A LAYERED DISTRIBUTED SIMULATION ARCHITECTURE TO SUPPORT THE C2 ENTERPRISE

Arno Duvenhage
Derrick G. Kourie
Gerhard P. Hancke

University of Pretoria
Pretoria
South Africa

aduvenhage@csir.co.za, dkourie@cs.up.ac.za, gerhard.hancke@up.ac.za

Keywords:
Layered Architecture, QDEV, C2 Enterprise, Interoperability, Modelling and Simulation, Virtualisation

ABSTRACT: *Modelling and simulation can be applied to support Joint Command and Control which involves the interoperability of network-centric systems as well as legacy command and control systems. It is assumed that software will always be the glue between these systems and that a capability is required to demonstrate, support and evaluate interoperability. This paper discusses the layered software architecture of a C++ software application framework for developing applications that support the Command and Control Enterprise. The framework provides for both interoperability and modelling and simulation – the modelling and simulation features of the framework can provide for key interoperability support capabilities that would otherwise not exist. Applications that have been built using the framework are also described.*

1 Introduction

This article deals with a Joint Command and Control (JC2) context that relies on the interoperation of various military systems as well as civilian systems. There are potential advantages in leveraging the power of Modelling and Simulation (M&S) in such J2C contexts that involve Joint Operations across the whole of the defence force as well as other services.

It is assumed that the different command and control systems (legacy and net-centric) involved in joint operations will act as loosely coupled services within a bigger enterprise. This *C2 Enterprise* consists of many systems that were not designed to work as services and that do not all support the common communications protocols. A layered, distributed architecture framework is presented that provides M&S and interoperability capabilities within the C2 Enterprise.

The proposed simulation application framework makes it possible to support the C2 Enterprise in the following ways:

- The inherent M&S capability within the framework makes it possible to create applications and tools that can deploy virtual systems and equipment. The virtual systems and equipment are deployed to interact with the C2 Enterprise systems as the real systems would. This fools the enterprise into thinking all systems are available, even though specific systems could not be deployed.
- Applications and tools created with the framework are ideally equipped with the right components to interoperate with external systems and simulators.
- The framework can be used to create software bridges, adapters or gateways for *existing* systems that have to be part of the enterprise, but do not necessarily support the correct protocols or interfaces.

The framework design builds on past experience with using modelling and simulation for decision support within the military environment. The framework represents the next step in the modelling and simulation capability that has become heavily focused on the ongoing JC2 work for Joint Operations. This environment also requires rapid application development

to quickly evaluate possible software solutions for interoperability problems with ad-hoc user requirements.

The paper briefly discusses the background to the framework and how it evolved, followed by two sections that detail the requirements for the framework architecture and discuss the framework design. The paper then gives a brief overview of the framework implementation and the applications built with the framework up to now. The paper ends with a conclusion and a discussion on possible future work. The future work includes applying the framework in other environments where virtualisation and interoperability can be useful. The focus in this paper is on the software framework architecture and not on the principles, practises or formalisms of M&S.

2 Background

The Council for Scientific and Industrial Research (CSIR), South Africa, has been providing an M&S service to the local defence force and industry for more than a decade. The M&S capability (developed in-house) was initially referred to as the ***V**irtual **G**round Based Air Defence **D**emonstrator (VGD)* and was primarily applied in the air defence environment. The capability has however been steadily upgraded during the past few years in support of joint operations. The increasing focus on joint operations and interoperability is also clear from the literature on VGD:

- The design and application of an early version of VGD is discussed in *Implementing a Low Cost Distributed Architecture for Real-Time Behavioural Modelling and Simulation* (Euro SIW 2006) .
- Using VGD for decision support is discussed in *Modelling and Simulation of a Ground Based Air Defence System and Associated Tactical Doctrine as Part of Acquisition Support* (Fall SIW 2006) .
- The design of VGD is again discussed in *A Peer-to-Peer Simulation Architecture* (HPC&S 2007) .
- *Migrating to Real-Time Distributed Parallel Simulator Architecture* (SCSC 2007) proposes an update to the VGD architecture to transform it from a discrete time based to a Quantised Discrete Event (QDEV) based simulation.
- *A State Estimation Approach for Live Aircraft Engagement in a C2 Simulation Environment* (Fall SIW 2007) review a method used to inject real-time sensor data into VGD.
- *The Contribution of Static and Dynamic Load Balancing in a Real-Time Distributed Air Defence Simulation* (SimTecT 2008) proposes an update to VGD that will automatically balance the load when distributing complex scenarios.
- *An Alternative to Dead Reckoning for Model State Quantisation when Migrating to a Quantised Discrete Event Architecture* (ECMS 2008) investigates the use of alternative algorithms for converting model interaction from discrete time based to QDEV architecture.
- *The Evolution of a C2 Protocol Gateway* (Euro SIW 2008) discusses the interoperability capability developed in parallel with VGD.

The current software framework is the result of a need for rapid application development and is a culmination of the original M&S capability and the more recent C2 protocol gateway and interoperability work. The framework has a layered architecture consisting of three main layers as well as additional application specific layers: a publish-subscribe-type Inter Process Communications (IPC) backbone layer; a Quantised Discrete Event (QDEV) infrastructure layer, and; an interoperability layer. The framework architecture is described in more detail in Section 4 of this paper. The next section discusses the requirements for the framework.

3 Framework Requirements

The framework requirements can be divided into three main points: interoperability with C2 systems; virtualisation of C2 equipment using M&S, and; good code quality (i.e. scalability, usability, extendibility and other such metrics)

The framework requirements listed in this section are based on experience gained with the ongoing M&S research as well as an extensive literature review. The requirements were also checked against a UML Use Case diagram for the framework.

3.1 Interoperability with C2 Systems

To function within the C2 enterprise the framework should enable the following:

- interoperability with legacy and net-centric C2 systems and simulators,
- protocol translation when communicating with real systems and other simulations,
- object attribute translation when translating to and from external data representations,
- generic and extendable internal object data model,
- protocol bridging (acting as an adaptor or gateway for systems that do not support the correct protocol or interface), and
- dynamic addition and removal of nodes when running distributed.

3.2 Virtualisation of C2 equipment using M&S

Applications and tools created with the framework can support the C2 enterprise by deploying virtual systems when the real systems cannot be deployed. This introduces the following requirements:

- dynamic addition and removal of simulation objects like services and models,
- operator in the loop (OIL) support (with seamless switching between constructive and virtual modes),
- running in real-time and the ability to catch up if the simulation was slowed down temporarily (soft real-time),
- running in reverse, running as fast as possible and pausing execution,
- the ability to jump in time, and
- a configurable frame rate.

It should be possible to distribute the execution over multiple nodes for increased performance. Parallel execution (distribution over multiple CPU/Cores on one node) should also be considered – to utilise the power of multi-core nodes.

3.3 Good Code Quality

The framework is intended for rapid development of technology demonstrators and prototyping of software. More often than not the applications are also subject to ad-hoc changes in user requirements. The quality of the framework code base will ultimately determine how the framework is used. The framework code base should adhere to the following:

- use of Standard Template Library (STL),
- use of 3rd party components,
- object-oriented design,
- memory usage tracking,
- built-in profiling,
- real-time execution,
- fault-tolerance,
- long up times,
- usability,
- maintainability,
- extensibility,
- reliability, and
- portability.

The framework should additionally support XML based scenario loading and saving. It should also support logging and off-line review (debriefing) of scenario execution.

Ultimately the framework should make it easy for application developers to create good quality applications and tools that support the C2 Enterprise.

4 Framework Design

A layered architecture was proposed in order to meet various aspects of the requirements outlined in the previous section. The framework architecture is divided into five layers shown in Figure 1. The five layers are: the *backbone* layer; the *infrastructure* layer; the *interoperability* layer; the *simulation* layer, and; the *application* layer. The shaded layers are not part of the framework yet, but will be required for future applications of the framework and will be discussed briefly in Section 7 of this paper.

Using a layered architecture has the following advantages:

- The framework is more usable in the sense that changes in one layer don't affect other layers as much. Each layer is a different C++ project that can be compiled and modified without affecting the other layers.
- Applications have a reduced development time since application development involves extending and using existing components rather than creating everything from scratch.
- Applications have a reduced build time since you only have to compile the layers that have changed.
- It supports multiple teams working on the same code base since different teams would normally focus on different layers of the framework. This allows different teams of developers to develop multiple independent simulations concurrently using the same framework code base.

The layers of the framework can be mapped onto specific layers of the OSI model (Figure 1) to get some perspective on the functionality of each layer – from the application developer's point of view.

4.1 The Backbone Layer

The backbone layer contains the inter process communication (IPC), memory management and networking components. This layer supports distributed and parallel execution of processes. The backbone layer components are grouped according to their function: *core* components, *common* components, *network* interfaces and *backbone* components.

Core Components

The backbone layer contains an object factory that can uniquely identify and inherently construct any object within the object hierarchy. Abstract and concrete class types can be added into the object hierarchy and any object in the hierarchy is automatically added to the factory. The factory enables automatic construction of objects based on type.

An object can be added to the object hierarchy by inheriting from a specific interface and including the relevant class members. Objects can be identified within the hierarchy in two ways: based on the object's class name (string value), and based on the hash value of the class name for faster lookups.

Operations to check the type of object pointers as well as perform safe casting are available in the backbone. The object type information also indicates parent type which allows an object to be identified based on the object's type as well as the type of any one of the object's parents within the object hierarchy.

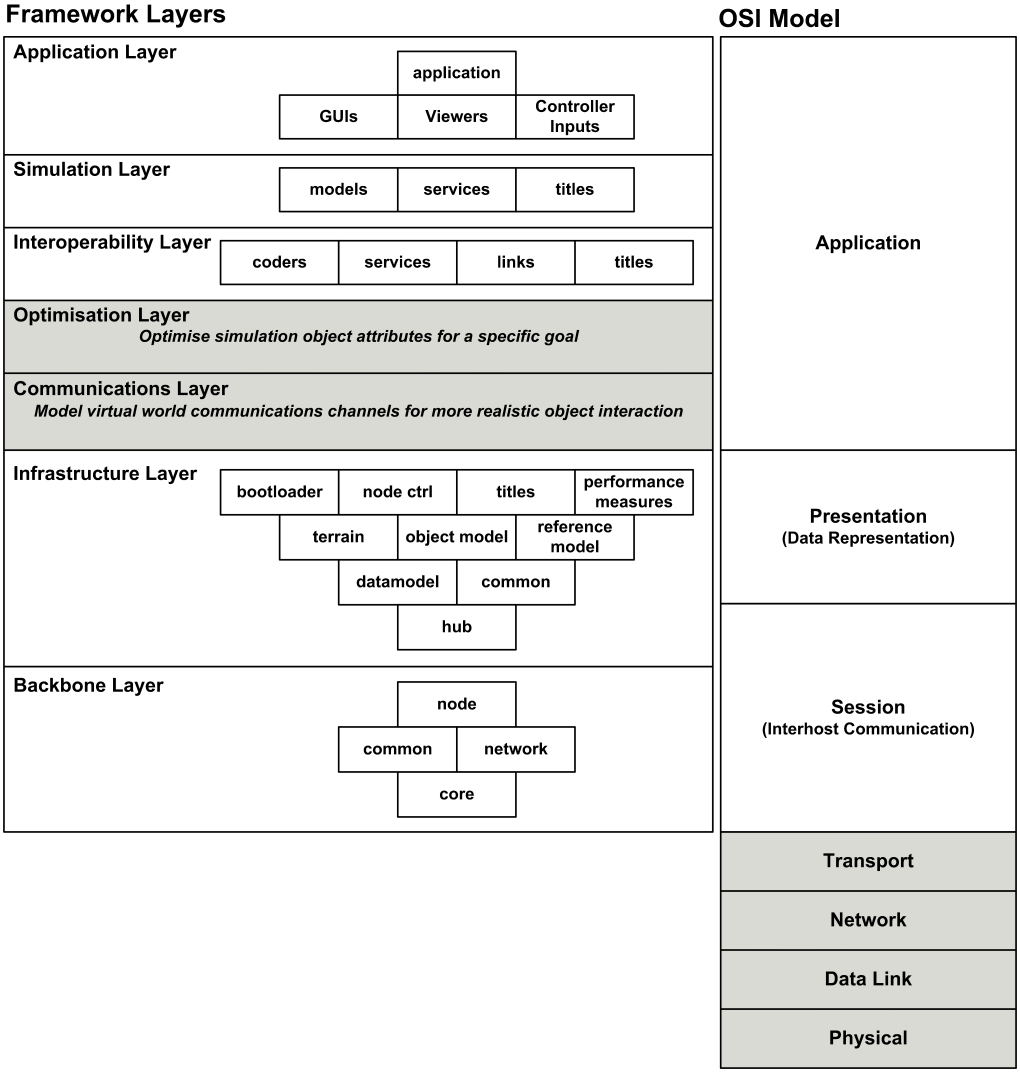

Figure 1: VGD Layers

The core components include a custom memory manager that helps track down memory leaks. All objects in the backbone object hierarchy inherit from a base class that have the memory operators overloaded to store the file and line number of the allocation and to register the allocation. It is then possible to at any point examine the

registered memory allocations. Doing this when the applications exits provides the location in the source code of memory allocations that were never de-allocated (memory leaks). It is possible to add additional features like memory buffer overwrite and underwrite protection to the overloaded memory operators.

The core components also include:

- error and exception handling classes,
- a custom pseudo random number generator,
- a high performance timer,
- a customizable output class (console output, logging, etc), and
- operating system abstractions that support portability.

Common Components

The backbone layer contains the following common components:

- advanced containers like bidirectional maps and indexed maps,
- utilities for performing string hashing, etc.,
- multi-threading base class and process control,
- utilities for retrieving raw and formatted time of day, date, etc.,
- a time manager capable of translating between simulation time and the time external systems and simulations are running at,
- generic IPC components that enable state quantisation and integration (makes QDEV simulation possible).

Network

The networking components are split into two main component types:

- **Network interfaces**: The network interface components do the low-level reading and writing of binary data from various interfaces like files, network transport interfaces and even hardware interfaces like RS232. The network interface classes all present the exact same interface, providing a unified way of accessing binary streams.

- **Network coders**: A coder object is a wrapper for an interface object and is responsible for translating or formatting higher-lever application data. Unlike network interface classes, network coder classes do not present the same interfaces, since the interfaces are application specific. Any network interface can however be used by any network coder.

The interoperability layer uses extended network coders referred to as protocol coders that translate the various data formats and protocols used by the different external systems and simulations. The interface-coder-concept allows any available interface to be used with any protocol coder. This adds to the flexibility of the interoperability capability of the framework.

Backbone Node

The IPC is based on a publish-subscribe-type message passing scheme among backbone objects. The *backbone node* components represent the bulk of the IPC functionality within the backbone. Data flows from one backbone object to another in the form of issues, where an issue encapsulates an event or list of events. What issues a backbone object can publish and where the issues go are determined by the *titles* a backbone object registers and the subscriptions other backbone objects make to these titles.

Issues are also inherently generated by the backbone layer for each object in the following cases:

- The backbone generates a *title issue* whenever an object registers or deletes a title. The issue is then broadcast to all objects on all nodes.
- The backbone generates a *subscription issue* whenever an object registers or deletes a subscription to a title. The issue is then broadcast to all objects on all nodes. Any object that has the relevant title registered will then process the subscription to that title.
- The backbone generates a *subscription issue* in response to the delivery of a *title issue* if the recipient of the *title issue* has a subscription to the relevant title. The *subscription issue* is then sent to the object that originated the *title issue*.

This passing of issues allows backbone objects to register and delete titles and to add and remove subscriptions to other titles in an ad-hoc fashion during runtime: registering a title will trigger a *subscription issue* from all objects that have a interest in the title; making a subscription will create a *subscription issue* that is processed by all the objects that have the relevant title registered.

An object can publish any type of information in the form of issues. Issues contain objects which are also called *titles*. Titles are also part of the core object hierarchy. The basic title interface contains methods for streaming and de-streaming the title attributes to and from a binary stream. This means that title objects can be automatically created by the core object factory as well as be converted to and from binary when transmitted from one node to another. This makes it very easy to support any number

of title types without having to modify the backbone layer.

Title subscriptions deliver all the data published on the title (i.e. a subscribing object will receive everything the relevant publishing object publishes). This is the default subscription type: objects can also subscribe to only have access to the last issue from each publisher. More subscription types can also be implemented. The common components in the backbone that enable state quantization and integration (mentioned earlier) can then be used by an object to decrease the amount of information that is required to flow over the backbone.

The backbone runs at a fixed frame rate which determines the simulation time step size. Each backbone object has a very simple interface that is called by the backbone at a pre-configured trigger-frame (i.e. at every *n*'th frame where *n* is the trigger-frame). The backbone calls an object to give it time to update itself and read and publish issues.

The backbone layer uses a separate component, called a hub, to transfer issues from one node to another. The hub manages the inter process communication (IPC) without affecting the rest of the backbone layer. The hub interface is part of the backbone layer, but the hub implementations are part of the infrastructure layer and will be discussed in more detail in the next section.

The backbone executes each simulation frame in five steps using conservative (or lock-step) time management:

1. The backbone reads all the issues published in the previous frame.
2. The backbone then delivers the issues to the correct backbone objects. The backbone keeps on reading and delivering issues until all nodes are finished with the previous frame.
3. The backbone then calls all the backbone objects that have a trigger frame matching the current frame. The backbone objects update themselves and get a chance to publish any new issues. New published issues are temporarily stored in the backbone.
4. The backbone then sends out all the new published issues to the other nodes.
5. The backbone then indicates to other nodes that it is finished with its current frame and continues to the next frame.

The reading of issues and sending out of issues as well as the distribution of the issues among different nodes are handled by the backbone hub. The node also uses the hub interface to signal the end of its current frame. The hub interface will be discussed in more detail in the next section.

Backbone objects always publish issues for the next frame and the backbone only sends out those issues once all the objects have been called. This can be seen as a form of *double buffering* since objects only have access to new issues in the next frame. This situation is ideally suited to parallelisation and the backbone can safely call objects concurrently within a frame. Concurrently calling backbone objects makes sense when objects become very resource intensive. This will allow the framework to better utilise the potential of multi-processor systems or multi-core CPUs.

4.2 The Infrastructure Layer

The infrastructure layer extends the generic IPC capability of the backbone layer for modelling and simulation of spatial, time-based objects. This layer also includes components to help debug and measure the performance of simulation objects. The infrastructure layer components are grouped according to their function: *hub* components, *datamodel* components, *common* components, *terrain* components, Simulation Object Model (SOM), Simulation Reference Model (SRM), *bootloader* components, the *node control service*, Object Performance Measures (OPM) and some basic title objects.

Backbone Hub Components

The backbone uses a separate component, called a hub, to transfer issues from one node to another. The hub implementations are found in the infrastructure layer and not in the backbone layer since it was desirable to be able to configure the hub with the boatloader using the XML scenario.

The hub specifies the type of inter process communication (IPC) used. This makes it possible to change the backbone infrastructure from a distributed peer-to-peer TCP scheme to a parallel memory-mapped scheme (or any other scheme), by only replacing the hub. The hub is in charge of doing any relevant optimisation or configuration of the transport medium.

The hub controls the inter-node communication, synchronisation, node addressing and inter-node connection brokering. It also controls which objects can be loaded onto which nodes. This gives the hub full control over how a scenario is distributed among multiple nodes (for load balancing, etc). The hub interface allows the backbone node to read and deliver issues published in the previous frame, synchronise with other nodes and send out new published issues.

Data Model, Common Components and Terrain

The infrastructure layer contains components for defining the internal data model of the software as well as components for defining multiple external data models to represent data from external systems. The infrastructure also contains components that can then translate between the external and internal data models. Improving the data model representation and translation is actively being pursued.

The infrastructure layer contains the following common components:

- a set of constants used for coordinate conversions, unit conversions, geodetic systems, etc., and
- an xml parser that parses xml files as a set of objects.

The infrastructure layer contains the following terrain components:

- terrain loading components,
- terrain-based line of sight (LOS) calculation components.

The terrain components are designed to be easily extendable to support different terrain formats. The terrain is loaded as a set of terrain tiles. The terrain and the relevant tiles can be specified in the scenario file.

Simulation Object Model

Simulation Object Model (SOM) components extend the backbone components: the backbone object and basic object titles are extended for modelling and simulation of spatial, time-based phenomena; the backbone objects are extended to allow loading and saving object attributes to and from XML.

Simulation Reference Model

The Simulation Reference Model (SRM) contains the coordinate representation classes for Meridian and Cartesian coordinates and vectors as well as orientation. The SRM also contains coordinate conversion classes for converting from one coordinate system to another. The SRM currently supports the Earth-Centred-Earth-Fixed (ECEF) and the North East Down (NED) coordinate systems.

Bootloader Components

The bootloader components perform the XML scenario loading and saving. The boatloader introduces an object interface that enables object attribute loading and saving in a XML format.

Any object that is in the backbone's object hierarchy and inherits from the XML interface can be loaded by the boatloader. The boatloader identifies and creates objects from the object hierarchy based on the object type name. This means that the XML scenario element names should correspond to the relevant object class and namespace names as defined in the backbone object hierarchy. The boatloader XML object interface adds methods to the object that enable it to read and write XML. Each object is in charge of loading and saving its own attributes. This, along with the use of the backbone object factory, allows the boatloader to support an arbitrary number of objects.

Node Control Service

The Node Control service is a backbone object extended to handle the backbone execution and time management. It tries to keep the backbone executing in real-time when appropriate and will try to catch up if the execution was temporarily delayed or slowed. It also does the synchronisation of events like pausing and stopping between different nodes.

The node control service provides an interface that can be used by the application layer to control the simulation execution and has to be present on each node.

Object Performance Measures

The performance of the backbone and speedup during distribution needs to be measured and analysed. This helps to optimise the application. The infrastructure layer has some profiling built in to help measure the following:

- the overall application load, which provides an indication of how well the application in running in general,
- the object execution times, which allows per object optimisation if required,
- the backbone overhead (as a percentage of frame time), which shows how much of the time is spent on modelling and how much on overheads like reading and writing issues (the overhead is an indication of the amount of issues transported over the backbone and gives an indication of how successfully the application could be distributed),
- the hub bandwidth usage (throughput and overhead), which gives an indication of the utilisation of the underlying transport medium when running distributed, and
- the ratio of titles sent to local objects vs. titles sent to objects on other nodes, which indicates how well the objects are distributed among the different nodes (objects that interact closely, exchanging a lot of data, should typically be located on the same node to minimize intra-node bandwidth usage).

4.3 The Interoperability Layer

The interoperability layer adds the protocol coders, links and services required to inter-operate with external command and control systems and simulators. The interoperability layer consists of the following:

- the *protocol coder* components,
- common interoperability services,
- the link objects for the protocol coders, and
- the titles specific to the interoperability services and links.

The *protocol coder* components are extended network coder components (discussed earlier). Communicating with external systems and simulations involves creating one or more network coders which are responsible for translating between the external system and the application. These *protocol coders* operate on the syntax or structure of the foreign data and only map one data format onto another without understanding the data (i.e. on a syntax level and not on a semantic level).

The protocol coders are not backbone objects and need to be wrapped inside extended backbone objects, called *Link* objects that can be loaded from the scenario file and called by the backbone. The link objects also have the ability to synchronise with the external systems and buffer incoming data when real-time execution cannot be maintained.

There is also a gateway service, which is a backbone object, extended to act as a router for the titles from the different links. The gateway service can route data to and from other backbone objects and can even filter and modify the titles if required.

4.4 The Simulation and Application Layers

The simulation layer allows developers to create unique simulations or tools by adding the required models, services, etc. This includes 2D/3D displays, operator interaction, logging, etc.

The application layer can be anything from a Graphical User Interface to a console box. This layer is responsible for the user interaction with the relevant application. The user interaction normally involves objects from the simulation layer as well as from the application layer.

A typical rapid application development code base would consist of many different simulation and application layer implantations that could be reused, extended or modified to quickly create new technology prototypes or demonstrator applications.

5 Framework Implementation and Applications

The entire framework design, as discussed in the previous section, is currently implemented using C++ for the Microsoft Windows platform. Porting to Linux is not actively pursued at this stage, but the design and implementation do support it.

Real-time performance is desired, but in most cases this is actually soft real-time since the operating system itself does not support hard real-time applications. There are however some cases where interoperability with existing systems that have very strict timing requirements is required. In these cases satisfactory results have been obtained by putting the time critical components of the relevant protocol coders in separate high priority threads and managing the inter-thread communication (locking, etc.) very carefully.

The bulk of the *backbone, infrastructure* and *interoperability* layer implementation were done by one of the authors of this paper (Arno Duvenhage) as part of his Masters studies in Software Engineering. The following should give an indication of the complexity of each layer:

- *Backbone*: 80 classes, 8100 lines of code
- *Infrastructure*: 70 classes, 7970 lines of code
- *Interoperability*: 40 classes, 7050 lines of code

Some interoperability links as well as the framework applications, discussed next, involved other software developers from the CSIR. Multiple applications have already been built with the framework. Three of these applications are discussed in the next section.

5.1 Testing the Gap-filling Capability of Short Range Radars

During a specific training exercise, aircraft transponder data as well as real track data from a Local Warning Radar (LWR) were available. Because of the positioning of an LWR or the terrain profile, there might be gaps in its coverage. The gaps can normally be filled by deploying short range radars at specific points on the terrain.

To test this, a model of a short range radar was deployed using an application created with the framework. The model was run on the live transponder data. The tracks generated by the model as well as tracks generated by the real LWR were then used to generate an air picture for a virtual air defence battery simulated in VGD. By doing this it was possible to determine the usefulness of short

range *gap-filling* radars as well as optimal positioning. This is an example of deploying virtual equipment within the C2 Enterprise.

5.2 C2 Protocol Gateway and Router

The framework was used to build a gateway application that can act as a message router for various systems and simulators. The C2 Enterprise can consist of many systems and simulators that cannot on their own connect to other systems in the enterprise. The gateway implements all the links required to connect to the relevant systems and exchange information with these systems. The gateway also translates the information to and from an internal representation. This allows the gateway to root information between systems, acting as a C2 hub.

An example would be the LWR (mentioned in the previous example) that cannot interface with a specific air picture display system. Using the gateway application it was possible to transfer LWR tracks as plots onto the air picture display system in real time. This gateway has been used successfully at several defence force field exercises.

5.3 C2 Protocol Gateway for an Existing Simulator

The gateway application (discussed in the previous section) was wrapped in a Dynamically Linked Library (DLL) and loaded by VGD. VGD is a virtual simulator and can simulate and entire air defence battery with specific air defence operator terminals implemented. The DLL allowed VGD access to live air picture data and operators could then use VGD to exercise and evaluate tactical doctrine on live targets. This is an example of the framework being used to create a bridge between an existing system (in this case VGD) and live systems (representing the C2 Enterprise).

6 Conclusion

It is assumed that software will be the glue between C2 Enterprise systems. The complexity of interoperability and deployment of C2 systems is however easily underestimated. This makes specifying the software requirements difficult and implementing and maintaining the software becomes very cumbersome. The requirements are sometimes unclear and frequently change as new systems are introduced. The complexity of the enterprise also makes it impractical to deploy all the systems during field exercises.

The key to enhancing the quality of C2 software solutions is using a software application framework specifically designed with quality and rapid development in mind. The framework should also support multiple teams working on different applications. This paper discusses such a framework and gives its design and the current applications of it as proof of its success.

The code-base has not undergone the rigorous testing and validation required to qualify it for use in operational systems. For now it remains part of the support capability provided by the CSIR. The performance and scalability of the framework also still need to be formally analysed and documented.

It is worthwhile mentioning that the use of the framework in no way negates the use of HLA. The framework can be applied to enhance the capability and quality of federates and could very well be extended to be a federate development framework. This is discussed in the next section.

7 Future Work

The M&S and interoperability capabilities of the framework will need to be extended as the research continues and the C2 Enterprise grows. The framework development efforts are currently focused on extending the interoperability layer as well as packaging the framework for use in existing systems and simulators. To this end, it might be usefull looking into a plugin architecture for links and coders that allow dynamic addition of coders and links during runtime, without having to recompile the code.

Figure 1 shows two layers of the framework that are not currently implemented, but will be required for future applications of the framework:

- A *model optimisation* layer that assists in determining optimal model configurations.
- A *communications effects* layer that enables accurate simulation of communications equipment and links.

The framework can possibly be extended to be a HLA federate development framework. It is possible to create an interoperability *link* component for the framework that supports HLA and the framework already supports modelling and simulation. The framework also has the potential to parallelize the federate's internal elements by calling backbone objects concurrently.

Essentially all the components of the framework except the application specific models and services that contain sensitive information can be made an open source project. Making the framework open could advance the

framework's development as well as increase the number of applications created with it.

Acknowledgements

The authors would like to thank both the Armaments Corporation (Armscor) of South-Africa and the Council for Scientific and Industrial Research (CSIR) of South-Africa for supporting this research.

References

[1] W.H. le Roux: "Implementing a Low Cost Distributed Architecture for Real-Time Behavioural Modelling and Simulation" 2006 European Simulation Interoperability Workshop, Stockholm, Sweden, June 2006.

[2] S. Naidoo and J.J. Nel: "Modelling and Simulation of a Ground Based Air Defence System and Associated Tactical Doctrine as Part of Acquisition Support" 2006 Fall Simulation Interoperability Workshop, Orlando, Florida, September 2006.

[3] B. Duvenhage and W.H. le Roux: "A Peer-to-Peer Simulation Architecture" 2007 High Performance Computing and Simulation Conference, Prague, June 2007.

[4] B. Duvenhage and D.G. Kourie: "Migrating to Real-time Distributed Parallel Simulator Architecture" 2007 Summer Computer Simulation Conference, San Diego, California, July 2007.

[5] A. Duvenhage and W.H. le Roux: "A State Estimation Approach for Live Aircraft Engagement in a C2 Simulation Environment" 2007 Fall Simulation Interoperability Workshop, Orlando, Florida, September 2007.

[6] B. Duvenhage and J.J. Nel: "The Contribution of Static and Dynamic Load Balancing in A Real-Time Distributed Air Defence Simulation" SimTecT 2008 Conference, Melbourne, Australia, May 2008.

[7] A. Duvenhage and B. Duvenhage: "An Alternative to Dead Reckoning for Model State Quantisation when Migrating to a Quantised Discrete Event Architecture" 2008 European Conference on Modelling and Simulation, Nicosia, Cyprus, July 2008.

[8] B. Duvenhage and Derrick G. Kourie: "Migrating to Real-time Distributed Parallel Simulator Architecture" Masters Thesis in Computer Science, University of Pretoria, South Africa, 2008.

[9] A. Duvenhage and L.Terblance: "The Evolution of a C2 Protocol Gateway" 2008 European Simulation Interoperability Workshop, Edinburgh, Scotland, July 2008.

ARNO DUVENHAGE is a Researcher for the Council for Scientific and Industrial Research (CSIR), South Africa. He joined the CSIR's Mathematical and Computational Modelling Research Group in January 2005 as a Software Engineer. Arno's current work involves modelling and simulation for decision support, focusing on joint operations, specializing in distributed and networked systems. Arno has a BEng Degree in Computer Engineering from the University of Pretoria, South Africa, and is currently busy with a Masters in Software Engineering.

DERRICK KOURIE joined the Computer Science Department at Pretoria University as a senior lecturer in 1978. As co-director of the Fastar/Espresso Research Group, he supervises various MSc and PhD projects. As staff member, he has various teaching responsibilities and co-ordinates graduate admissions. His experience includes a 20 year spell as editor of the South African Computer Journal, service on various academic and professional committees, engagement in peer-reviewing activities, and industry consultation.

GERHARD HANCKE joined the University of Pretoria in 1976, where he is a Professor and Coordinator of the Computer Engineering Program in the Department of Electrical, Electronic and Computer Engineering, as well as Head of the Research Group on Distributed Sensor Networks. He is a member of the Editorial Board of the Elsevier "Ad Hoc Networks" journal and has wide-ranging experience in organizing major international conferences. He is a Professional Engineer and held offices in various national and international scientific and professional bodies.