how a component-based approach to federate development enables a design to be decomposed into a relatively simple specification. Finally, a range of visualisation techniques will be explored. These techniques enable the developer to use visual tools in the design and management process.

SIMULATION DEVELOPMENT MODELS

While HLA describes a feature-rich middleware layer for the development of time-regulated distributed

simulations, it does not provide a design methodology or model for the creation of federates. As a result, the development of federates tends to be performed on an *ad hoc* basis, adversely affecting simulation logic reuse and increasing development and maintenance costs.

This observation is not unique to HLA development: neither CORBA nor RMI include design methodologies as a part of their specification. The CORBA Component Model (CCM) (Object Management Group, 2001) and Enterprise Java Beans (EJB) (Kassem, *et al.*, 2000) were created to address the design issues associated with the middleware layer of component-based systems. In general terms these frameworks sit above the middleware or service layer and present a component based development approach to aid in the control of development and maintenance overhead.

A Component-Based Design Model

Component-based development (CBD) describes a methodology where a system is broken into self-describing modules (Components) that describe their services (the underlying implementation) in the form of an interface specification. The goal of CBD is to increase reuse while reducing development and maintenance costs (Syzperski 1998; Gamma, *et al.*, 2000; Heinemann and Council, 2001). CBD also strives to maintain a separation between integration and business logic concerns, thereby preventing the underlying middleware or integration requirements polluting the business logic. It is important to note that CBD is not dependant on object-oriented programming languages and in general remains implementation neutral, providing the component remains self contained and meets its interface contract obligations.

The design goals and philosophy of CBD share many similarities with the goals of HLA federate development. However, HLA currently has no standardized development approach analogous to CCM or EJB. As a result, there is no commonality in design and implementation between federates. This increases the development and maintenance costs associated with each federate in a federation, and impacts on the potential for reuse outside of the FOM for which the federate was designed.

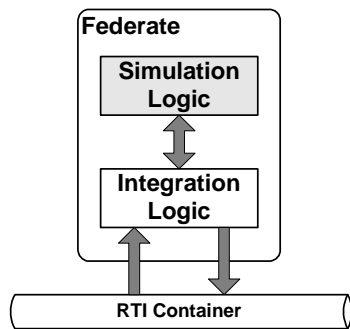


Figure 1: Federate separation in the Component-Based Extensions to HLA.

The HLA Component Architecture (Radeski, *et al.*, 2002) is one candidate for the definition of a standardized framework for simulation development. This architecture employs Component-Based Development techniques, providing a clear federate development model and describing how the separation of integration logic and simulation logic can be achieved. This development model also provides a commonality of design between federates, allowing them to access HLA and extended CBD services (such as data transformation services between federates and the FOM) in a consistent manner. Furthermore, it provides a structure for the integration logic that maximizes the flexibility and potential for reuse of the simulation logic. Many of the concepts proposed in the HLA Component Architecture are drawn from existing industry-standard initiatives such as CCM and EJB.

A Component Model for HLA

A schematic of the HLA Component Architecture can be seen in Figure 1. In this architecture, a Federate is internally separated into *Simulation Logic* and *Integration Logic*. The Integration Logic manages the interactions between the simulation logic and the RTI, providing an RTI independent API to the simulation code, and a simulation independent API to the RTI services. This layer of abstraction between the RTI and the actual simulation logic not only simplifies the development effort but also allows the integration code to be regenerated for various middleware layers (or new versions of the RTI) without impacting upon the simulation logic.

The simulation logic encompasses the algorithmic description of the process under simulation; any simulation related services required by the Simulation Logic are accessed via the Integration Logic, rather than through direct interaction with the RTI (although direct RTI access can remain an option). The use of these abstractions ensures that the Federate remains

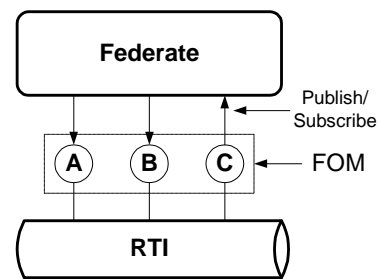


Figure 2. A traditionally developed federate, tightly coupled to the FOM and RTI.

decoupled from a specific RTI implementation. The Federate is then deployed in an RTI container.

Advantages of the CBD Approach

The separation of Simulation Logic from Integration Logic has two key advantages. Firstly, the separated design improves code readability. Although more code is usually required to implement the separated framework, the resultant code is easier to comprehend as the complexity associated with RTI integration is removed from Simulation Logic. Code that is easier to read is easier to maintain; as a result, there is a lower engineering effort associated with developing and maintaining a CBD federate.

The second advantage of the CBD approach is an improvement in the potential for federate reuse. Using traditional HLA development techniques, the FOM of a federation must be the superset of all the participating federate SOMs. Under this model individual federates are written to adhere to a SOM, and the federate must be deployed into a federation with a compatible FOM. In such federates, Simulation Logic interacts directly with the RTI via the SOM and as a result the federate's code is tightly coupled to its specific SOM. If a federation does not provide a compatible FOM the simulation logic must be rewritten. This arrangement is shown in Figure 2. In this example, the Federate publishes A and B, and subscribes to C. This federate can only be deployed into a simulation whose FOM contains A, B and C.

As a result of this approach, the development effort tends to become very FOM-centric, with undue pressure being placed upon the correctness of the upfront FOM specification. This directly impacts upon the capacity for federate reuse outside of the original FOM specification, thereby conflicting with the core goals of HLA (i.e., reuse and interoperability). However, this limitation is not a direct artefact of the RTI, but rather a result of an inconsistent and unstructured approach to federate design.

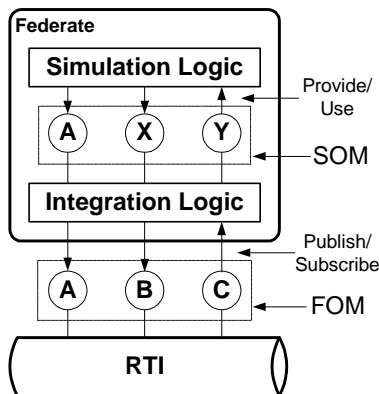


Figure 3. A Component-Based federate, with loose coupling to the simulation's FOM and tight coupling to the component's own SOM.

One of the fundamental advantages of the HLA Component Model is that the SOM utilized by the Simulation Logic need not be tightly coupled to the FOM exposed by the RTI. By introducing an intermediate Integration Logic layer within the component architecture (as shown in Figure 1), the CBD model encourages a distinction between the data classes (i.e., the Object and Interaction classes) that are *provided* by the simulation logic of a federate, and the data classes that are *published* to the RTI. Similarly, there is a distinction between the data classes that are *used* by the Simulation Logic of a federate and the data classes to which a federate *subscribes*. In this model, the Simulation Logic of a federate is tightly coupled to a SOM that defines its set of provided and used classes, which are then published (and if necessary, transformed) through the integration logic layer to the FOM used by the federation. This arrangement is shown in Figure 3. In this example, the Federate provides A and X, and uses Y, but publishes A and B, and subscribes to C. The Integration Logic manages the X|B and C|Y transformations.

Under this arrangement, the binding to the FOM is loose and can be easily rebuilt given a definition of the transformations required between the SOM and FOM definitions. Changes can then be made to the FOM without affecting Simulation Logic and vice versa.

In the simple case where there are no differences between the SOM and FOM (for example, in Figure 3 the federate provides A that is then published as A to the RTI), there is no transformation required through the intermediate integration layer. The transformation logic in this case becomes a simple "pass through" mechanism. In optimised code, the physical

transformation layer could be omitted, although the conceptual intermediate layer would still exist.

The power in this abstraction comes when the need arises for a federate to be reused in a different FOM, which is semantically equivalent but syntactically distinct. Having developed a federate with a given set of provided and used classes, deployment of the federate in an incompatible FOM requires only the generation or regeneration of integration code to map between the two. No simulation logic of the federate need be altered; all the changes exist in integration code.

However, the implementation of a component based federate does introduce an additional coding overhead, as the integration code layer requires the writing of code that would not be required for a non-componentized federate. This is especially the case when there are no differences between the SOM and FOM definitions. In the following section we will see how this concern and other design requirements can be directly addressed through the use of tools.

APPLICATION OF CBD TOOLS

While a component-based approach to development delivers a number of benefits to the developer, including an approachable and repeatable integration model, it does not address the overhead and complexity associated with creating HLA compliant federates. In fact, the CBD approach has the potential to increase the integration requirement within the federate, especially when there are no differences between the SOM and FOM definitions.

However, this increased code requirement need not correspond to an increased development effort. Another significant advantage to the component-based federate architecture is the fact that *all* of the integration code required by a federate can be automatically generated from a description of the federate and a set of templates. This description can also be used to generate skeleton or stub code for the body of the simulation logic, leaving only the development of the actual simulation algorithm to the developer.

The use of tools in the federate development process also has the benefit of causing a major shift in engineering focus. Using traditional development techniques, a significant portion of the development effort is spent on the integration task. However, using tools and a CBD approach to federate development, no engineering effort need be expended in developing integration logic; engineers can therefore concentrate on developing and maintaining simulation logic.

Describing a HLA Simulation

Although the integration logic for a federate can be generated from templates, this process requires the definition of the basic characteristics (the *specification*) of the federate and its relationship within the federation. The template code provides a structural framework describing the objects that will exist within a federate, and defines any boilerplate code required to allow these objects to interact with the RTI; however, any federate specific details must be derived from a federate specification.

The description required for code generation must specify a number of key characteristics of a federate:

- **Composition:** Basic structural information defining the Attributes of each Object Class, the Parameters of each Interaction, the Dimensions of each Routing Space, and the definition of the SOM of each Federate.
- **Inheritance:** The inheritance hierarchy of Object and Interaction Classes in a FOM.
- **Interaction:** The definition of the FOM, and the relationship between the SOM of a Federate and that FOM.
- **Conversion:** The transformations required to convert SOM data classes into FOM data classes.
- **Assignment:** The deployment of federates on hosts.

These characteristics are sufficient to uniquely define the construction of a federate, the interaction of the federate with the RTI, and the distribution of the federation over a network. By abstracting the template definition to these descriptors it is possible for code templates to remain application independent; once written, they can be used for any federate in any federation.

This approach to tool based code generation is not without precedent. For example, it shares many similarities with the way in which IDL compilers are used to generate stub and skeleton code for CORBA applications. These core federate characteristics could also be defined using a simple grammar, with code generated from definitions built in this grammar. However, these characteristics are also a good candidate for design by visual tools as they contain parallel hierarchies (for example, composition and inheritance are both natural hierarchies, but they conflict in structure) and interconnected relationships between these hierarchies.

In the remainder of this paper, a range of visualisation techniques will be presented which enable the

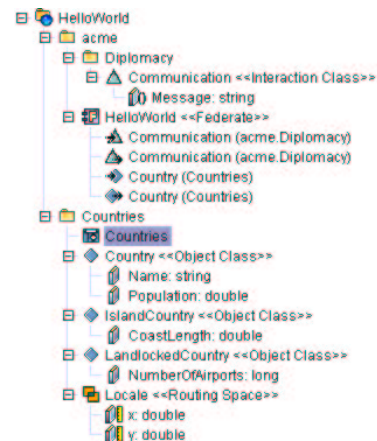


Figure 4. A Tree View of the structural hierarchy of a federate.

characteristics defined in this section to be designed visually.

Composition: Tree View

The fundamental hierarchical relationship in the system is the composition relationship between Object Classes and their Attributes, Interaction Classes and their Parameters, and Routing Spaces and their Dimensions. In addition, the composition of the SOM of each Federate can be defined in a hierarchical fashion; used and provided object and interaction classes can be represented as children of the federate that uses or provides them.

The natural visualisation for a hierarchical structure is a tree, as shown in Figure 4. Each child is shown as a leaf of its parent; these parents can themselves be leaves, and so on to the root of the tree. Icons can be used to distinguish the different types of children present on the tree, especially where a parent has different types of children (e.g., the used and provided data classes on a Federate). The familiar tree widget allows for collapsing of these parent nodes, enabling easy browsing of a complex specification.

The tree also encourages the use of namespaces in the construction of complex systems. Although HLA has no concept of namespaces, they can be used as an organisational unit during design, and in code, without affecting the operating characteristics of a federate. Namespaces can be easily represented as folders or packages within the tree hierarchy, providing a mechanism to logically group elements of a complex specification.

It is interesting to note that the tree view of a simulation model mirrors the structure that one would expect to observe in the indenting style of an IDL-style grammar.

In an IDL style grammar, each package namespace would involve a level of indenting; all children of a parent would be indented one level deeper than the parent. This supports the idea that the composition of the federate is the fundamental hierarchy of the federation design, which should therefore be represented in a simple hierarchical fashion.

Inheritance: Class View

The hierarchical nature of the Tree View is good for displaying the composition of individual data classes. However, the structural relationships described in the previous section are not the only hierarchy in the simulation model. The inheritance relationships between object and interaction classes are another natural hierarchy in the federation model.

It is possible to represent the inheritance relationships present in the simulation model on the same tree as the structural relationships; node labels on the tree can be extended to show the names of superclasses, and inherited attributes can be added to the structural tree alongside aggregated attributes. However, while a tree is well suited to displaying a single hierarchy, it is poorly suited to the representation of multiple conflicting hierarchies, especially when complex inheritance relationships are involved (such as occurs in the RPR-FOM). If a single tree were to show the inheritance of attributes, the tree would rapidly become cluttered with the duplicated attributes of parent classes. Furthermore, while the relabelling of tree nodes is able to display superclass information (for example, the inheritance relationship for `IslandCountry` in Figure 4 could be represented on the tree node as `IslandCountry «Object Class» (from Country)`), this approach fails to effectively visualise inheritance relationships. As a result, complex inheritance relationships are not obvious in a tree structure.

The structural tree is therefore best augmented using a visualisation that is well suited to handling inheritance relationships. UML provides such a visualisation in class diagrams (Rumbaugh *et al.*, 1999). Class diagrams are designed to visualise the complex relationships between objects, interfaces, and instances in an Object-Oriented or Component-Based system, and to allow the easy alteration of these relationships. In UML, a Class is represented using a square with the name of the class in it. This square can be subdivided to provide attributes and operations of the class. Relationships between classes are represented using arrows. Inheritance is represented using an arrow with a solid shaft and empty arrowhead at one end, originating from the child class.

Using these visual elements it is possible to visualise and design complex inheritance structures.

Although there is no perfect correspondence between Object Oriented and HLA nomenclature, UML notation can be easily adapted for designing HLA structures. The UML concept of a Class is analogous to Object Classes and Interaction Classes in HLA; UML notation can therefore be used to represent HLA structures. This mapping of HLA into UML is made easier by the fact that UML provides a notation for expanding UML into new domains. *Stereotypes*, annotated by the use of chevrons (« ») can be used to extend the semantics of UML diagrams into the HLA domain. «Object Class» and «Interaction Class» stereotypes can therefore be used to annotate UML Classes; inheritance relationships can then be shown in the normal UML fashion. Alternatively, color could be used to represent classes with different stereotypes.

UML Class Diagrams are not limited to the design of inheritance relationships. Dependencies between classes can be shown using arrows with a dashed shaft and an open arrowhead at one end, originating at the dependent class. Aggregations of classes are represented using a solid line between classes, with cardinality of the aggregation indicated on the endpoints of the line. Relationships of this sort also exist within HLA. Dependencies can be used to document the expected communications between component interfaces. Federates map well to the UML notation of a Component, represented using a rectangle with two nested rectangles on the left hand edge. The aggregation of federates by other federates could be shown on a Class diagram. In a sophisticated visualisation, the use and provide relationships of the SOM of a federate could also be shown using aggregation notation.

Figure 5 shows a class diagram using this notation. In this diagram, the Object classes `IslandCountry` and `LandlockedCountry` inherit from `Country`; `Communication` is an Interaction Class. `Federate` provides `IslandCountry`, and uses `Communication` and `LandlockedCountry`. Attributes and Operations are shown for `Country`; these are hidden for `IslandCountry`, `LandlockedCountry`, and `Communication`.

Interaction: Publication and Subscription

The Tree and Class Views enable the user to visualise the structural characteristics of Object and Interaction Classes and the SOM of a Federate. In order to participate in a federation, the relationship between the federate's SOM and the federation's FOM must be

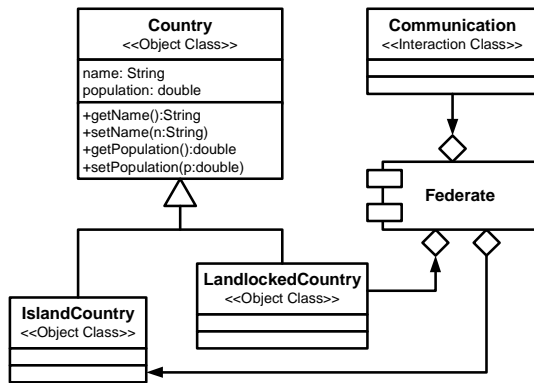


Figure 5. A UML Class diagram for a federate, with HLA stereotypes.

defined. This relationship is used in the code generation process to define the publication and subscription of Object and Interaction classes and any transformations required between the integration code and the RTI during execution.

The FOM elements used by a federate could be defined using the same mechanism as the SOM; that is, as a structural characteristic of a federate displayed in a tree. In this way, FOM elements would be added as children of the federate. However, this approach blurs the distinction between the SOM, which must be closely coupled to the simulation logic of the federate, and the FOM, which is weakly coupled to the federate and whose definition is shared amongst all federates. In addition, the composition approach fails to visualise cross-federate communication. Although the HLA model is based upon broadcast communications (i.e., direct federate-to-federate communication does not occur), the definition of a SOM implies communication with other federates. As a result, the publication of a data class by one federate implies that some other federate will subscribe to the same data class. A cross section of FOM usage across the federation is therefore required to ensure parity between publication and subscription. Again, this cross-federate communication is not easily visualised using a tree structure.

An alternate visualisation mode can be developed by considering the design task to be the definition of an intersection between the SOM and the FOM. These intersections can be easily visualised as a two dimensional (2D) grid. On the vertical axis of the grid, the FOM elements (or a subset of these elements) are displayed; these include any Object or Interaction Classes in the FOM with which the federate may wish to interact. These elements are candidates for publication and subscription. On the horizontal axis of the grid, the composition of the Federate is displayed;

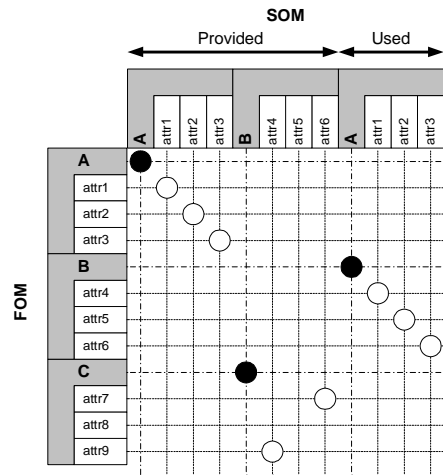


Figure 6. A fully exposed visualisation of the intersection between the SOM and the FOM of a federate.

any Object or Interaction Class in the SOM (i.e., those which are provided or used by the federate) is listed.

The intersections within this 2D space represent the communication between the federate's SOM and the federation's FOM. A set of points marked on this space defines the existence of a communication channel between the SOM and the FOM. The horizontal coordinate of an intersection point indicates the participant federate's SOM element; the vertical coordinate indicates the corresponding FOM element. If the FOM and SOM data elements indicated by the intersection are different, a transformation will be required. Not all potential intersections on the 2D map are valid; intersections between incompatible entities must be restricted. For example, Object Classes cannot intersect with Interaction Classes and vice versa. An example of this visualisation is shown in Figure 6.

Within this representation, the publication and subscription of object classes is performed at the attribute level; therefore, it must be possible to show intersections at the level of individual attributes. However, the publication/subscription of Interaction Classes is performed on a per class basis; there is therefore no need to show communication at the parameter level.

However, while this visualisation is adequate for representing the relationships between a single federate's SOM and the FOM, it is subject to some difficulties. Firstly, unless there is extensive repeat publication and subscription, the intersection map will be extremely sparse; most of the map will be blank. Therefore, a full 2D grid is wasteful as a visualisation. Secondly, because of the real estate required to display

all the SOM elements, it is difficult to represent multiple federates on a single canvas.

A vastly improved visualisation can be achieved by collapsing the horizontal axis of the 2D map. In the full 2D map, the horizontal axis is somewhat redundant, as the composition of the SOM is already shown on the Tree View (see Figure 4). In addition, the real purpose of this visualisation is to capture the relationships between the federates and the FOM. By collapsing the horizontal axis (the Federate's SOM elements), FOM composition is the only presented information. The set of intersection points on a vertical line now shows the utilisation of a FOM element by a federate; the set of intersection points on a horizontal line shows all federates and their involvement with a particular element of the FOM. This approach also requires less visual real estate, allowing multiple federates to be displayed concurrently. The collapsed view is therefore able to represent more relevant information in less space than the full 2D map.

This approach does require two additional features. Firstly, a method is required for differentiating between publication and subscription. On the full 2D map, provided and used entities are shown in their entirety; the horizontal display of the SOM includes details of the provided/used status of each interface. However, when collapsed, there is no distinction between publication and subscription; a single intersection point is provided. This limitation can be overcome by providing two intersection points: one for publication, and one for subscription. Secondly, a method is required for visually differentiating between pass through connections (SOM equivalent to FOM) and transformed connections (SOM different to FOM). This can be achieved through the use of colour on the intersection points to highlight transformed connections. An example of this collapsed visualisation is shown in Figure 7.

Routing Spaces

Although not immediately obvious, the use of Routing Spaces (or Data Distribution Management) by a federate can be handled using the same visualisation technique. When the user publishes an object class they indicate an intention to create instances of that class in the federation. Similarly, when a user publishes a Routing Space, they indicate an intention to create regions in that routing space. The "published" routing space need not correspond to the routing space used internally by the federate; the externally presented routing space can be abstracted in the same way as data classes. As a result, the publication/subscription

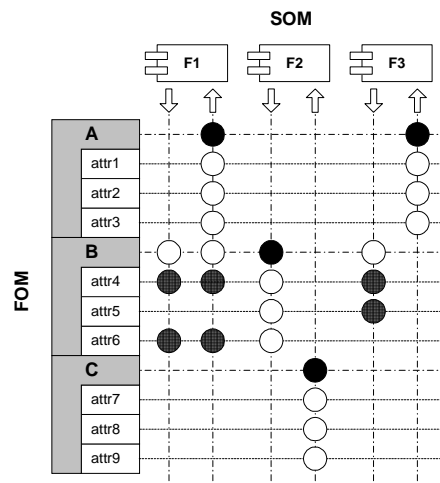


Figure 7. A collapsed representation of the intersection between the SOM (publication on left column, subscription on right column) of a federate, and the FOM.

visualisation can be used to show the use of Routing Spaces by a federate.

However, unlike data classes, Routing Spaces cannot be "subscribed". Federates can create Regions, but they do not obtain references to other regions in the federation. This restriction can be accommodated by not displaying subscription points for any Routing Space on the vertical axis.

Conversion: Transformation View

The publication/subscription view allows the user to describe FOM interactions for a federate, and visualise cross-federate communications. This visualisation also captures the need for any transformations between the SOM and FOM. However, it does not describe the nature or form of any required transformation; only the existence of the transform is indicated. An additional visualisation is required to qualify the transformation.

This problem is not unique to SOM/FOM transformations; other domains exhibit analogous problems. The most notable example is the definition of XSLT (Extensible Stylesheet Language for Transformations) specifications for converting XML documents, often employed in B2B solutions such as Microsoft Biztalk. The most common form for tools of this type is to show a list of data sources down the left hand side of the view, and a list of data targets down the right hand side of the view, with lines connecting input sources and output targets. Figure 8 shows an example of this type of data conversion. In this example, the SOM contains classes A, B and C, and the

FOM contains classes X, Y and C. Class C is passed untransformed; attributes conversion is performed on the A|Y and B|X transform.

However, this approach is subject to a number of limitations. If the entities to be transformed are large, the crossings on these diagrams can potentially span large distances; it may not be possible to display the source and target at the same time. If a large number of attributes are involved in a transformation, a large number of crossing lines will be required. When combined with the large vertical span, this can result in extremely complex, hard to interpret transformation diagrams. Finally, crossing diagrams are best suited to the display of simple conversions, such as resolving a naming conflict (for example, if the source object class for a country has an attribute “name”, and the target has an attribute “countryName”). Conversions requiring data manipulation (such as currency conversions) cannot be easily represented in this form. More complex transformations involving multiple attributes (for example, converting Euclidian to Polar coordinates) are even more difficult to represent in a clear manner.

An alternative approach to visualising transformations is therefore required. One possible approach is to consider the FOM to be the result of a functional transformation of the SOM. If this concession is made, the transformation function can be visualised using a *function graph*. A function graph is a directed acyclic graph that shows the flow of data from source to target. This technique is used successfully in tools like *Agilent VEE OneLab* and *National Instruments LabView* to perform transformations and analysis of data gathered from instruments.

The visualisation employed by these tools shows the conversion task as a function graph. Each input source is visualised as a potential starting point for a link, and each target is visualised as a potential end point for a link. On a *publication transform graph*, the sources are formed from the set of provided data classes on the SOM; the targets are formed from the set of published data classes on the FOM. Conversely, on a *subscription transform graph*, the sources are formed from the set of subscribed data classes on the FOM; the targets are formed from the set of used data classes on the SOM. By convention, sources are listed on the left hand side of the canvas, and targets are listed on the right hand side of the canvas, with data flow shown from left to right. However, these source and target points can be freely placed on the graph canvas, allowing maximum freedom in expressing the flow of data. The user then connects source and target points with links, representing the flow of transformation. In order to be

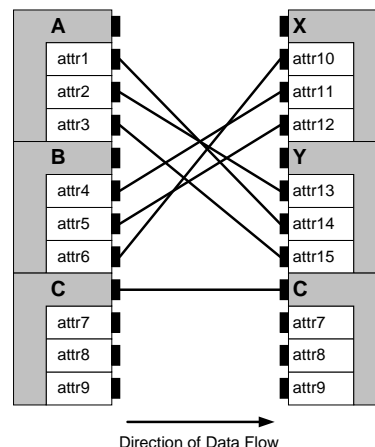


Figure 8: Crossover visualisation of a transformation from a SOM to a FOM.

connected, a source and target point must have the same data type; source and target points that do not share the same data types must be connected via an intermediate function.

The function graph can also contain intermediate *operations*. Each operation, represented as a box on the graph, contains a number of target points, representing inputs to that operation, and source points, representing outputs of the operation. These operation source and target points can be used in the same way as graph sources and targets.

Each source can provide data to multiple targets; however, targets must be connected to one and only one source. This restriction is imposed by the underlying process of code generation. The graph targets, and the inputs to each intermediate operation must be uniquely satisfiable, as they must be used as a single return value or operation parameter. However, an individual source can provide data to as many targets as is required.

Publication/Subscription graphs impose another restriction on cardinality. The net connection between sources and targets - that is, the overall connections between graph inputs and graph outputs, ignoring intermediate operations - must also be restricted. Regardless of the path through the intermediate operations, a graph output may be connected to at most one graph input; however, a graph input may be connected to multiple graph outputs. This restriction is imposed by the fact that the uniform availability of graph inputs cannot always be guaranteed. In order to satisfy a many-to-one relationship between inputs and outputs, both sources must be available at the same time. However, neither a federate nor the RTI can guarantee the availability of an Object Class or

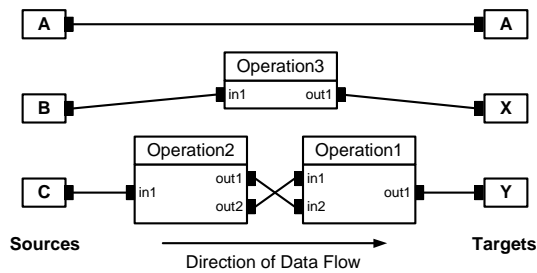


Figure 9. A publication function graph, showing the transform from a SOM to a FOM.

Interaction at any given time step. As a result, situations requiring such a guarantee must be avoided. This restriction could be overcome through the application of queuing or caching strategies; however, approaches of this nature are beyond the scope of this discussion but would be a good candidate to become a service of the component-based model described earlier.

Figure 9 shows an example publication transformation graph. In this example, a SOM containing classes A, B and C, are transformed into a FOM containing classes A, X and Y. Class A is passed untransformed; the B|X and C|Y transforms are processed by operation chains. These sources and targets could be Object Classes, Interaction Classes, or Routing Spaces. A separate function graph would be required for subscription; on this graph, the FOM would be the source.

Operation definitions

The definition of transformations between SOM and FOM requires the use of intermediate operations. These intermediate operations are also functional relationships; as a result, the same flowchart visualisation can be used to design simple conversion operations. When defining an operation, the graph sources are formed from the inputs to the operation; graph targets are formed from the outputs of the operation. These function graphs can also use intermediate operations, which in turn use operations of their own, and so on.

Operation definitions are subject to cardinality restrictions on individual links; individual targets must be supplied by exactly one source. However, the net connection restrictions do not apply, as the availability of input parameters is guaranteed; a function call cannot be made unless all parameters are uniquely satisfied.

Dead Reckoning

The process of Dead Reckoning can also be posed as a functional dependency: attribute values at time $t + \delta t$ are a function of the attribute values at time t . The

definition of Dead Reckoning strategies is therefore another candidate for definition using a function graph.

In these graphs, the sources are the values of attributes at the initial time step, while targets are the values of attributes at the next time step. Using the component-based development model, a dead reckoning strategy can then be applied to the object class within the integration logic without impacting upon the simulation logic.

Dead Reckoning graphs also require an additional 'time' source. This source is not present in publication/subscription or operation graphs. Time plays an important part in the dead reckoning process; however, time is a property of the simulation rather than the object class to be dead reckoned. As a result, the graph has no access to time in formulating representations for dead reckoned values. This problem can be overcome through the use of a "pseudosource" to represent time. This pseudosource behaves in the same way as a graph source; however, its value is obtained from the simulation, rather than the object class undergoing dead reckoning.

Problems with Function Graphs

The function graph is a flexible mechanism for representing the internal logic of data transformation, intermediate operations, and dead reckoning. However, there are some notable limitations to this visualisation technique.

The function graph is ideal for the representation of simple data flow. The simple directed graph clearly represents the flow of data from source to target through intermediate operations. When converted into generated code, the graph corresponds to a series of function calls and assignment statements. However, very few algorithms can be expressed in such terms; complex data processing requires looping and decision constructs. These constructs are difficult to represent using a graph, as they break with the directed flow metaphor underlying the visualisation. Such complex algorithms are better suited to description in code. Tools like LabView and Vee have made attempts to represent looping and decision constructs visually; however, they are generally non-intuitive, and weaken the visual metaphor.

One workaround for this problem is to permit the use of library operations whose API definitions are known, but whose internal workings are not. These operations can be inserted into function graphs as intermediate operations; however, unlike normal operations, no operation definition graph is provided. The user utilizing these library operations need not know or have

access to the implementation details. A combination of graph definitions and library operations can then be used to satisfy any data transformation requirements.

The freeform nature of the function graph also introduces the potential for obfuscation. The ability to freely place graph elements could potentially lead to unintuitive or needlessly complex graph layouts. While not strictly in error, poor graph layout could lead to confusion as to the purpose of the function represented by the graph. This problem could be overcome through the application of automated layout algorithms; however, automated graph layout is a complex problem in itself.

Assignment: Deployment View

Once the composition and integration requirements of a federate have been defined, the visualisation task can turn to federation deployment. In order to deploy federates across a network, they must be assigned to hosts.

UML provides a convenient notation for the deployment of components over a distributed network in deployment diagrams (Rumbaugh, *et al.*, 1999). In this notation, each host is represented as a cube, with components deployed on that host drawn inside or below the cube. Figure 10 shows an example of a UML deployment diagram. In this figure, Host1 is physically connected to Host2; F1 and F2 are components deployed on Host1. No component is deployed on Host4; however, all communications between Host3 and Host1/Host2 must pass through it.

However, UML models connections between machines as a graph. This is a useful representation for a WAN or internet, where direct communication channels between hosts are sparse. However, on a simulation network using a central RTI, this graph is fully connected. In such a case, there is little value in visualising the full extent of physical network connectivity.

An alternative representation utilises a backbone model of network connectivity. The network is represented as a backbone, with hosts connected to it. This visualisation is better suited to LAN models as the available bandwidth is shared between all connected hosts. This model is also well suited to HLA federations. The broadcast communications model employed by HLA does not allow for private communication channels between federates; as a result, visualising private communication channels is wasteful and potentially misleading.

To minimize the potential for confusion, the UML notation for individual hosts can be maintained within

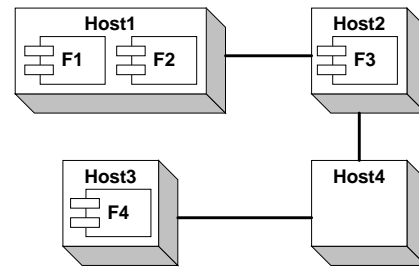


Figure 10: A UML Deployment diagram.

this visual model. Federates can then be listed inside the representation of each host. However, Federates are not the only entities that must be deployed in a federation. The RTI, and any runtime services required by a federation to manage initialisation, synchronisation and entity management must also be started. These entities are participants in a federation, just like federates; as a result, they can be displayed in the same manner as federates, with some highlighting to indicate their significance to the federation.

Just as tools can be used to automate the code generation process, tools can also be used to automate the deployment and execution process. One of the characteristics of a component is the ability to package all of its constituent files, including its interface and metadata description, into a self-contained and self-describing package. This package contains all the information required to execute the component. By adopting a CBD approach to federate design, we provide a similar degree of self-containment for federates. Given the metadata wrapped into the self-contained federate, it is possible for a deployment tool to automate the delivery of federates to remote hosts, and to control the execution of each federate.

The assignment of federates and runtime services to hosts is an important aspect of deployment behaviour; however, it represents a static view of the deployment process. Once a federation has been started using the deployment tool, the deployment diagram can become a dynamic model of the executing federation. If this visualisation is integrated with the deployment tool, it becomes possible to not only visualize changing runtime conditions of the federation, but to alter the deployment characteristics of federates to suit these changing conditions.

The volume of network traffic generated by a federate and the processing load required to process a federate are both features that can be visualised and displayed in real-time on a deployment diagram. Changes in font size, font face, color, or highlighting can be used to indicate changes in these runtime characteristics. After observing these changes, the user can perform runtime

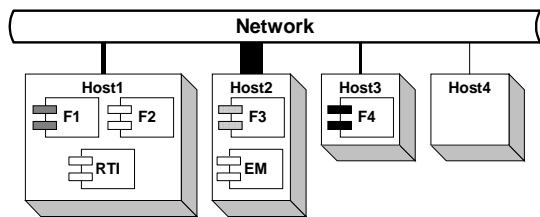


Figure 11: A Backbone deployment diagram, showing runtime characteristics of the simulation.

load balancing of federates by dragging federates from one host to another. The process of persistently saving federate state, resigning from the federation, rejoining the federation on another host, and restoring state can all be handled transparently by the deployment tool as a response to a drag operation on the visualisation.

Characteristics of the network itself can also be visualised: changes in backbone thickness or color can be used to symbolise the extent of network utilization. These characteristics can be used to identify the need for improved network infrastructure. Alternatively, they could be used to indicate problems with federate design; a constantly overloaded network could indicate a communication-heavy FOM that should be redesigned to reduce the impact upon network resources. Figure 11 shows an example of this sort of deployment diagram. In this example, line thickness indicates network utilization; the shading of components indicates processor utilization.

CONCLUSION

The Component-Based Extensions to HLA (Radeski, *et al.*, 2002) provides a strong architectural model for the design of federates in a federation. This architectural model provides a clean development framework and facilitates the easy reuse of simulation logic. More importantly, it allows a federate to be characterised using a relatively simple set of descriptors. These descriptors can form the basis for visualisations that can be used to design and develop the federate without the need to write and maintain an excessive quantity of integration code.

The visualisations presented in this paper address the design of integration requirements and the design of federations. However, the issue of simulation logic design has not been addressed. One option for the design of simulation logic is to treat the internal workings of a federate as a state machine, with state transitions triggered by the arrival of data from the RTI. This state machine could be designed visually; in addition, the logic associated with state transitions could be designed visually using a mechanism similar to the function graphs described in this paper. This

approach could then be integrated easily into the code generation process. This aspect of federate design is left as an avenue for future research.

Visualisation is a well-proven mechanism for simplifying the design process of complex systems. The use of tools that utilize these visualisations promises to significantly reduce the complexity and costs associated with HLA development, as well as providing a standard, approachable and repeatable development model. As a result, the HLA design goals of simulation reuse and interoperability become easier to realize.

REFERENCES

- Calytrix Technologies (2001). *A Visual Tool to Simplify the Building of Distributed Simulations using HLA*, White Paper, <http://www.simplicity.calytrix.com>
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2000). *Elements of Reusable Object-Oriented Software*. Addison Wesley.
- Heineman, G., & Councill, W. (2001). *Component-Based Software Engineering*. Addison Wesley.
- Kassem, N., et al. (2000). *Designing Enterprise Applications with the Java™ 2 Platform, Enterprise Edition*. Addison Wesley.
- Object Management Group (2001). *Updated CCM Specification*. <http://www.omg.org>.
- Parr, S., Radeski, A., & Whitney, R. (2002). *The Application of Tools Support In HLA*. Proceedings of the Simulation Technology and Training Conference 2002 (SimTecT 2002). Paper ID 26.
- Radeski, A., Parr, S., Keith-Magee, R., & Wharington, J. (2002). *Component-Based Development Extensions to HLA*. Proceedings of the 2002 Spring Simulation Interoperability Workshop (SISO Spring 2002). Paper ID 02S-SIW-046.
- Rumbaugh, J., Jacobson, I., & Booch, G. (1999). *The Unified Modelling Language Reference Manual*. Addison Wesley Longman, Inc.
- Syzperski, C. (1998). *Component Software – Beyond Object Oriented Programming*. Addison Wesley