A SOFTWARE FRAMEWORK TO SUPPORT DISTRIBUTED COMMAND AND CONTROL APPLICATIONS

by

Arno Duvenhage

Submitted in partial fulfilment of the requirements for the degree

Master of Engineering (Software Engineering)

in the

Faculty of Engineering, Built Environment and Information Technology

Department Electrical, Electronic and Computer Engineering

UNIVERSITY OF PRETORIA

July 25, 2011

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

SUMMARY

A SOFTWARE FRAMEWORK TO SUPPORT DISTRIBUTED COMMAND AND CONTROL APPLICATIONS

by

Arno Duvenhage

Promoters:	Prof G.P. Hancke and Prof D.G. Kourie
Department:	Electrical, Electronic and Computer Engineering
University:	University of Pretoria
Degree:	Masters (Software Engineering)
Keywords:	Distributed simulation, interoperability, command and control, legacy systems, net-centric systems, IPC, middleware, software framework, software quality, software architecture

This dissertation discusses a software application development framework. The framework supports developing software applications within the context of Joint Command and Control, which includes interoperability with network-centric systems as well as interoperability with existing legacy systems.

The next generation of Command and Control systems are expected to be built on common architectures or enterprise middleware. Enterprise middleware does however not directly address integration with legacy Command and Control systems nor does it address integration with existing and future tactical systems like fighter aircraft. The software framework discussed in this dissertation enables existing legacy systems and tactical systems to interoperate with each other; it enables interoperability with the Command and Control enterprise; and it also enables simulated systems to be deployed within a real environment.

The framework does all of this through a unique distributed architecture. The architecture supports both system interoperability and the simulation of systems and equipment within the context of Command and Control.

This *hybrid* approach is the key to the success of the framework. There is a strong focus on the quality of the framework and the current implementation has already been successfully applied within the Command and Control environment. The current framework implementation is also supplied on a DVD with this dissertation.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

OPSOMMING

'N SAGTEWARERAAMWERK WAT BEVEL- EN BEHEERTOEPASSINGS ONDERSTEUN

deur

Arno Duvenhage

Studieleiers: Prof G.P. Hancke en Prof D.G. Kourie

Departement: Elektriese, Elektroniese en Rekenaaringenieurswese

Universiteit: Universiteit van Pretoria

Graad: Magister (Sagteware Ingenieurswese)

Sleutelwoorde: Verspreide simulasie, interoperabiliteit, bevel en beheer, ouderwetse stelsels, netwerkgesentreerde stelsels, IPC, middelware, sagtewareraamwerk, sagtewarekwaliteit, sagtewareargitektuur

Hierdie verhandeling bespreek 'n sagtewareraamwerk wat gebruik kan word om toepassings in die bevel- en beheeromgewing te ontwikkel. Hierdie tipe toepassings sluit die interoperabiliteit met netwerkgesentreerde stelsels sowel as interoperabiliteit met ouderwetse militêre stelsels in.

Die volgende generasie van bevel- en beheerstelsels gaan heelwaarskynlik geskoei wees op die tipe middelware wat algemeen in die besigheidswêreld voorkom. Hierdie tipe middelware spreek ongelukkig net nie die integrasie van bevel- en beheerstelsels, operasionele stelsels of taktiese stelsels aan nie. Die sagtewareraamwerk wat in hierdie verhandeling bespreek word, help met die integrasie van ouderwetse stelsels, taktiese stelsels en bevel- en beheer besigheidsagteware. Dit vergemaklik ook die ontplooing van gesimuleerde stelsels in die regte wêreld.

Die raamwerk doen al hierdie dinge deur 'n unieke verspreide agitektuur. Die argitektuur ondersteun interoperabiliteit en die simulasie van bevel- en beheerstelsels en -toerusting. Hierdie tweevoudige argitektuur is die geheim vir die sukses van die raamwerk. Daar is 'n baie sterk fokus op die kwaliteit van die raamwerk en die raamwerk word tans gebruik om suksesvolle bevel- en beheertoepassings te ontwikkel. Die raamwerk is op 'n DVD saam met hierdie verhandeling ingesluit.

LIST OF ABBREVIATIONS

List of abbreviations where the notation is different than the norm.

ADC	Air Defence Control
JC2	Joint Command and Control
GBADS	Ground Based Air Defence System
MSDS	Modelling and Simulation based Decision Support
OIL	Operator In the Loop
OT&E	Operational Testing and Evaluation
VGD	Virtual GBADS Demonstrator

Contents

INTRODUCTION

This part of the dissertation introduces the reader to the relevant concepts that help clarify what the Command and Control enterprise is and the role software plays in it. An overview of the research and the general layout and flow of the dissertation are also provided.

1. Background

The software framework discussed in this dissertation represents an *hybrid approach* to building support software within the context of Command and Control (C2). It combines Modelling and Simulation (M&S) and system interoperability and provides a distributed infrastructure for application development.

The framework allows legacy systems and tactical systems (like fighter aircraft) to communicate with each other and with the C2 enterprise. This is achieved by supporting the native protocols of the relevant systems and by being able to translate between them. The modelling and simulation capabilities of the framework also make it possible to model systems and equipment that are not available or maybe do not even exist yet (i.e. filling gaps in a real deployment with *virtual* systems). Applications developed with the framework can also be executed and distributed over multiple hosts using a proprietary internal publish-subscribe backbone to speed up the simulation of equipment and systems.

This chapter will give a brief overview of some C2 and M&S concepts to help the reader understand what the C2 enterprise is and the role interoperability and modelling and simulation plays within it.

1.1 The Software Application Framework

A real-time distributed M&S capability has been under development since 1998. This M&S capability was used to simulate potential system configurations within the air defence environment. This provided the South African National Defence Force (SANDF) with valuable information for their acquisition risk reduction efforts (specifically on the Ground Based Air Defence System (GBADS) acquisition program of the SANDF). In this way the M&S capability was used to establish a credible acquisition and decision support capability (Nel, Roodt and Oosthuizen, 2007).

The M&S capability has also successfully been used by the South African Army to assist in *concurrent tactical doctrine development* as well as the *evaluation of operational doctrine*: *tactical doctrine* addresses issues such as troop deployment, operational procedures and roles and responsibilities within the military environment; *concurrent tactical doctrine development* refers to the development of tactical doctrine in parallel to the system acquisition process—to such an extent that the doctrine for using a system could be ready by the time the system becomes operational (Naidoo and Nel, 2006).

The software application framework discussed in this dissertation replaces the existing M&S

capability, with a strong focus on building high quality software applications that support Joint Operations. *Joint Operations* involves the integration of C2 systems with each other and with M&S systems. The framework makes it possible to support Joint Operations in three 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 environment as the real systems would. This fools the other systems into thinking all systems are available, even though specific systems are not even deployed.
- Applications and tools created with the framework are ideally equipped with the right components to interoperate with existing systems and simulators.
- The framework can be used to create software bridges, adapters or gateways for existing systems that do not support the correct protocols or interfaces to interact with other systems. An example of this might be: an Air Traffic Control (ATC) terminal has the ability to communicate to other ATC terminals, but nothing else; a gateway could then be created that translates the ATC terminal's native protocol and information format to be understood by other systems.

This environment also requires rapid application development with ad-hoc user requirements. The applications can range from simple system-level protocol translation (making different systems communicate with each other) to military commander consoles.

1.2 Modelling and Simulation

Simulations are often classified as either live, virtual or constructive (Fujimoto, 2000):

- A *live* simulation has human operators interacting with simulated systems using real world equipment and terminals. An example of this would be operators manning real equipment that has a built in simulation or training capability.
- A *virtual* simulation has human operators interacting with simulated equipment and terminals in a simulated world. This is also referred to as operator-in-the-loop (OIL) simulation (i.e. operators manning virtual terminals).
- A *constructive* simulation has simulated operators interacting with simulated systems in a simulated word. A constructive simulation contains only computer controlled models.

Distributed simulation refers to a simulation with models spread out over multiple hosts that are communicating over a local area network (LAN), a wide area network (WAN) or the internet. The remote hosts are connected together to share resources and collaborate in a simulated environment (Tanenbaum and van Steen, 2007).

1.3 C2/M&S Interoperability

Integrating C2 systems and M&S systems is comparable to constructing software enterprise systems. System interoperability can be broken up into the following two levels of

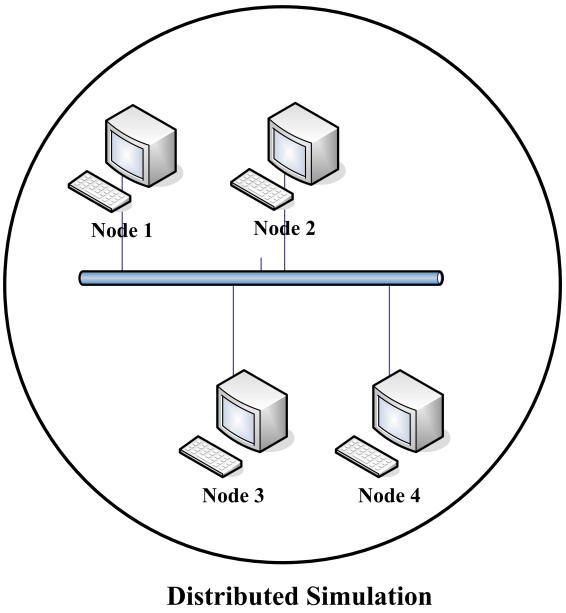

Figure 1.1: The Physical Deployment of a Distributed Simulation.

interoperability:

- low-level issues like protocols, links standards and data models (interoperability at a tactical level), and
- higher level interoperability, which involves understanding and applying the relevant information to be able to make decisions (Daly and Tolk, 2003).

The following software architecture concepts describe the interoperability of legacy and future systems:

- *Stovepiped Systems* gather, process and present specific sets of data independently, with little or no opportunity to intercept or utilise the data until it has been completely processed by the system.
- In *Network-Centric Systems* all elements are robustly networked (tightly coupled) and system interoperability is required at both the systems-of-systems level and at a sub-system level.
- Systems built using a *Service Oriented Architecture (SOA)* are loosely coupled through the use of commercial messaging technologies. A good example of this would be applications built using web-services.
- *Enterprise Systems* are created from existing systems by connecting the systems using a layer of software called *middleware*. This is very similar to the SOA approach, but enterprise middleware takes the concept one step further by formalising the way in which applications are created with services. Large business or banking systems fall into this category with many loosely coupled systems providing services to each other.

In this dissertation, military system will be referred to as either *operational* or *not qualified*: *operational systems* refer to military systems that have been qualified and officially included

as part of the capabilities of the military; systems that are *not qualified* are not officially part of any military capability and can be seen as experimental systems or prototypes. This may apply to hardware and software.

Systems can additionally be classified as *tactical systems* which refer to operational military systems and equipment that have an immediate influence over the current military situation. A *Tactical Data Link Standard* is a message-based link standard used for communication between real-world systems. Tactical Data Link (TDL) standards are often included in simulation to either simulate communication more realistically or to interoperate with the real-world systems.

1.4 Supporting the C2 Enterprise

The next generation of C2 applications are expected to be web-based or follow a service oriented architecture with common sets of enterprise middleware enabling integration. The *C2 enterprise* refers to an aggregate of loosely coupled C2 systems, software systems and simulations.

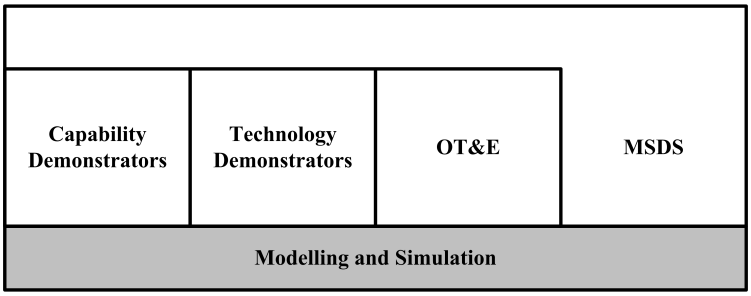

Figure 1.2: The Modelling and Simulation Competencies

The support work discussed in (le Roux, 2008) relies extensively on M&S. Figure 1.2 shows the different competencies enabled by M&S:

- Technology Demonstrators: this is the ability to develop new technologies that demonstrate a set of new capabilities within a specific user environment.
- Capability/Concept Demonstrators: this is the ability to quickly and effectively take existing technologies, put them together and then use it to demonstrate and evaluate potential capabilities or concepts within a specific user environment.
- Operational testing and evaluation (OT&E): this is the ability to validate the behaviour of operational systems.
- Modelling and Simulation based Decision Support (MSDS): modelling and simulation can be used to evaluate scenarios provided by the user; the analysis of the results from the simulations then help the user make *smart* decisions; Technology Demonstrators, Capability Demonstrators and OT&E can contribute to decision support.

It is important to note the difference between enterprise middleware and the framework discussed in this dissertation: enterprise middleware is used to construct the actual C2

enterprise; applications and tools created using the framework will only support the enterprise by providing concept evaluation, system integration and system virtualisation.

In this chapter the reader was introduced to the concepts and terminology required to understand the rest of this dissertation. The next chapter states the research objective and research question and explains the research plan. A chapter outline is also presented to help explain how the different chapters of this dissertation fit together.

2. Research Overview

The methodology followed in conducting the research is laid out in this chapter. The research problem, research question, expected outcomes and the relevance of the work are discussed.

2.1 Research Characterisation

The research objective was to design, implement and evaluate a *software application framework* for supporting the C2 enterprise. The research question is: *What should a software application framework for creating support software within the command and control environment look like?* The research outcomes include the actual software application framework as well as the key requirements for supporting the C2 enterprise. These research outcomes will contribute to further research in the field of system interoperability and M&S within the context of C2. The work also contributes to a larger vision of a unified open architecture for C2 system development.

2.2 Research Plan and Dissertation Outline

Figure 2.1 shows the chapter outline of this dissertation and shows that it is divided into four distinct parts. Part one gives the reader a brief overview of some C2/M&S concepts to help the reader understand what the C2 enterprise is and the role interoperability and modelling and simulation plays within it. Part one also gives an overview of the research performed and gives the reader the general layout and flow of the dissertation.

A literature review is done in part two of this dissertation. The literature review follows two separate paths:

- The first research path looks at how software architecture can be described. This path leads up to the formal methods required to describe and evaluate the simulation framework.
- The second research path reviews distributed simulation and system interoperability in the context of command and control software. There are a large number of technologies, specifications and standards that need to be reviewed to be able to understand the requirements and the role of software in the C2 enterprise.

The literature review places the work into perspective and provides the tools required to define

the architecture and behaviour of the software framework. Part three defines the requirements for the software framework based on existing experience within the command and control domain. The focus quickly shifts to the current framework design and implementation and the evaluation of it. The evaluation of the framework is based on a wide range of criteria that cover performance, scalability, fault-tolerance, usability, maintainability, extensibility and reliability. Critical components of the software framework design are also formally described and evaluated using the methods identified in part two of this dissertation.

Part four of this dissertation discusses the lessons learned and possible future work on the framework.

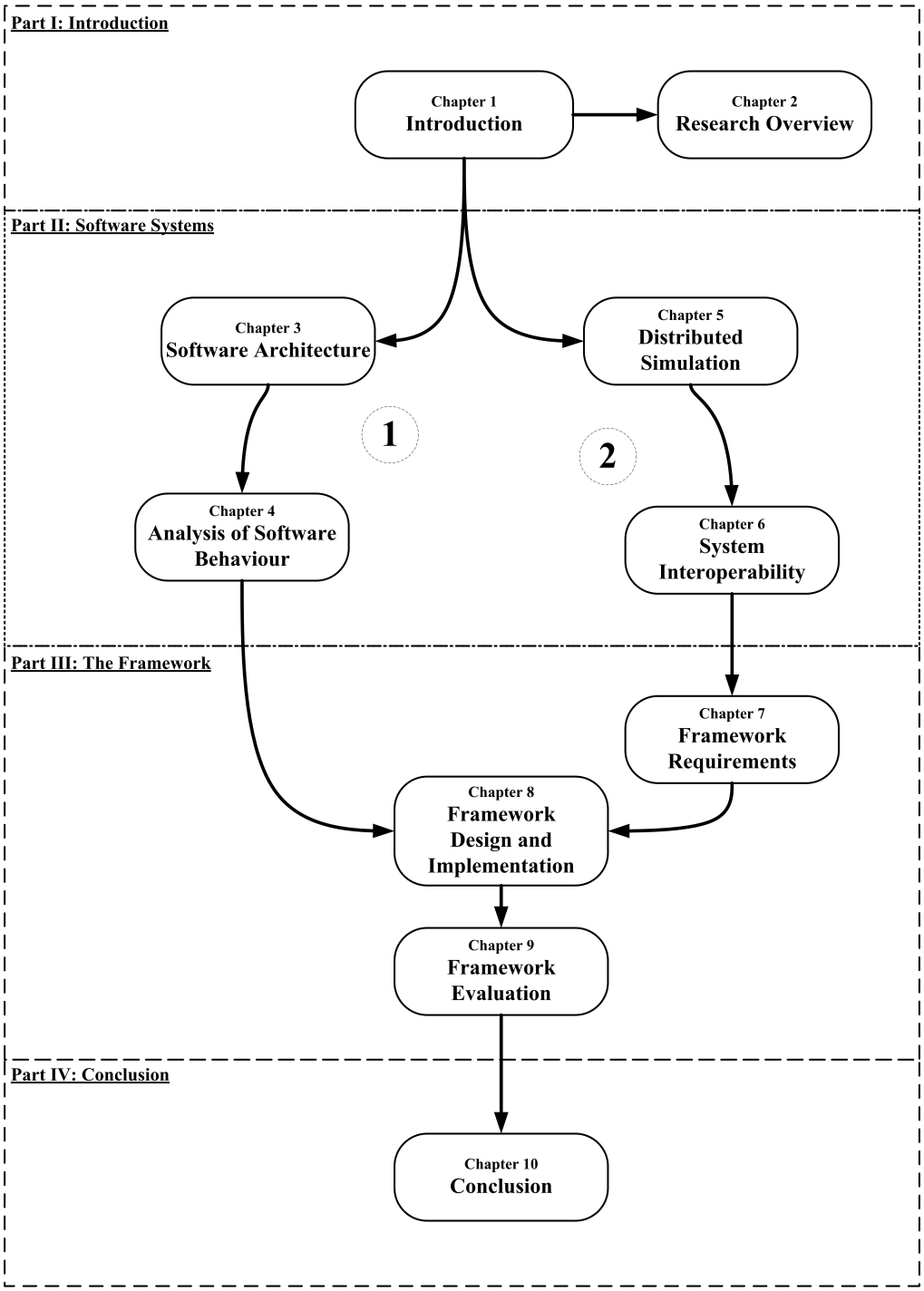

Figure 2.1: The Dissertation Chapter Outline

LITERATURE REVIEW

This part of the dissertation presents a literature review that follows two separate paths. The first research path reviews how software architecture can be described and leads up to the formal methods required to define and evaluate software behaviour. The second research path reviews distributed simulation and system interoperability in the context of Command and Control software. The literature review places the work into perspective and provides the tools to define the architecture and behaviour of the software framework.

3. Software Architecture

This chapter marks the start of the first research path of this dissertation (refer to Figure 2.1) and gives an overview of software architecture and its importance in both the design and analysis of distributed simulation and software systems. The software architecture concepts and terms discussed here are used to formally describe the proposed application framework architecture in part three of this dissertation. The different sections of this chapter discuss different views of software architecture that complement each other, but cannot necessarily be mapped to each other.

3.1 Moving from Stovepipe to Network-Centric Architectures

Stovepiped systems gather, process and present specific sets of data independently, with little or no opportunity to intercept or utilise the data until it has been completely processed by the system. The analogy comes from the clusters of chimneys from wood or coal burning stoves visible on the rooftops of some old residential buildings—if new data needs to be processed a new system is simply added in parallel to the existing systems and not all the systems process and present data in a consistent way. Stovepiped systems are not designed to be interoperable with other systems. Each individual system is kept operational until the entire system can be replaced by a new system.

In *Network-Centric Systems* all elements are robustly networked and system interoperability is required at both the systems-of-systems level and at a sub-system level. Systems are moving away from the traditional strict hierarchical approach (*stovepipe* approach) of processing and presenting information. Network-centric command and control (C2) systems endeavour to share as much information as possible by making the information flowing between the sub-systems of a system available to other systems (Daly and Tolk, 2003).

Network-centric operations require sharing both situational awareness (tactical awareness) information and *operational context information*. Operational context information includes objectives, plans, orders and priorities. This allows global access to a fused homogeneous view of the situation, enabling the commander to make the desired decisions more effectively (Chaum and Lee, 2008):

- Net-centric operational concepts seek to flatten, broaden and speed information sharing between people, between information systems and between people and information systems.
- Net-centric operational concepts seek to help military commanders be better informed about the situation and synchronise their actions with the rest of his forces.

3.2 Key Architectural Styles

Software architecture describes the top-level structure of the over-arching system and describes the software components. An architectural style defines a set of components and connector types, and a set of constraints on how they can be combined. The information contained in this section is taken from (Shaw and Garlan, 1996). The following is a list of the key architectural styles:

- *Pipes and Filters*: Multiple independent components, each with a set of inputs and a set of outputs, are stringed together. The components act as filters taking the output from the previous filter and transforming it into input for the next filter. A linear sequence of filters is called a pipe.
- *Data Abstraction and Object-Oriented Organisation*: Data representations and their associated primitive operations are encapsulated in an abstract data type or object.
- *Event-Based, Implicit Invocation*: Instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with it. When the event is announced, the system itself invokes all of the procedures that have been registered for the event.
- *Layered Systems*: The system is organised hierarchically, each layer providing service to the layer above it and serving as a client to the layer below. In some layered systems inner layers are hidden from all except the adjacent outer layer.
- *Repositories*: The system consists of a central data store and independent components that operate on the data store.
- *Interpreters*: Interpreters or *virtual machines* are software systems that execute programs like a real machine would. Interpreters separate program execution from the underlying systems or hardware and the program semantics provide a level of abstraction between the system capabilities and the underlying system implementation.

Most systems are built using some combination of several styles. This allows the system architect to utilise the benefits of two or more architectural styles. The system architect can also, in some cases, use the benefits of one style to mask the drawbacks of another style. Distributed systems can be analysed using the Data Abstraction and Object-Oriented Organisation, Event-Based Implicit Invocation and Layered Systems styles. The *pipe and filter* and *interpreter* styles are also found in distributed systems, but are not as prevalent.

The advantages of data abstraction and object-oriented organisation are:

- It is possible to change the implementation of a component without affecting the interface to it.
- Problems can be decomposed into collections of interacting agents.

The disadvantages of data abstraction and object-oriented organisation are:

- An object must know the exact identity of any object it interacts with (An object needs to know which object's interface to use, even though multiple objects could have the same interface).

- If one object is replaced with another, all objects that interact with it will have to be informed of the new identity.

The advantages of event-based, implicit invocation are:

- Strong support for reuse. Any object can gain access to the system by just registering for the relevant events.
- Eases system evolution. Components may be replaced without affecting the interfaces of any other components.

The disadvantages of event-based, implicit invocation are:

- A component that sends out an event has no control over the processing of that event.
- If a system only supports data transfer through events then accessing/changing data in a central repository, for example a terrain database, could become complicated or a performance bottleneck.
- The meaning of a procedure that sends out an event depends on the procedures registered on that event. This can make analysing and debugging a system complicated.

The advantages of layered systems are:

- Support design based on increasing levels of abstraction.
- Changes to the function of one layer affects at most the two adjacent layers.
- Support reuse—different implementations of the same layer can be used interchangeably.

The disadvantages of layered systems are:

- Not all systems are easily structured in a layered fashion.
- The overhead of having to call through all the layer interfaces might be too much.
- It might be difficult to find the right levels of abstraction.

Identifying which style(s) a system uses can help identify the good and bad traits of the system during system evaluation as well as during system design. This is used to evaluate some distributed simulation technologies in Chapter 5.

3.3 Design Patterns

A design pattern abstracts or describes key aspects within software architecture that help make the architecture reusable and extendable. Design patterns are like templates for software design that free software developers from thinking about implementation details when designing or describing software systems. This section serves as a quick reference and

provides only a short definition of the various design patterns from (Gamma, Helm, Johnson and Vlissides, 2004) and (Schmidt, Stal, Rohnert and Buschmann, 2000). Part three of this dissertation describes key components of the *simulation software framework* architecture using specific design patterns.

The standard set of design patterns are defined in (Gamma et al., 2004). Design patterns can be divided into *creational patterns*, *structural patterns* and *behavioural patterns*. Design patterns can also be divided into *class* patterns and *object* patterns, with class patterns utilising inheritance while object patterns utilise object composition and association. Additional design patterns, specialised for concurrent and networked objects, are discussed in (Schmidt et al., 2000).

3.3.1 Creational Patterns

Creational patterns hide how instances of classes are created and put together. The creational patterns are:

- Abstract Factory: Provides a class for creating objects without having to specify the exact type of the object. The abstract factory has to be extended for each class that needs to be constructed.
- Builder: Separate the construction and representation of a complex object to allow the same *builder* to represent an object in many different ways. One object can be represented in many different ways by using a hierarchy of builder classes.
- Factory Method: Defer class instantiation to subclasses by only defining an abstract interface to create objects. Subclasses of the creator knows how to construct the relevant set of objects.
- Prototype: The creator uses a *cloning* operator on a example instance of the relevant class. Each creator only has one example instance or *prototype* and can only create one type of object.
- Singleton: This creational pattern ensures that a class has only one instance that can be accessed globally.

3.3.2 Structural Patterns

Structural patterns specify how sets of classes and objects are composed into larger structures. The structural patterns are:

- Adaptor: Wrap a class to convert its interface into another interface to be used within a different context.
- Bridge: The bridge pattern supports having only one full class hierarchy that can operate on various implementations. Each class or object in the hierarchy would have an implementor object for each implementation. Client requests are forwarded to the relevant implementor objects.
- Composite: Objects are composed into hierarchies using recursive composition (all objects in hierarchy have the same interface). Clients can then manipulate objects or aggregates of objects in exactly the same way.

- Decorator: A decorator is a wrapper that adds additional responsibilities to a object. The decorator's interface conforms to the object's interface and clients use the decorator in the same way as the object. A decorator can also be wrapped by another decorator making it possible to add several layers of added responsibility to a object.
- Facade: Hide the complexity of a subsystem by providing a simple unified interface (high-level interface) to clients.
- Flyweight: An object operates on external attributes (i.e. attributes that are associated with the object in some way, but are not part of it). Associating different sets of external attributes with the object allows the object to assume the responsibilities of many different objects. This makes fine grained object support possible without the overhead of having a large amount of object instances.
- Proxy: A proxy is a placeholder for another object to be able to defer instantiation, control access to the object or provide a local representation of a remote object.

3.3.3 Behavioural Patterns

Behavioral patterns describe the behaviour of sets of classes or objects as well as the flow of control or communication between them. The behavioural patterns are:

- Chain of Responsibility: Senders and receivers of messages or requests are decoupled by having one or more handler objects between them. The sender has no explicit knowledge of who will handle the request.
- Command: Requests are encapsulated in objects with the same interface to execute them. Requests can then be handled like any other object and users of the requests do not have to know the exact format of a request or even what a request does.
- Interpreter: The interpreter pattern describes how to define a grammar for a specific language, represent constructs in the language and how to interpret the constructs.
- Iterator: The iterator pattern allow aggregate objects like vectors and maps to accessed sequentially without exposing the type or implementation of the aggregate objects.
- Mediator: A mediator is responsible for controlling and coordinating the interactions of a group of objects. The objects only have to know the mediator. This reduces the coupling between objects. It also reduces the amount interconnections between objects and make the objects more reusable.
- Memento: Allow objects to use opaque tokens to save and restore their state.
- Observer: The observer design pattern defines a one-to-many relationship between objects. One or more *observers* can be registered on one *subject* with the observers called inherently when the subject changes state.
- State: Object behaviour can be encapsulated into specific state objects that allow the object's behaviour to change when its internal state changes.
- Strategy: The strategy design pattern encapsulates algorithms to make them interchangeable (have the same interface). Clients can use any algorithm from a family of algorithms by instantiating the relevant *strategy* from the strategy hierarchy.
- Template Method: Defines an algorithm or operation in terms of abstract operations that subclasses can override to create concrete behaviour.
- Visitor: A visitor is an object that encapsulates an operation that can be performed on the elements of an object structure. A new operation can be defined by a new visitor without having to change the implementation of the object structure.

3.3.4 Service Access and Configuration Patterns

Service Access and Configuration patterns address how to effectively design and configure application access to the interfaces and implementations of services and components in stand-alone or networked systems:

- Wrapper Facade: This pattern addresses application portability by wrapping low-level or system specific application programming interfaces (APIs) with more generic reusable interfaces.
- Component Configurator: This pattern allows an application to link and unlink its own component implementations during runtime.
- Interceptor: The interceptor pattern allows additional application-specific services to be *plugged into* existing software. This is different from the component configurator pattern since these *plugins* are unknown to the application framework.
- Extension Interface: This pattern allows interfaces to be defined that specify how application extensions may be created. This helps to make the behaviour of *plugins* safe for the rest of the application.

3.3.5 Event Handling Patterns

Event Handling patterns describe the handling of events in networked event-driven systems:

- Reactor: The reactor pattern allows an event-driven application to accept service requests from multiple clients and then route the requests to the correct components within the application.
- Proactor: The proactor pattern allows an event-driven application to accept service requests that are triggered by the completion of asynchronous operations and then route the requests to the correct components within the application.
- Asynchronous Completion Token: This pattern allows an application to accept and process the responses of asynchronous operations it invoked on services.
- Acceptor-Connector: This pattern decouples the connection and initialisation of cooperating peer services in a networked system from the processing they perform once connected and initialised.

3.3.6 Synchronisation Patterns

Synchronisation patterns simplify locking in concurrent systems to prevent the corruption of a system's internal state:

- Scoped Locking: The scoped locking pattern ensures that a lock is automatically acquired when entering a specific scope and then released automatically when leaving the scope, regardless of the return path.
- Strategised Locking: This pattern parameterises the synchronisation mechanisms of a component. This allows one component to use different synchronisation mechanisms, depending on the concurrency architecture being used.

- Thread-Safe Interface: This pattern minimises locking overhead and also ensures that a component does not try to acquire a lock when it already has it (re-acquiring a lock can cause self-deadlock).
- Double-Checked Locking Optimisation: This pattern ensures that critical sections of code that only need to be run once are accessed in a thread safe manner, but avoids the locking overhead in consequent calls. This can be used, for example, with singletons, where the singleton needs to be created safely once and then only accessed.

3.3.7 Concurrency Patterns

Concurrency patterns address the various types of system architectures that address specific concurrency problems:

- Active Object: The active object design pattern allows method execution to be separated from the method invocation i.e. the method invocation and execution run concurrently. This simplifies synchronised access to objects that reside in their own thread of control.
- Monitor Object: This pattern synchronises concurrent method execution to ensure that only one method runs at a time within an object.
- Half-Sync/Half-Async: This pattern allows asynchronous and synchronous service processing in concurrent components. The system can make use of either or both components without the disadvantages of the one affecting the other.
- Leader/Followers: The Leader/Followers pattern has a pool of threads that can accept and process service requests from a set of event sources. Free threads wait in a queue for new service requests with the first one in the queue called the leader and the others the followers.
- Thread-Specific Storage: This pattern allows multiple threads to access the *logically* global state of operations without a locking overhead. This is done by keeping thread-local state copies for all the relevant operations and having the operations operate on the thread-local states.

A software system can utilise many different design patterns that together create the correct architecture. Describing and analysing a software system using design patterns provides a common understanding of what the software architecture looks like and how it behaves.

3.4 Publish/Subscribe Networking

The publish/subscribe interaction scheme can provide the loosely coupled connections between simulation nodes required for web-based simulation. It is an event-based interaction style and provides time, space and synchronisation decoupling between hosts (Eugster, Felber, Guerraoui and Kermarrec, 2003):

Time decoupling: The interacting hosts do not have to be connected (or *online*) at the same time.

Synchronisation decoupling: Hosts do not block when sending events and hosts get notified asynchronously of events from other hosts.

Space decoupling: The hosts do not need to know each others' identities.

The more decoupled the simulation nodes are the more scalable the simulation will be. This is because decoupled nodes operate independently of each other. The publish/subscribe style is very similar to the event-based implicit invocation architecture style. Subscribers express their interest in a specific set of events to a central event service. The event service then *intelligently* forwards events from publishers to the relevant subscribers when they are available (Eugster et al., 2003).

There are three subscription schemes that can be used to specify the set of events a subscriber is interested in (Eugster et al., 2003):

- Topic-Based: The event topic or group is identified using a keyword. This scheme is very simple to implement.
- Content-Based: The subscription is based on the actual content of the events. This is more powerful than topic-based subscriptions since an event's desired internal properties can be specified using this scheme. But it is less efficient than the topic-based scheme.
- Type-Based: The subscription is based on the event type. This is meant to be used instead of topic-based since it enables closer integration between the actual programming language and the middleware.

Publish/subscribe systems can also provide quality of service like persistence, priorities, transactions and reliability. Transactions provide a way for events to be grouped into sequences that have to be delivered completely or not at all. This provides mechanisms to help ensure not only the delivery of individual events, but also ensure complete delivery of sets that consist of more than one event (Eugster et al., 2003).

One possible drawback of this particular scheme is the use of the central event service. The event service will undoubtedly be a bottleneck for both communication and processing performance in large virtual environments. Distributing the event service over multiple nodes or hosts could help alleviate this problem. The publish/subscribe scheme can be used to construct a real time distributed simulation architecture as described in (Duvenhage and Kourie, 2007) where the event dispatching is shared among six to eight machines.

The publish/subscribe scheme is also used for service oriented architectures and message oriented middleware (MOM). In both these cases implicit invocation and abstraction allow services or entities to be added or removed without affecting the rest of the system. Services receive messages or event notifications implicitly and the messages themselves are normally part of a predefined set of objects, also called an object model.

3.5 Service Oriented Architecture

Service Oriented Architecture defines middleware[1] architectures based on the concept of reusable services (Keen, Acharya, Bishop, Hopkins, Milinski, Nott, Robinson, Adams and Verschueren, 2004a):

[1] *Middleware* is computer software that connects software components or applications. The software can function on a single node or on multiple nodes across a network.

- Services are defined by explicit, implementation-independent interfaces.
- Services are loosely bound and invoked through communication protocols that stress location transparency and interoperability.
- Services encapsulate reusable business functions.

A system built using interacting web-services is a good example of a system utilising an SOA. A C2 enterprise will likely utilise an SOA and any software or tools that will form part of the enterprise must then be deployable as services. A SOA implementation usually includes standardised technologies that enable information exchange and the discovery of services (Schulte, 2002):

- SOAP: The Simple Object Access Protocol is a protocol for exchanging structured information. It makes use of Extensible Markup Language (XML) and Remote Procedure Calls (RPC) or HTTP. SOAP provides basic messaging capabilities and forms the bottom layer of the web-services protocol stack.
- WSDL: The Web-Services Description Langauge is an XML-based language for describing web services. WSDL is used to describe what functions are available on a web-server. SOAP can then be used to call these functions.

A well defined software architecture is critical to system interoperability and is a means of defining systems composed of systems. The concept of an *Enterprise Architecture* addresses architecture on this larger scale. The enterprise architecture defines how a set of systems will achieve its vision and goals (Hamilton and Catania, 2003).

3.6 Software Frameworks

Software can be divided into three broad classes: applications, toolkits and frameworks. This dissertation is primarily concerned with frameworks. Frameworks have the following properties (Gamma et al., 2004).

- A framework is a specific set of cooperating classes that make up a reusable design for a specific class of software.
- A framework encapsulates the application i.e. the framework calls the application code. This is contrary to an application calling a library or toolkit.
- A framework specifies the structure and interaction of classes in the application. The framework therefore specifies the application architecture and design to a large extent.
- Different applications using the same framework seem more consistent.
- Frameworks can consist of loosely coupled and reusable components to make the framework flexible and extendable.

3.7 The OSI Reference Model

The ISO model of architecture for Open Systems Interconnection (OSI) is a layered architecture with seven distinct layers. The model serves as a framework for the definition

of standard networking protocols. It will be used in part three of this dissertation to aid in the discussion of the proposed software application framework architecture.

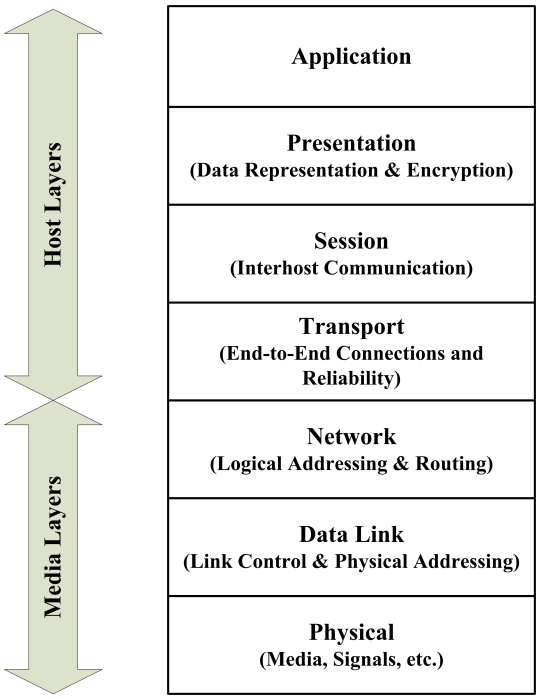

Figure 3.1: The OSI Reference Model

Figure 3.1 shows the different layers of the model (Zimmermann, 1980):

- The *Application Layer*: This layer interfaces with the software application and represents the application protocol.
- The *Presentation Layer*: The presentation layer enables the application layer to interpret the meaning of the data exchanged.
- The *Session Layer*: The session layer supports the higher-level interaction between presentation entities.
- The *Transport Layer*: This layer hides the complexity of transferring data in a reliable way between session entities.
- The *Network Layer*: The network layer allows two transport entities to transfer data over a network connection using logical addressing—the routing of data occurs between network entities and is transparent to transport entities.
- The *Data Link Layer*: This layer is in charge of link control and physical addressing between network entities.
- The *Physical Layer*: This layer provides the mechanical and electrical characteristics to establish, maintain and release physical connections and transfer data.

This chapter gave an overview of software architecture and its importance in the design and analysis of software systems. The next chapter will discuss some more formal methods required to evaluate concurrent software systems.

4. Formal Analysis of Software Behaviour

The behaviour of software systems can be attributed to or even explained by the architecture of the system. The previous chapter showed that a system's structure and behaviour can be defined by the architectural styles or design patterns employed in the system and its components. There are however formal specification techniques that can be used to analyse the behaviour of software systems. This chapter gives an overview of three such techniques that are applied in the third part of this dissertation to help evaluate the proposed software application framework.

4.1 UML Use Case Diagrams

The Unified Modelling Language (UML) defines *Use Case Diagrams* as a way to specify the functionality of a system. A use case diagram consists of two types of elements (see Figure 4.1): ellipsoidal elements, called *use cases* and stick figures, called *actors*.

The ellipsoidal use cases can be associated with actors or other use cases using a series of arrows or lines which can indicate communication and dependencies like aggregation and extension of use cases (see Figure 4.1). Each use case describes a functional requirement from the perspective of the relevant actor (Alhir, 2003). Actors may be human users or external systems. A use case diagram can be used to help verify the requirements of a system. Use case diagrams can however not be used to capture non-functional or implementation requirements.

4.2 UML Finite State Machines

Finite State Machines (FSMs) enables one to describe the logic of a system. A FSM consists of the system states and the transitions between them. States are drawn as rounded rectangles and transitions are drawn as arrows from one state to another. Transitions are labelled with the event that triggers the transition as well as an action to be performed. A black dot with an arrow is used to point towards the initial state.

Figure 4.2 shows a FSM with two states, A and B. State A is the initial state.

A FSM models the behaviour of a system with a finite number of states. FSMs help to test all the possible system states against the relevant system events. FSMs can be used to evaluate

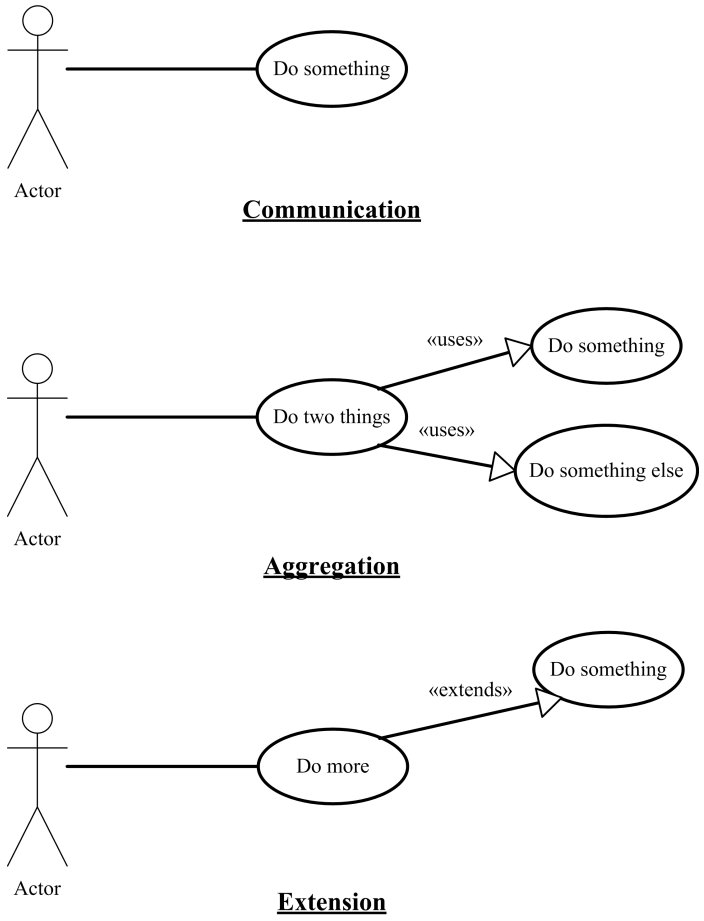

Figure 4.1: UML Use Case Diagrams

the completeness and robustness of a system design.

4.3 Communicating Sequential Processes

Communicating Sequential Processes (CSP) is a language capable of describing interaction in concurrent systems. CSP is a form of process algebra and can be used to specify and verify the concurrent aspects of systems. This section will give a brief overview of CSP.

4.3.1 Language Constructs

In CSP, a *process* can be any thread, buffer, etcetera, that acts on communications events. A *communications event* is a transaction or synchronisation between two or more processes. Communications events are instantaneous with an abstract sense of time. Communication events are assumed to be drawn from a set, called the *event alphabet*, which contains all possible events for all processes under consideration. The event alphabet for a process P is defined by αP. Some fundamental language constructs are (Roscoe, 2005):

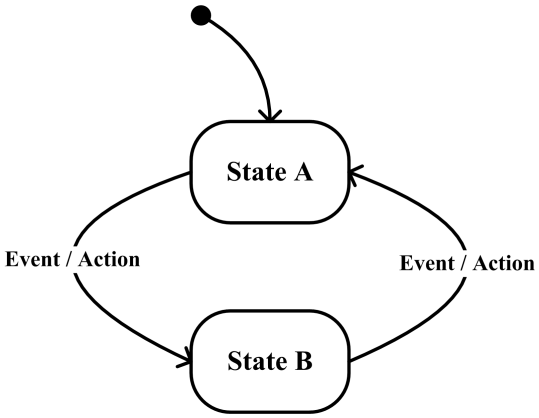

Figure 4.2: UML Finite State Machine

- *Prefixing*: Let x be an event and let P be a process. Then $(x \rightarrow P)$ (pronounced x then P) describes a process which first engages in the event x and then behaves exactly as described by P.
- *Guarded Alternative*: If x and y are distinct events, $(x \rightarrow P \mid y \rightarrow Q)$ describes a process that initially engages in either of the events x or y, depending on which event is first offered by the environment. If x is first offered, then the process subsequently behaves as process P. If y is first offered, then the process subsequently behaves as process Q.
- *Recursion*: Let $P = (x \rightarrow P)$ then $P = (x \rightarrow x \rightarrow P) = (x \rightarrow x \rightarrow x \rightarrow P)$. This can easily be generalised to more than one process. Let $MP = (x \rightarrow P | y \rightarrow Q)$ with $\alpha MP = \alpha P = \alpha Q$ and something like $P = y \rightarrow Q$.
- *External Choice*: $P \,\Box\, Q$ allows the environment to choose the first events of P and of Q and then behaves accordingly. Also $(a \rightarrow P) \,\Box\, (b \rightarrow Q)$ means the same as $(a \rightarrow P | b \rightarrow Q)$.
- *Nondeterministic Choice*: $P \sqcap Q$ describes a process that either behaves as process P or as process Q. However, the behaviour is determined by conditions internal to P and/or Q, rather than controlled by events offered by the environment. Thus, even though the environment might, at some moment, offer an event in which P can engage, the process $P \sqcap Q$ might not respond, but instead—for reasons not apparent to the outside environment—only respond at that moment to an event in which Q can engage. The behaviour of $P \sqcap Q$ as seen from the environment thus appears as non-deterministic.
- *Conditional Choice*: $P \lhd b \rhd Q$ can be read as *if* b *then* P *else* Q. These conditionals can be applied to events as well.

4.3.2 Describing Distributed Simulators

Distributed simulations will have two or more concurrent processes. $?x : A \rightarrow P(x)$ indicates that the process P communicates in the alphabet A. The following operators may apply to concurrent processes (Roscoe, 2005):

- *Synchronous Parallel Operator*: $?x : A \rightarrow P(x) \parallel ?x : B \rightarrow Q(x)$ insists that the two concurrent processes P and Q agree on all events that occur (P and Q synchronise on

events in $A \cap B$ and ignore other events) where $P = ?x : A \rightarrow P(x)$ and $Q = ?x : B \rightarrow Q(x)$.

- *Alphabetised Parallel Operator*: $P \ {}_A\|_B \ Q$ is a more general version of the synchronous parallel operator. P is allowed to communicate in the alphabet A and Q is allowed to communicate in the alphabet B. P and Q must synchronise on the events in $A \cap B$. $P \ {}_X\|_X \ Q$ is the same as $P \parallel Q$.
- *Generalised Parallel Operator*: $P \underset{X}{\parallel} Q$ states that P and Q must synchronise on any event in X, but may proceed independently with events outside X. This also means that P and Q cannot execute on any event in $\alpha P \cap \alpha Q$ outside of X at the same time.
- *Interleaving Operator*: $P \ ||| \ Q$ states that P and Q execute independently i.e. $P \underset{\{\}}{\parallel} Q$. This also means that P and Q cannot execute on any event in $\alpha P \cap \alpha Q$ at the same time.

Deadlock can be defined as the situation where one or more processes are waiting for access to a specific resource like a lock and cannot continue. *Livelock* can be defined as the situation where a process is not progressing, but also not deadlocked. Passage of time in concurrent processes may be indicated by a communications event that can be used to represent synchronisation between two or more processes. Distributed simulator specifications in CSP can be analysed for both deadlock and livelock between communicating sequential processes.

This concludes the first research path (refer to Figure 2.1) of part two of this dissertation. The next chapter will start with the second research path and does an in depth literature review of the history as well as current trends in distributed simulation, command and control systems and system interoperability.

5. Distributed Simulation

This chapter marks the start of the second research path of part two of this dissertation (refer to Figure 2.1) and it reviews specific distributed simulation technologies. This is done to understand the evolution of distributed simulation as well as understand the context of C2/M&S interoperability. Figure 5.1 shows a time line populated with the distributed simulation technologies discussed in this chapter.

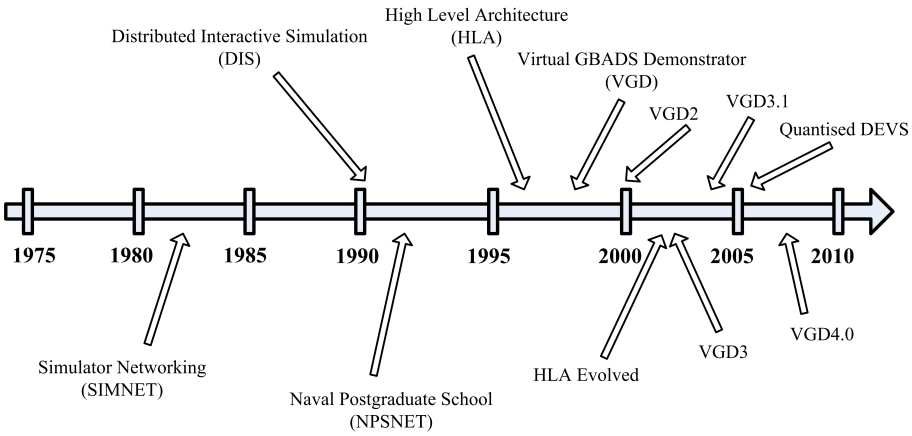

Figure 5.1: The Simulation Architecture Developments.

5.1 SIMNET

Early simulation efforts were largely fuelled by the need to scale up training and/or evaluation of new concepts and designs with ever more limited resources. Simulator Networking (SIMNET) was the first successful large-scale, real-time, man-in-the-loop distributed simulator. It was used for team training and mission rehearsals in military operations. SIMNET was developed between 1983 and 1990 by the Advanced Research Projects Agency (ARPA), then still called DARPA. It is considered to be one of the seminal contributions to the field of distributed virtual simulation technology development (Miller and Thorpe, 1995).

SIMNET's goal was to create a virtual battlefield with combatants joining from anywhere on the network using his/her simulator. As SIMNET's development progressed, its application was extended from training to evaluation of new ideas, concepts, tactics, etcetera. It was possible to manage several independent virtual worlds simultaneously. Given that the participants adhered to the rules they were subject to in the specific world, the following

was possible (Miller and Thorpe, 1995):

- training,
- mission rehearsal,
- concept development,
- doctrine and tactics development,
- testing, and
- after-mission review.

SIMNET models the virtual world as a collection of objects that interact with each other using a series of events. Terrain and other objects (buildings, trees, etcetera) are assumed to be known to all objects. SIMNET does not support changing cultural features (destroying a bridge for example) (Miller and Thorpe, 1995).

There is no central control process and simulation nodes broadcast events to each other. This means that each node on the simulation network has to receive, decode and at least partially process every event in the entire simulation. Each simulation node is in charge of processing any relevant events (like being killed) and then updating its state accordingly. Nodes always transmit their correct state and the receiver has the responsibility of, for example in a radio communications case, degrading or denying the message based on line of sight, etc. Objects are very loosely synchronized and it is possible for objects of different fidelities[1] to be part of the same virtual world (Miller and Thorpe, 1995).

SIMNET has features of a event-based, implicit invocation architecture style since nodes react to events broadcast by other nodes. The advantages of this style are reuse and easy evolution of system objects. This style can however be quite hard to analyse and debug since it is unclear what an event generated by an object means if one does not know exactly what each object does with that event (Shaw and Garlan, 1996).

Dead Reckoning is a technique SIMNET uses to minimise the amount of state update events sent by nodes: the simulation nodes agree on the dead reckoning prediction algorithm to use and each node should be able to predict the state of any object in which it is interested based on the last state update for that object; nodes also run the prediction algorithms for the objects for which they are responsible and send out state updates if the prediction and actual state are too different (Miller and Thorpe, 1995).

SIMNET has several limitations that prevent it from scaling well to more complex virtual environments. The nodes broadcast all events and each node has to read every single event on the network. The network would simply be overloaded if the simulation has too many objects or nodes and the nodes would not be able to process network packets or Protocol Data Units (PDUs) fast enough. SIMNET does not support any means of transferring static data, like terrain information, between nodes. This means that each node should have a complete copy of all static objects in the virtual world. Additionally there is no guarantee that each node's static *view* will be consistent or complete. It is expected that SIMNET can not support more than 1000 objects (Macedomia, Zyda, Pratt, Brutzman and Barham, 1995).

[1] Model fidelity can range from simple behavioural models to complex engineering models that infer behaviour by simulating internal components, dynamics, etc.

5.2 NPSNET

The Naval Postgraduate School Network (NPSNET) was developed around 1990. It was aimed at providing features similar to SIMNET, but unlike SIMNET, be able to run on off-the-shelf hardware—providing a significant cut in cost.

NPSNET went through several versions and is compliant with the Distributed Interactive Simulation (DIS) standard. The initial NPSNET-1 was demonstrated at ACM's Siggraph 91 conference. NPSNET-2 and -3 focussed mainly on improving the visualisation of the simulation. NPSNET-IV was the most popular version and was compliant with almost all DIS virtual environments and incorporated techniques like dead reckoning to reduce network traffic. NPSNET-V focuses on improving the NPSNET code base by employing component based solutions in the Java programming environment (Capps, McGregor, Brutzman and Zyda, 2000).

The fundamental idea behind NPSNET is to divide the virtual world into logical partitions in order to limit the amount of redundant or unnecessary information on the simulation network. NPSNET associates spatial, temporal and functionally related entity classes with network multicast groups (Macedomia et al., 1995). Multicasting is similar to broadcasting except that only a specific set of recipients receive the messages. This helps to lift the burden of nodes having to read and decode each event on the simulation network.

NPSNET divides the terrain into hexagonal cells (spatial partitioning). Each simulation entity's node only sends state update PDUs to the multicast channel associated with the cell it is currently in and listens for all PDUs in its current cell and all adjacent cells (within a certain radius). This limits the network traffic and also prevents entities from missing PDU's as they move from one cell to another (Macedomia et al., 1995). Functional partitioning can be used to simulate, for example, radio communications since it can stretch over large spatial areas.

The Area Of Interest Manager (AOIM) is responsible for the partitioning of the virtual environment. The AOIM is a software concept while multicasting is a hardware concept. Each node has a AOIM to distribute partitioning processing among hosts (Macedomia et al., 1995).

NPSNET is still very similar to SIMNET in the sense that it sends out events that inherently trigger specific behaviours in the relevant simulation entities. This indicates that NPSNET also has an event-based, implicit invocation architectural style. The combination of implicit invocation and data abstraction can be extremely powerful. It promotes a natural way of component reuse and modification. Components can be modified or swapped with other ones without having to change the relevant interfaces, leaving the rest of the system unchanged (and unaware of any changes) (Shaw and Garlan, 1996).

The disadvantages of the event-based, implicit invocation style largely remain, but if the system is constructed correctly the disadvantages of the data abstraction style can become completely trivial or even disappear from a simulation developer's point of view. The review of the *HLA*, later in this chapter, shows this.

SIMNET, DIS and NPSNET use either broadcasting or multicasting for communication among simulation nodes. Multicasting and broadcasting are not efficient (and quite often not supported) over WAN networks like the internet. This means that SIMNET, DIS and NPSNET will not be able to compete with web-enabled simulation technologies. SIMNET

and NPSNET also only support running objects in real-time. Faster or slower than real-time execution is also not possible since there is no dedicated time management between nodes.

5.3 DIS

The SIMNET protocols went through various revisions and extensions around 1991 and 1992. This led to the Distributed Interactive Simulation (DIS) standard which was made a IEEE industry standard in March 1993 (IEEE 1278-1993 and its successors). DIS is still extensively used in military training simulators (Miller and Thorpe, 1995).

DIS adopted the basic SIMNET Protocol Data Unit (PDU) structure for message transfer, but there are some differences: DIS was planned for larger scale environments and it was decided to move from SIMNET's local flat earth coordinate system to a spherical coordinate system to take the curvature of the earth into account. DIS also uses Euler angles to represent rotation instead of rotation matrices as used in SIMNET PDU's (Miller and Thorpe, 1995).

DIS standardised the SIMNET features and added more features in an effort to provide support for larger scale environments and more advanced training capabilities. The inherent limits of the SIMNET architecture are still present though—DIS has the same architectural limits as SIMNET that prevent it from scaling well to more complex environments.

5.4 HLA

The High Level Architecture (HLA) is the current IEEE standard (IEEE 1516) architectures for distributed simulation systems. HLA and DIS are the two most common military or defence training simulation architectures or technologies in use today.

During the 1990s it became apparent that none of the existing simulation architectures would hold for the ever increasing budget limits and scalability requirements. For a simulation architecture to survive it had to be reusable and appeal to the larger simulation community outside the defence industry. In 1995 investigations into the HLA (High Level Architecture) began with the HLA baseline being approved later on as the standard simulation architecture for all DoD simulations (Kuhl, Weatherly and Dahmann, 1999).

HLA implementations provide a way for standalone simulations, called federates, to collaborate with other federates. The power of the HLA lies in its ability to separate the complexities of the simulation middleware (or Runtime Infrastructure (RTI) in the HLA case) from the federate implementations. The RTI and federates together are called a federation (Kuhl et al., 1999).

The internet age has brought about exciting web-based technologies like XMSF that are extremely portable (cross-language and cross-platform) that enable true cross-platform application development (Pokorny, 2005). Web-based middleware has now become more popular than the traditional middleware architectures like CORBA (Common Object Request Broker Architecture) and RMI (Remote Method Invocation). XMSF (the eXtensible Modelling and Simulation Framework) has successfully been applied to web-enable DoD and non-DoD HLA RTIs. This allows existing HLA federates to collaborate with each other

over the web (web-based simulation) without any changes to the federations being necessary (Morse, Drake and Brunton, 2004). XMSF has however now given way to *HLA Evolved* which will be discussed after the next section.

5.4.1 An Overview of the HLA

The HLA standard only defines a software architecture or set of services with no implementation details specified. The HLA separates the implementation of the simulation infrastructure (RTI) and the simulation-specific federates. By doing this RTIs can be reused for many different simulations and federate developers do not need to bother with the complexities of the simulation infrastructure (Kuhl et al., 1999).

A federate is a single point of attachment to a RTI and could internally represent one or many simulated entities. A federate would normally either be a self-contained simulation or a viewer, database, etcetera. Federates use a common object model called the Federation Object Model (FOM) to communicate with each other (Kuhl et al., 1999). The FOM defines the names of the events that federates can send to the RTI, but does not describe things internal to any federate and is only used to transfer information between federates (Kuhl et al., 1999).

HLA exhibits several architectural styles:

- data abstraction and object-oriented organisation,
- event-based, implicit invocation, and
- layered systems.

In the Data Abstraction and Object-Oriented Organisation style data representations and their associated primitive operations are encapsulated in an abstract data type or object (Shaw and Garlan, 1996). A generic interface is provided by the RTI to a federate and by a federate to the RTI behind which the implementation details of the federate and RTI respectively are hidden from each other. Each federate still needs to know which RTI to join and each RTI still needs to maintain a list of federates, but federates never communicate directly with each other. The advantage of this style is that changing the implementation of either the RTI or federate will not influence the other as long as the interface stays consistent. The disadvantage of this style is that the objects or components interact using explicit procedure calls and each object needs to know the identities of interfaces it needs to call. This is not a problem in the HLA since federates only need to explicitly connect to the RTI.

In the Event-Based, Implicit Invocation Organisation style instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. When an event is announced the system itself notifies interested parties or components (Shaw and Garlan, 1996). Federates do not communicate directly with one another. A federate has to communicate via the RTI and rely on the RTI to invoke it whenever an event in which it has an interest comes through. A federate announcing an event would call the RTI explicitly and then let the infrastructure take care of invoking the relevant interested federates. The advantage of this style is that it supports reuse and eases system evolution. Components can easily be added or swapped with other ones without affecting the relevant interfaces or import/registration structure. A possible disadvantage of this style is that the invoking

component has no control over how events are understood or processed when received by interested federates.

A layered system is organised hierarchically, each layer providing service to the layer above it and serving as a client to the layer below (Shaw and Garlan, 1996). In the HLA the low level networking code and simulation infrastructure is found in the RTI while simulation specific code can be found in the relevant federate. Having the RTI in a separate layer makes it very easy to, for example, transfer the simulation from a small network or single machine to a web-based environment by just swapping out RTIs. One disadvantage of layered systems that affect the HLA is the added complexity or overhead of having to make calls through the layer interfaces (RTI Ambassador and Federate Ambassador shown in Figure 5.2).

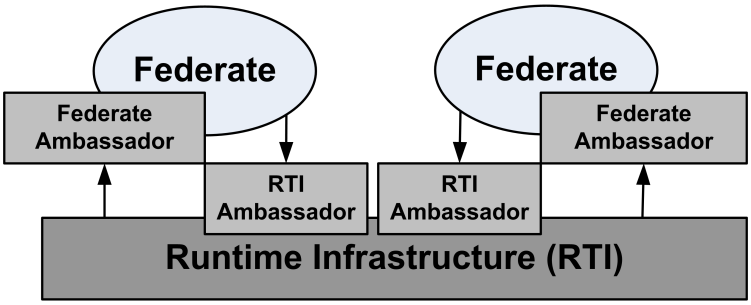

Figure 5.2: The HLA Interface between RTI and Federates

The interfaces are sets of procedure calls that take and return parameters with pre- and post conditions on the calls and with exceptions. Alternatively the *RTI Ambassador* (figure 5.2) can be seen as the interface to the RTI (from a federate's point of view) and the *Federate Ambassador* can be seen as the interface to a federate (from the RTIs point of view). An ambassador is similar to an object with a set of methods representing the interface (Straßburger, 2000).

In addition to the HLA Object Model specification and the HLA interface specification there are also the HLA rules and HLA services that form part of the IEEE standard. The HLA rules are a set of conventions that must be followed to achieve proper interaction of federates. The rules are design principles for the interface specification and object model template. The rules also describe the responsibilities of federates and federation designers. The HLA offers services in six areas or groups. The services are as independent as possible and are listed below (Kuhl et al., 1999):

- federation management,
- declaration management,
- object management,
- ownership management,
- time management, and
- data distribution management.

The services are meant to be independent so that a federate developer who does not need the functions of specific services can ignore those services without undesirable side effects (Kuhl et al., 1999).

5.4.2 An Overview of the HLA Evolved

The XMSF (eXtensible Modelling and Simulation Framework) is not a replacement for HLA and is not meant for developing distributed simulations. XMSF is a series of exemplars and descriptions (called profiles) that are intended to enhance interoperability of distributed simulation using web-based technologies (Pokorny, 2005).

One example of an exemplar employs the Simple Object Access Protocol (SOAP) and the Block Extensible Exchange Protocol (BEEP) to map the HLA RTI API to XML to create a Web-Enabled RTI. The Web-Enabled RTI still maintains a consistent HLA RTI interface to federates. An exemplar can be seen as an example or pattern of implementation for a specific purpose (Morse et al., 2004). A profile is supposed to explain how an exemplar works and how it fits into the M&S domain (Pokorny, 2005). A profile can be defined as (Morse et al., 2004):

- a tailoring of the set of selected standards,
- data and meta-data standards, and
- recommendations and guidelines for implementation.

The Web Enabled RTI has the potential for creating new simulation capabilities that did not exist before. Using open mainstream web-based technologies for distributed simulation has seen the following benefits (Pokorny, 2005):

- really large scale virtual environments now possible,
- multi-language federates working together,
- managing the RTI through open standards like the Jabber (XMPP) instant messaging protocol,
- web-service based simulation clients running on portable devices like PDAs, and
- rapid development of simulation clients like remote 2D/3D viewers with great benefit to the client/stakeholder.

The major technical improvements of *HLA Evolved* (the new version of the HLA) are (Möller, Morse, Lighter, Little and Lutz, 2008):

- modular FOMs and simulation object models (SOMs),
- web services support through the new Web Services Description Language (WSDL) API,
- federate and federation fault tolerance support for handling network errors and unreliable federates,
- smart update rate reduction to reduce network traffic, and
- dynamic link RTI interfaces that allow switching between different RTIs without having to recompile federates.

Existing IEEE 1516 HLA federates can be migrated to *HLA Evolved* (Morse, Lighter, Lutz, Saunders, Little, Möller and Scrudder, 2005), but this includes changing the federates to use the updated API and data types. The new HLA functionality should make it easier to develop (or extend) and deploy high-quality federates (Möller et al., 2008).

5.5 VGD

VGD is a Virtual GBADS (Ground Based Air Defence Simulator) Demonstrator developed by the CSIR DPSS[2]. VGD represents a modelling and simulation capability for acquisition decision support and concurrent tactical doctrine development. VGD has gone through several versions (figure 5.3).

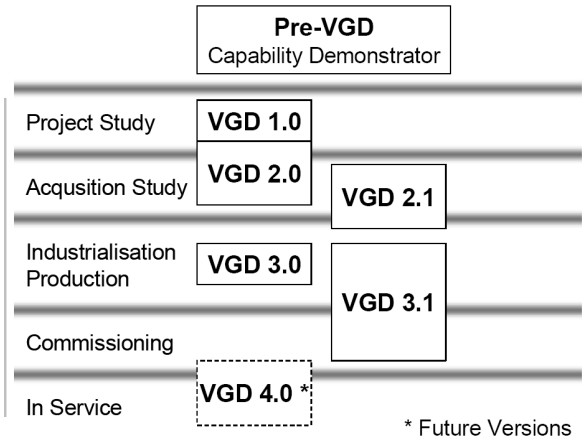

Figure 5.3: The VGD Versions and Acquisition Phases
Taken from (le Roux, 2006), with permission.

5.5.1 VGD 2

VGD 2 (developed in 2000 to 2002) extended the original VGD to include environmental information like terrain elevations and made reusable high fidelity simulation models possible. The VGD 2 architecture also made features like OIL (Operator In the Loop) and HIL (Hardware In the Loop) possible.

Figure 5.4 shows the architecture of VGD 2. It is essentially an HLA Federation with G2[3], STAGE[4], Simulator, C++ and MATLAB federates. The Simulator Manager federate provides time management services to the other federates in VGD 2. The ModIOS package provides 3D visualisation of the simulation entities and is developed by Motorola. HLA wrappers are used to transform C++ and MATLAB models as well as OIL and HIL equipment and software into federates compatible with the rest of the federation. HeliSim and FLSim are high fidelity modelling applications developed by Virtual Prototypes Inc (le Roux, 2002).

[2]DPSS (Defence Peace Safety and Security) is a division of the CSIR (Council for Industrial and Scientific Research) South Africa.
[3]G2 is a real-time expert system developed by Gensym Corporation.
[4]STAGE is a product of Virtual Prototypes Inc and provides an environment for high fidelity models and environmental entities. It can be used for scenario planning and 2D visualization.

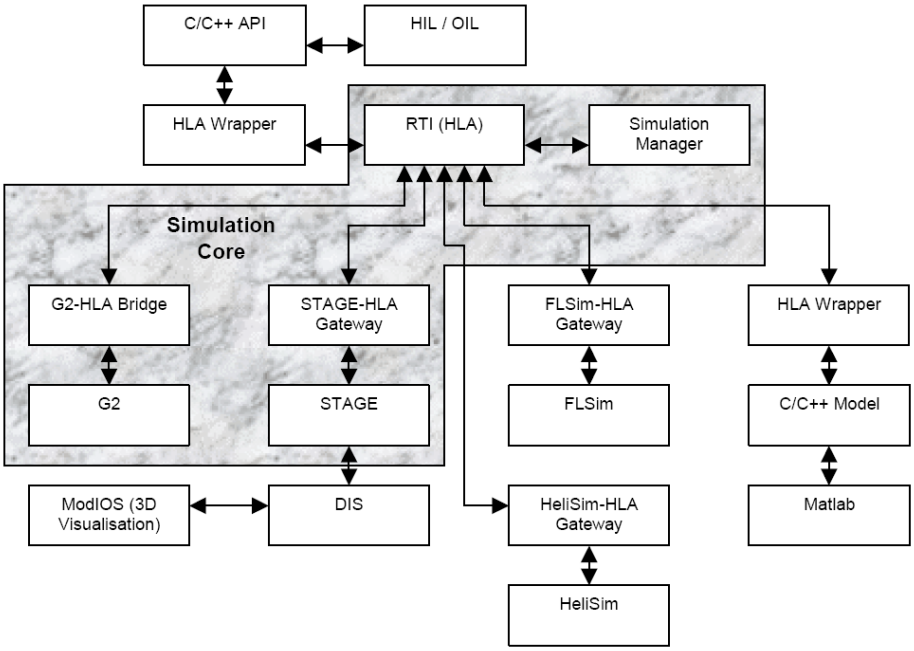

Figure 5.4: The VGD 2.0 Architecture
Taken from (le Roux, 2002), with permission.

5.5.2 VGD 3.0

VGD 3.0 (developed in 2002 to 2004) has a less complex distributed architecture than VGD 2. VGD 3.0 uses a client/server type architecture and does not use the HLA as VGD 2.0 does. VGD 3.0 uses a lightweight architecture instead of the HLA to reduce simulation overheads. This is because the focus shifted towards batch processing and statistical analysis. The performance benefits of running distributed over multiple nodes were not required (Duvenhage and Senekal, 2004).

Figure 5.5 shows a deployment diagram of VGD 3.0. The simulation can run on one or two machines as shown. The VGD 3.0 architecture has three primary components (Duvenhage and Senekal, 2004):

- the Entity Model Server (EMS),
- the Air Defence Control (ADC), and
- the ADC Communications Server (ACS).

The Entity Model Server (EMS) is a constructive simulation with optional virtual simulation capabilities. As mentioned in Chapter 1 constructive simulation contains only computer models where a virtual simulation can also contain operator-in-the-loop (OIL) capabilities. The EMS maintains all the equipment-related models and threat models. Additionally the EMS provides the required natural environment services to the entity models and also maintains the connections to the various consoles and viewers (Duvenhage and Senekal, 2004).

The Air Defence Control (ADC) component determines the behaviour of the simulated

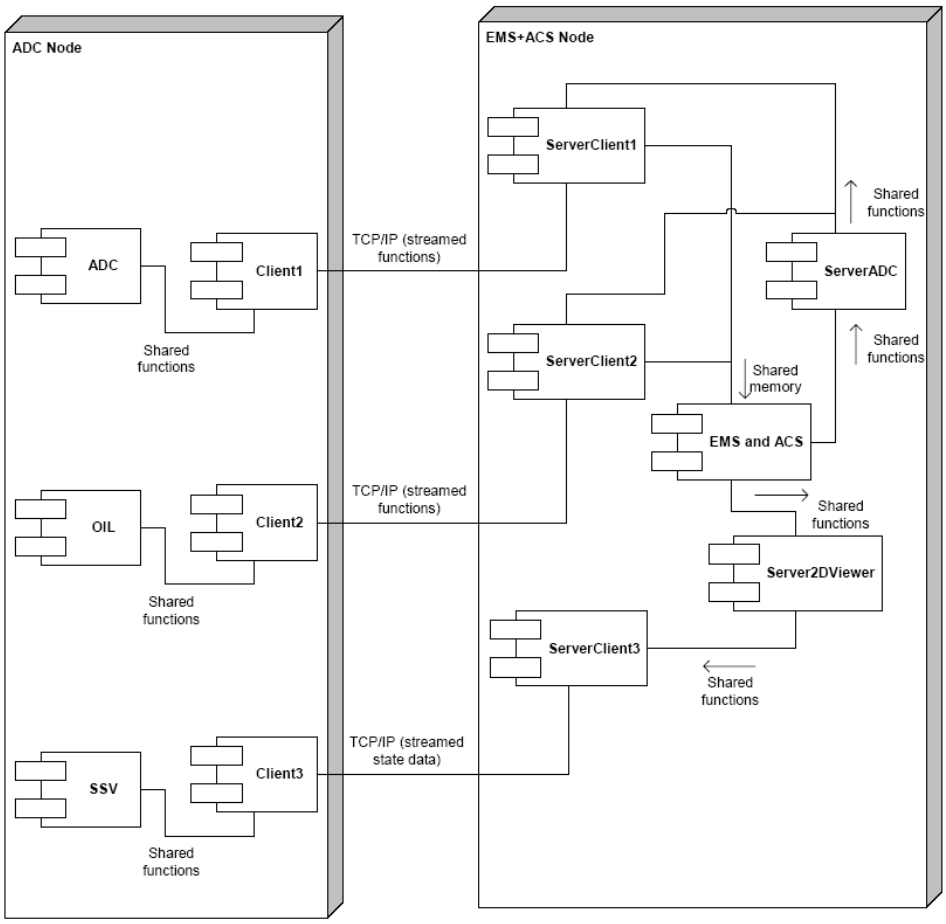

Figure 5.5: The VGD 3.0 Architecture
Taken from (Duvenhage and Senekal, 2004), with permission.

operators of the simulated equipment. From the EMS's point of view, operators modelled by the ADC are indistinguishable from human operators connected via OIL consoles. The ADC normally runs on a different machine to the EMS and connects via a TCP link (Duvenhage and Senekal, 2004).

The ADC Communications Server (ACS) simulates radio communications between operators. It is integrated with the EMS (Duvenhage and Senekal, 2004).

There are optional viewers and consoles that change the simulation from a constructive to a virtual one by allowing humans to replace some of the operator models that control simulated equipment. The OIL (Operator In the Loop) console can replace computer models in the ADC with real human operators without the EMS knowing the difference. The Simulation Scenario Viewer (SSV) is an online 2D viewer of the EMS scenario with a perfect view of the simulated environment (Duvenhage and Senekal, 2004).

The ServerADC allows multiple ADC and OIL clients to connect to the EMS and ADC. Each client connects via a ServerClient interface directly to the EMS and ACS. The Server2DViewer allows multiple SSV clients to connect to the EMS (Duvenhage and Senekal, 2004).

The VGD 3.0 architecture does not scale well to large scenarios since the simulation models

all run on one machine (except the operator models that can run on a different machine—in the ADC component). This prevents VGD 3.0 from running complex scenarios in real-time (batch processing stays possible, but at slower than real-time speeds).

5.5.3 VGD 3.1

VGD 3.0 does not support real-time execution of complex scenarios since all the models run on one or two machines. VGD 3.1 (developed 2004 to 2008) has a lightweight distributed simulation architecture to be able to run complex scenarios in real-time, but still maintains an efficient batch running capability if all models run on one host. VGD 3.1 allows for connection of human operators (OIL), 2D/3D viewers and external sensors (HIL) through consoles or gateways provided by the simulation. Simulation services like terrain and line of sight calculations are also part of VGD3.1 (Duvenhage and le Roux, 2007b). The entity models and services are distributed over all available nodes for the best performance of complex scenarios.

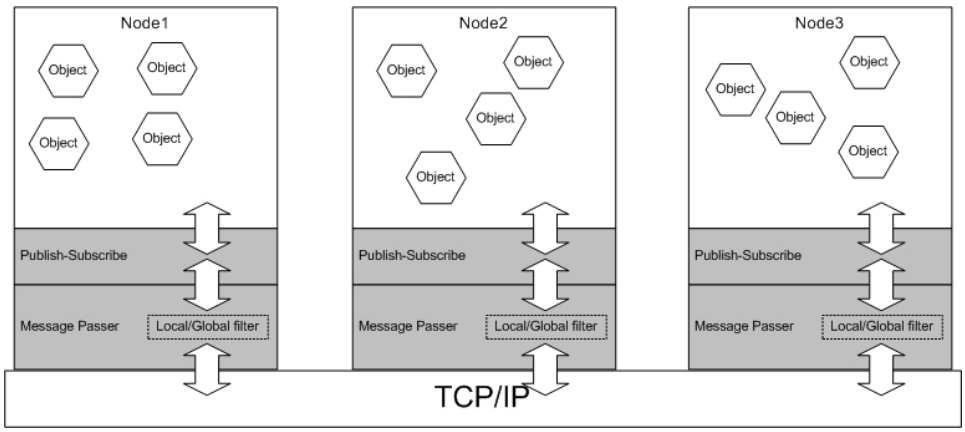

Figure 5.6: The VGD 3.1 Architecture
Taken from (Duvenhage and le Roux, 2007b), with permission.

The VGD 3.1 simulation backbone uses a publish-subscribe type infrastructure. This provides time, space and synchronisation decoupling (Eugster et al., 2003) between nodes and also between communicating models, making the VGD 3.1 architecture very scalable. The time decoupling is limited though since simulation nodes have to run in a lock-step mode for the hosts to stay synchronised. The backbone infrastructure runs distributed with each node being intelligent enough to either service its own models or forward events or states to the correct node. Models wishing to be notified of states or events from a specific model have to make the appropriate subscription to that model.

VGD 3.1 uses a topic-based subscription style. Subscriptions are made using a category keyword (topic) and title name (model or object id). Wildcard title names (only '*' currently implemented) can also be used if events from all objects are of interest.

VGD 3.1 also supports tactical communications simulation through the addition of a very basic communications framework. The backbone supports delayed message delivery that makes it possible for the communications framework (integrated into the simulation infrastructure) to delay or not deliver messages at all. This decision is based on the results

from simple radio and link models and line of sight queries (Duvenhage and le Roux, 2007b).

5.5.4 Migrating to a Quantised Discrete Event Architecture

VGD 3.1 follows a Discrete Time System Specification (DTSS) for modelling. It is proposed to update the VGD 3.1 backbone architecture to a hybrid architecture that contains some Discrete Event System Specification (DEVS) modelling elements (Duvenhage and Kourie, 2008). This hybrid DTSS/DEVS modelling lowers communication bandwidth between models and consequently between nodes, increasing performance and making the simulation more scalable.

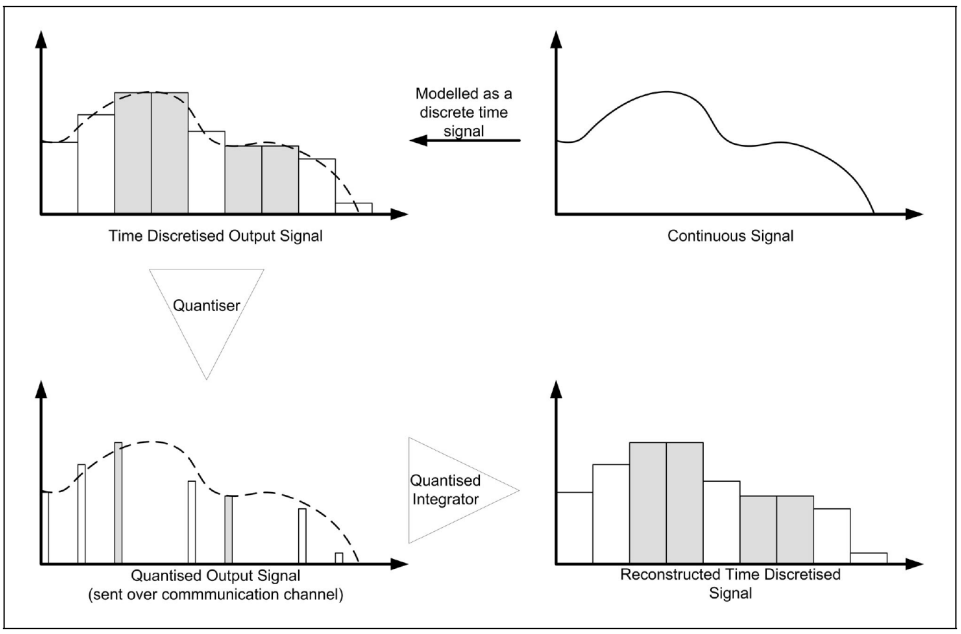

Figure 5.7: The State Quantisation and Integration
Taken from (Duvenhage and Kourie, 2008), with permission.

The hybrid modelling approach is referred to as Quantised DEVS and existing DTSS models can be wrapped with quantiser and quantised integrator pairs to make them compatible with the Quantised DEVS architecture (Duvenhage and Kourie, 2008). Model state quantisation can be done with techniques similar to dead-reckoning where a prediction/error-estimation algorithm is used to quantify the state of the model with state updates sent only if the model's state changed.

An alternative to using dead-reckoning for model state quantisation is suggested in (Duvenhage and Duvenhage, 2008). This approach uses an algorithm developed for *live aircraft engagement* to do the model state quantised integration. The *live aircraft engagement* algorithm provides a simpler approach and its application has been very successful (Duvenhage and le Roux, 2007a). The algorithm is however less formal and not proven to be more effective than dead-reckoning in this case.

The next chapter will help the reader understand what the current trends in Joint Command and Control (JC2) are and also help the reader understand what the C2 Enterprise would

look like.

6. System Interoperability

Modelling and simulation (M&S) applications can form the basis for planning and decision support tools and can also assist with the processing required for the visualisation and presentation of information. This chapter reviews the current trends in Joint Command and Control (JC2) and helps the reader understand what the C2 Enterprise looks like.

6.1 System Interoperability and Joint Command and Control

Interoperability implies the ability of systems to provide services to other systems as well as use services from other systems. Joint Command and Control (JC2) further implies the use of aggregates of existing C2 systems. The JC2 capability relies on software to integrate the increasing number of existing C2 systems into the JC2 environment in a robust manner (Daly and Tolk, 2003).

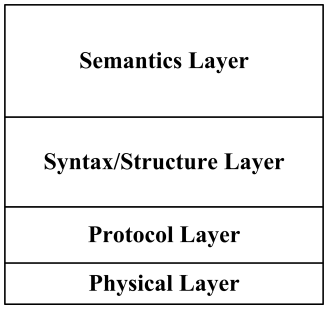

Figure 6.1: The Interoperability Layers

Figure 6.1 gives a layered view of system interoperability. The *physical* and *protocol* layers provide the connectivity mechanisms, for example TCP/IP over Ethernet, that determine the network bandwidth and link quality. The *structure* layer represents the higher level data structures or syntax of the information defined by the relevant communications protocols or data-link standards. The *semantics* layer represents the *understanding* of the data at the command level. *Network-Centric* operations require the sharing of situational awareness information as well as operational context type information, which includes objectives, plans, orders and priorities—information sharing can be at any level (Chaum and Lee, 2008).

The Multinational Information Sharing (MNIS) requirements are identified as (Chaum and Lee, 2008):

- It should be possible to share, collaborate on, or synchronise information with partners to be able to make decisions with partners.
- It should be possible to interoperate with and use partners' systems.
- It should be possible to extend own capabilities into the partner environments as well as utilise capabilities from partners.

6.1.1 Tactical Networks

Tactical Data Links (TDLs) are used for the exchange of tactical command and control data and near real-time exchange of situational awareness information. *TDL* standards often include the whole protocol stack to support encryption, jamming resistance, etc. No information history is kept in TDL networks. Only the most recent information is considered to be relevant. TDLs have the advantage of near real-time information exchange and resistance to jamming, but have the disadvantage of requiring expensive hardware to implement the relevant protocol stacks. The hardware also lacks modularity (Larsen, 2006). TDLs can operate on low-bandwidth ad-hoc networks and are well suited to tactical environments (Crane, Campbell and Scannell, 2008).

Systems can communicate with each other using specific sets of messages. *Messaging* standards usually specify a fixed structure and syntax for the messages. The Message Text Formats (MTFs) used by US and NATO systems are text based, but the contents of the messages are fixed to be validated easily (for example, XML based message standards can be validated using the relevant XML schemas). Messaging has the advantage of loose coupling, but may be non-real time and may require man-in-the-loop operation if the message processing cannot be automated (Larsen, 2006).

Systems interoperability can be achieved by adapting all the relevant systems to support a very specific interface to communicate via a set of web-services. *Web-services* have the ability to easily integrate with other web-services using open standards like the Simple Object Access Protocol (SOAP). Web-services are self-descriptive with a standardised interface for establishing connectivity and transferring data. This approach was used to integrate systems from America, Spain, Germany, Sweden and France, but the level of interoperability was limited by the latency and bandwidth of the internet (Daly and Tolk, 2003). *Enterprise Services* and the *Enterprise Service Bus* (discussed in the next section) are more formal ways of using web-services to integrate application services.

Interoperability through a database or central repository has the advantages of being modular with near real-time information exchange, but has the disadvantages of closer coupling between systems and a increasing data size because the database normally retains a history of all the information (Larsen, 2006). Collaboration is likely to be more successful and efficient when the participants have a common understanding of the shared information. Establishing a Community of Interest (COI) helps to have consensus on operational processes, activities, and the supporting data standards. A COI is a collaborative group of users who exchange information in pursuit of their shared goals, interests, missions or business processes (Larsen, 2006). The Joint Consultation, Command and Control Information Exchange Data Model (JC3IEDM) from the Multilateral Interoperability Program (MIP) is a shared multinational C2 data model that enables network-centric information sharing among different COIs (Chaum and Lee, 2008).

6.1.2 C2/M&S Protocol Gateways

The simulation capability of the CSIR DPSS is continually being extended for Joint Command and Control (JC2). This was done to such an extent that a separate application was required to manage all the different types of protocols, interfaces and link standards used to connect external systems to the simulation. This application can be called a C2 *gateway* or *protocol bridge*, since it translates different link standards or protocols to the native data format of the simulation. A gateway can provide the following (Duvenhage and Terblanche, 2008):

- time management and synchronisation,
- time stamping of data from external systems,
- translation between different data models,
- information storage to allow for partial or incomplete updates from external systems,
- fault tolerance, and
- easy access to the functionality of commercial packages like 3D game engines for visualisation, etcetera.

Each gateway operates as a bridge between an external system and the simulation and allows the simulation to participate in live C2 interoperability and training exercises. Multiple gateways can also be linked together which allow external systems to interoperate through the gateways (Duvenhage and Terblanche, 2008). The shared information allows simulations to collectively create a more complex simulation. A simulation can, for example, operate on, engage or kill models from other simulations with all relevant events from one simulation being reflected in the other participating simulations (Nel, le Roux, van der Schyf and Mostert, 2007).

The HLA federates use *adaptors* to bridge the gap between different versions and brands of RTIs or to translate between different FOMs. This is required because HLA federates created for different RTIs or HLA federates using different FOMs are not compatible. An adaptor is essentially a software gateway between a federate and the relevant RTI and handles the required syntactic and semantic transformations. Adaptors also allow non-HLA federates or legacy systems to become part of a federation (Moller and Olsson, 2004). Using adaptors are and important part of HLA federations and can extend the life or enhance the reusability of existing federates. Harless and Roos (Harless and Roose, 1999) recommend using gateways as part of the HLA interoperability tool suite to prolong the life of legacy HLA systems, to interoperate with non-HLA systems and to integrate dissimilar federations.

Saab Systems have also developed an integration platform called the Widely Integrated System Environment (WISE). WISE can connect to any external system if a custom WISE platform driver for that system is available or can be created. The WISE platform driver connects to the external system and translates the information model used by the external system to the information model used internally by WISE. Multiple external systems can be integrated using WISE if the relevant drivers are available or can be created. The information models used by the external systems should also map well to the information model used internally by WISE for the integration to be successful. The internal WISE information model is specified using a XML based format and platform drivers can be implemented using C++ (Olsson and Michalski, 2008).

6.2 The Command and Control Enterprise

Enterprise systems are created from existing systems by loosely coupling the systems together in an ad-hoc fashion or through *middleware*. The capabilities or scalability of the aggregated system depends on the middleware architecture. The C2 Enterprise also requires existing legacy systems to interoperate, which can be hard to achieve. *Scenario Definition Languages* and *Battle Management Languages* may however help to initialise the state and behaviour of the enterprise systems and simulations (Hamilton and Catania, 2003).

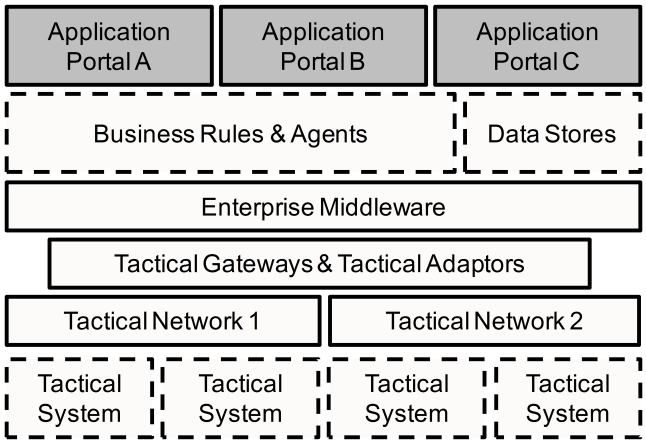

Figure 6.2: The Tactical Network and C2 Enterprise

Enterprise systems operate in a non-realtime fashion, but information can be provided in a timely manner that is still suited to JC2. The C2 enterprise can however not replace the existing tactical networks, since TDLs, messaging, etcetera. have superior bandwidth and security capabilities. Tactical systems (including *Live*, *Virtual* and *Constructive* simulations) integrate with enterprise systems through gateways or adaptors. The C2 enterprise will then actually consist of two layers (see Figure 6.2): the tactical network layer and the business or enterprise layer. The tactical network provides connectivity to tactical systems.

6.2.1 Enterprise Services

Enterprise Services refers to high-level cross-entity web-services-based components with (Daly and Tolk, 2003):

- loose coupling,
- broad application,
- inherent security,
- availability and reliability,
- reduced cost and complexity, and
- increased levels of interoperability and information sharing.

Web-services transfer information at the *syntatic* level and the higher level semantics still need to be agreed upon when constructing the enterprise (Larsen, 2006). A net-centric enterprise,

utilising an SOA, is not ideally suited for the tactical environment because of the bandwidth and connectivity limitations of tactical networks. The enterprise can however provide and consume tactical data through *tactical services* that act as gateways between legacy protocols and the net-centric enterprise (Crane et al., 2008).

6.2.2 The Enterprise Service Bus

An Enterprise Service Bus (ESB) can create an enterprise with a heterogeneous set of application servers from different vendors, built with different technologies and using different communication protocols—something that is not possible with a traditional SOA design. All the applications communicate through middleware with no direct contact between the applications. An ESB supports *service-oriented architectures*, *message-driven architectures* and *event-driven architectures* to be able to support all the interaction patterns that are required in a comprehensive enterprise (Keen, Acharya, Bishop, Hopkins, Milinski, Nott, Robinson, Adams and Verschueren, 2004b). ESBs provides an abstraction layer on top of some form of Message Oriented Middleware (MOM) and is not specific to web-services. An ESB usually supports the following (Schulte, 2002):

- the Simple Object Access Protocol (SOAP),
- the Web-Services Description Language (WSDL),
- the Universal Description Discovery and Integration (UDDI),
- asynchronous store-and-forward delivery,
- limited message transformation, publish-and-subscribe, and
- content based routing.

Open ESB is a Java based open source ESB implementation and is built purely on open standards. Using open standards imply that applications and services from different vendors can be moved freely between different enterprises.

This now concludes the second part of this dissertation. The next part discusses the implementation of the proposed software application framework.

FRAMEWORK IMPLEMENTATION

This part of the dissertation defines the requirements for the software framework based on existing experience within the Command and Control domain. The focus quickly shifts to the framework design and implementation. The framework implementation is then evaluated and the evaluation covers enough aspects of the implementation to address how well the design fits the original requirements. Critical components of the software framework design are formally described and evaluated. This is done using the software architecture concepts and behaviour analysis techniques reviewed in the previous part of this dissertation.

7. Framework Requirements

This chapter marks the start of the third part of this dissertation, which discusses the design and implementation of the proposed simulation software framework. The *use case* for the software framework is created based on existing experience with M&S as well as the requirements, identified in the previous part of this dissertation, for supporting the C2 Enterprise. The *use case* is then used to generate the requirements for the framework implementation.

7.1 Framework Use Case

Figure 7.1 shows a simple *use case* diagram for the software framework. There are five different actors defined for the framework: the software developer, the operator, the C2 system, the LVC simulator(s) and the user or stakeholder. The developer has to be able to extend the framework and use it to develop applications according to the user's requirements. The operator has to be able to use the developer's tools and applications within the C2 enterprise. The user may also be the operator.

The use case shows that the developer can also be the operator, but this is mainly for testing and evaluation. The operator is responsible for running the application and possibly managing the simulation execution or the links to external systems. The operator could also form part of the simulation in the case of virtual simulations or human-in-the-loop (HIL) type applications. The operator might require additional feedback or insight through a debriefing capability provided by the application. For this the application should be able to capture the relevant information and present it in a sensible way.

Developers have several tasks: they have to maintain the framework and ensure that the code-base is of a high enough quality; they have to build and test the application(s); they have to extend the framework to accommodate new features; and, they have to build the relevant models of systems in order to virtualise them.

The external C2 systems and LVC simulators can be anything from real equipment like search radars, aircraft and ships, to simulators used for training and/or planning.

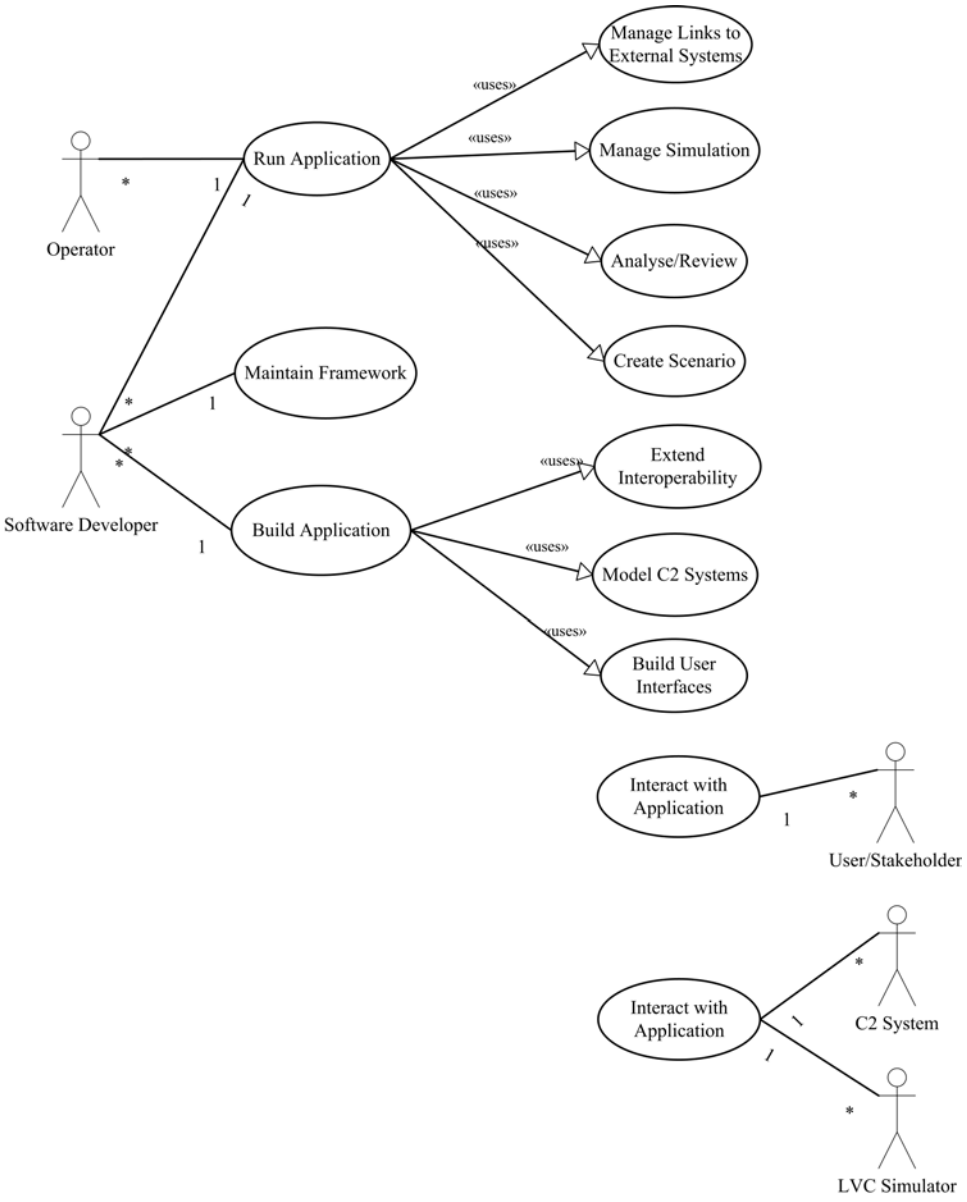

Figure 7.1: Framework Use Case Diagram

7.2 Framework Requirements

The software framework requirements are generated from the *use case* diagram. The focus is placed on supporting the tactical networks that feed data into and out of the C2 enterprise. The interoperability, M&S and application development capabilities provided by the framework supports the enterprise by enabling the integration of the tactical layer with the enterprise business layer and by filling any gaps in the tactical layer.

The knowledge gained during the literature review, presented in the previous part of this dissertation, also contributed to formulating the framework's requirements. These requirements can be divided into five main points: interoperability with C2 systems, virtualisation of C2 equipment using M&S, application development, good code quality and

performance and portability. Figure 6.2 in Chapter 6 shows how the framework capabilities should fill the gap between the C2 Enterprise and the operational tactical systems.

7.2.1 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 systems and simulators,
- automatic attribute translation when translating to and from external data representations,
- unified and extendable internal information model,
- protocol bridging (acting as an adaptor or gateway for systems that do not support the correct protocol or interface).

7.2.2 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,
- 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 and fault tolerance.

7.2.3 Application Development

The users and operators require a means of interacting with the virtual environment or controlling the system interoperability. This interaction could be through anything from a text console to a full Graphical User Interface (GUI). The framework should provide the software developer with a concise way of integrating with user interfaces and applications. The framework should also define how the application should interact with the underlying virtualisation and interoperability capabilities of the framework.

7.2.4 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 determine how the framework is used. The framework code base should be characterised by the following:

- a common interoperability infrastructure across the tactical environment,
- a common M&S infrastructure across the tactical environment,
- seamless integration of the M&S and interoperability infrastructures.
- unified data collection and analysis,
- fault-tolerance and reliability,
- usability, maintainability and extensibility

7.2.5 Performance and Portability

Real-time performance is desired, but in most cases this is actually *soft* real-time since the targeted operating systems do not support *hard* real-time. Parallel execution (distribution over multiple CPU/Cores on one node) should also be considered to utilise the power of the new generation of multi-core processors. There are cases where interoperability is required with equipment and systems that might have very strict timing requirements: this requires that some components of the framework have to be run in separate high priority threads to achieve the desired execution speed and reliability.

Ultimately the framework should make it easy for application developers to create good quality applications and tools that support the C2 Enterprise. The framework design and implementation should not be specific to any platform or operating system. The next chapter discusses the current framework design and implementation.

8. Framework Design and Implementation

This chapter discusses the design and implementation of the software application framework. The framework architecture is also described using the various software architecture concepts and terms reviewed in part two of this dissertation.

8.1 Design Overview

The framework is split into five functional layers: the *backbone* layer, the *infrastructure* layer, the *interoperability* layer, the *simulation* layer and the *application* layer. Figure 8.1 shows the layers of the framework mapped onto specific layers of the OSI model to give some perspective on the functionality of each layer.

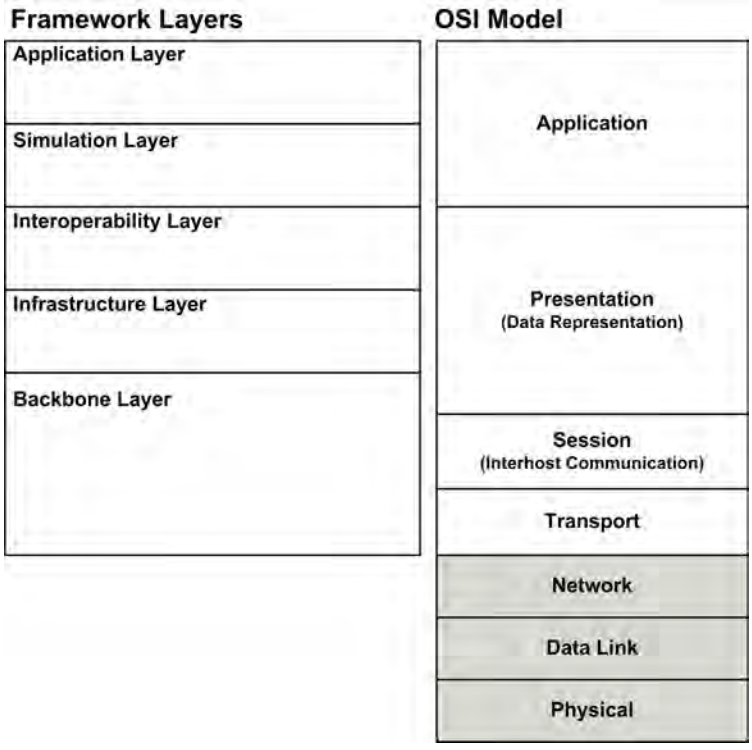

Figure 8.1: The Framework Layers

The layers are implemented as separate C++ libraries. There is a clear separation between the layers in terms of functionality and each layer can be compiled or modified without affecting the other layers. This layered architecture accommodates multiple teams of software developers either working on different layers of the framework or building different applications using the framework.

The work discussed in this dissertation covers all the layers depicted in Figure 8.1, but the framework implementation focuses on the *backbone*, *infrastructure* and *interoperability* layers. The *simulation* and *application* layers are more application specific, with some examples given in the next chapter. This chapter does however discuss the *simulation* and *application* layers in enough detail to show how these layers should be extended and how the framework interfaces with higher level user applications.

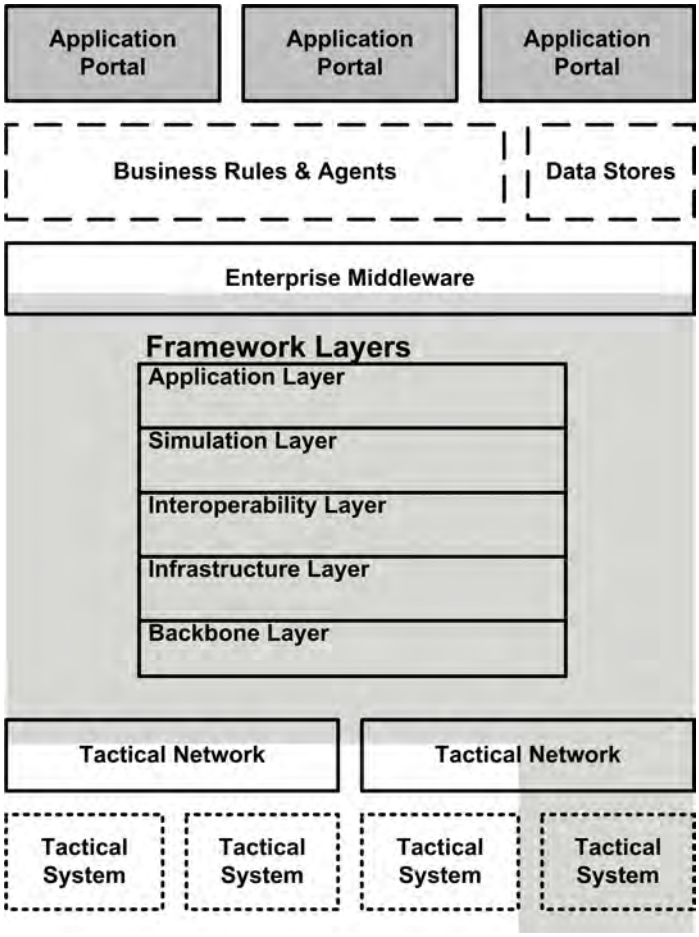

Figure 8.2: The Framework Supporting the C2 Enterprise

Figure 6.2 in Chapter 6 shows that the C2 enterprise consists of a tactical network layer and a enterprise layer—two distinct layers. Figure 8.2 shows how the framework supports the integration of the tactical layer with the enterprise business layer (the framework enables *interoperability with C2 systems*). The framework also helps fill any gaps in the tactical layer through *virtualisation of C2/tactical equipment using M&S*.

The use of proven cross-platform libraries improves the quality and usability of the applications created with the framework. The framework and current application implementations

make use of the following open-source or standardised libraries:

- The framework relies extensively on the *Standard Template Libraries* (STL) for C++ for data structures, containers, sorting, etcetera. STL is proven, well documented and comes standard with C++.
- The framework also relies on some of the *Boost* C++ libraries (*http://www.boost.org*) for things like multi-threading, string hashing and random number generation. *Boost* is a free cross-platform set of libraries with some of the libraries being considered for inclusion into the new C++ standard (currently being finalised).
- The applications created with the framework make use of user interface technologies like wxWidgets (*http://www.wxwindows.org*) and Qt (*http://qt.nokia.com/products*) for building the relevant user interfaces. *wxWidgets* and *Qt* are cross-platform and open-source—either can be used to create good user interfaces for applications.
- The applications use *Open Scene Graph* (OSG, *http://www.openscenegraph.org*) to create 2D and 3D visualisation panels in the user interfaces. OSG is a cross-platform, open-source 3D graphics toolkit.
- The framework builds on *libxml* (http://xmlsoft.org) to create a simple object-oriented XML reading and writing capability. *libxml* is a cross-platform XML processing library for C.

The remainder of this chapter discusses the five layers of the framework.

8.2 The Backbone Layer

The *backbone* layer provides Inter Process Communication (IPC) and object execution (i.e. functionally it corresponds to the OSI Transport and Session layers, as well as in part to the Presentation layer, as shown in Figure 8.1). The backbone also contains a set of core components that contribute to the portability and quality of the framework.

The executable objects are commonly referred to as *backbone objects*. The backbone object execution can be distributed among multiple hosts or among multiple CPUs of a single host (or both) (see Figure 8.3). The process responsible for managing a set of backbone objects is referred to as a *backbone node*. Data flows from one backbone object to another in the form of *issues*, where issues encapsulate events which are called *titles*.

The backbone is loosely based on the work described in (Duvenhage and Kourie, 2008): objects communicate with each other using topic-based publish/subscribe-type message passing (Eugster et al., 2003). The backbone architecture can also be described as having an *event-based, implicit invocation* style, since the backbone objects exchange data through the backbone and do not access each other directly (see Chapter 3). In this context the backbone layer follows the *mediator* design pattern, since backbone objects only communicate through the backbone and not directly with each other (see Chapter 3).

8.2.1 Inter Object Communication

What issues a backbone object can publish and where the issues go are determined by the publications which backbone objects register and by the subscriptions which backbone objects

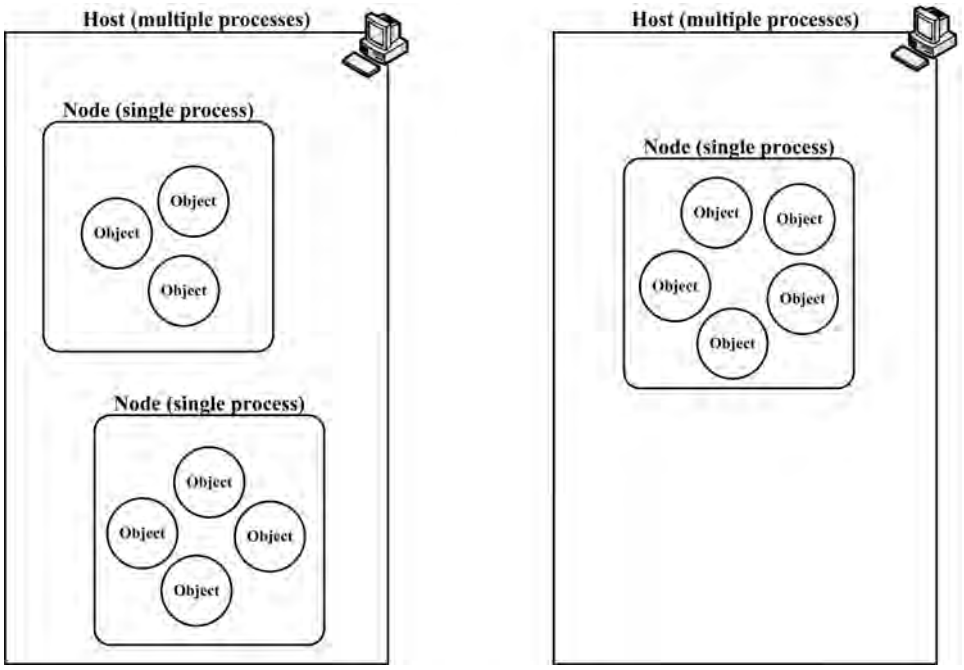

Figure 8.3: The Hosts, Nodes and Objects of the Framework

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

- The backbone generates a publication issue whenever an object registers a publication. The issue is then broadcast to all objects on all nodes.
- The backbone generates a subscription issue whenever an object registers a subscription. The issue is then broadcast to all objects on all nodes. Any object that has a matching publication then processes the subscription.
- The backbone generates a subscription issue in response to the delivery of a publication issue if there is a relevant subscription. The subscription issue is then sent to the publisher.

This passing of issues allows backbone objects to register and delete publications and to add and remove subscriptions to other titles in an *ad-hoc* fashion (i.e. during runtime). Registering a publication will trigger a subscription issue from all objects that have an interest in the publication. Making a subscription will create a subscription issue that is processed by all the objects that have the relevant publication registered.

The title interface contains methods for streaming and de-streaming the title attributes to and from a binary stream. Title objects are automatically identified and constructed (this is explained in the next part of this section) and are 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.

8.2.2 Inherent Object Construction

The built in runtime type information of C++ is not good enough to uniquely identify the type of objects. Because of this the backbone includes its own type system, referred to the *object hierarchy*. The backbone layer also contains an *object factory* that can automatically construct any class within the *object hierarchy*.

A class can be added to the *object hierarchy* by inheriting from a specific interface and including the relevant class members These class members are then used to identify classes within the object hierarchy. Classes are identified within the hierarchy in two ways: based on the class name (string value) or based on the hash value of the class name (for faster identification).

The object type information also indicates its parent type which allows an object to be identified by the object's type as well as by the type of any one of the object's parents within the object hierarchy. Operations to check the type of objects as well as perform safe casting between types are available as part of the backbone.

The *object factory* follows the *factory method* design pattern (see Chapter 3) and enables automatic construction of objects based on its type (as defined in the object hierarchy). Titles and backbone objects are part of the object hierarchy. The *object factory* and *object hierarchy* make it possible to identify and construct titles with the correct type when reading data from other nodes—other layers of the framework as well as applications can add new object and title types without having to explicitly specify them in the backbone layer.

8.2.3 Distributed Object Execution

The backbone runs at a fixed frame rate which determines the simulation time step size. The backbone calls an object to give it time to update itself and read and publish issues. Each backbone object has a very simple interface that is called by the backbone. Not all objects would have to run on every frame and for this an object can specify a *trigger-frame* which specifies the intervals at which the object should be called.

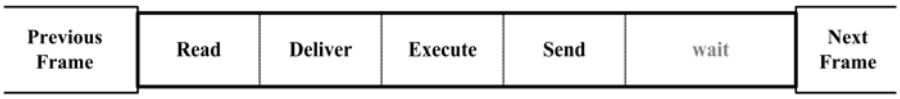

Figure 8.4: The Backbone Object Execution Frame

The backbone executes each simulation frame in five steps with conservative (or lock-step) time management between nodes (see Figure 8.4).

1. The backbone receives all the issues that were published in the previous frame. The backbone keeps on reading issues until all nodes are finished with the previous frame.
2. The backbone then delivers the issues to the correct backbone objects.
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. Newly published issues are temporarily stored in the backbone.

4. The backbone then sends out all the 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 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 node also uses the hub interface to signal the end of its current frame and then to wait for all the other nodes. The application should then wait for the proper period of time before starting a new frame to keep to the relevant object execution frame rate (see Figure 8.4).

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 objects can be executed concurrently within a frame. For this the backbone divides the object execution among several worker threads to better utilise the potential of multi-processor systems or multi-core CPUs.

The worker threads execute objects concurrently (and independently). Each worker thread has a fixed set of backbone objects it executes. Each worker thread is also responsible for delivering the relevant issues that were published in the previous frame to its set of backbone objects. The worker threads all use the same set of delivered titles since these titles are not modified by the backbone objects.

There might be multiple hub implementations in the framework and it is assumed that the hub implementations are not thread safe (i.e. allowing multiple threads to access the hub at the same time would result in undefined behaviour or errors)—this is to make it easier to implement new hubs. For this reason, one of the worker threads have to execute its objects and then wait for all the other worker threads before sending all the published issues to the *hub*. This worker thread is also referred to as the main thread.

8.2.4 Subscriptions and Publications

The backbone manages publications and subscriptions as queues of titles. A backbone object can push data onto the back of its publications and the backbone would pop the data off the front of the object's publications when the object execution has finished. The backbone also pushes titles onto the back of an object's subscriptions and an object could then pop the titles off the front of its subscriptions. Each backbone object has direct access to all the publications and subscriptions it has registered. The subscription and publication classes have specific interfaces that allow the backbone object to push titles to publications and pop titles from subscriptions.

The publication and subscription interfaces do not however accommodate quantisation of states or integration of events (see *QDEVS* at the end of Chapter 5). For this the backbone layer includes several template functions that control how titles are pushed to publications or popped from subscriptions. This enables title quantisation at the publisher and title integration at the subscriber, which makes things like *dead-reckoning* possible. The template

functions, combined with the object type casting and checking provided by the backbone, control how sets of titles are updated and published and then rebuilt at the subscriber.

8.2.5 Core Backbone Components

The backbone contains a set of core components that contribute to the portability and quality of the framework. The backbone abstracts aspects such as multi-threading, memory management and networking to be operating system independent. This means that the framework can run on any operating system for which the core components have been implemented.

Multi-threading abstractions are implemented in the backbone using one of the *boost* cross-platform libraries. The framework also makes use of *Boost* mutexes and barriers for thread synchronisation. *Scoped locks* are used extensively throughout the framework to make it easier for the developer to manage the locking and unlocking of resources (see Chapter 3). Process-control abstractions also make it possible to control operating system specific things like thread-priority, thread affinity and process-priority in a operating system independent way.

The backbone includes a custom memory manager that helps track down memory leaks. The memory manager is created as a *singleton* (see Chapter 3). All objects in the backbone object hierarchy inherit from a base class that has 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 time examine the registered memory allocations. Doing this when a application exits provides the location in the source code of memory allocations that were never de-allocated (i.e. potential memory leaks).

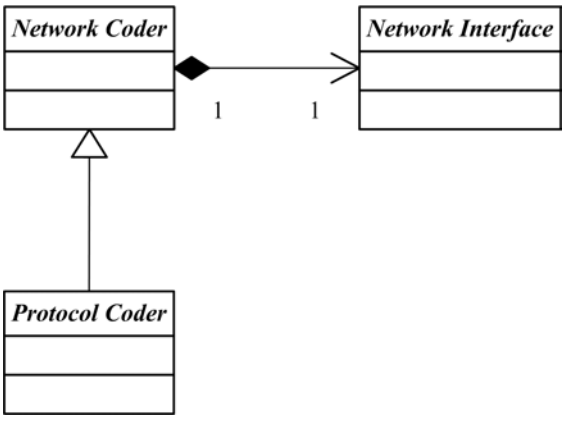

Figure 8.5: The Backbone Network Interface Classes

The backbone provides networking classes that are divided into *network interfaces* and *network coders* (see Figure 8.5). A network interface does the low-level reading and writing of binary data from various interfaces like files, network transport interfaces and even hardware interfaces like RS232. A network coder is a wrapper for a network interface and is responsible for translating the binary data of the interface to titles when receiving data and from titles when transmitting data.

The network interface classes all present the exact same interface, providing a unified way of

accessing binary streams (including files). This means that a network coder can operate on any one of the network interfaces, making the networking much more flexible. The network coders also log all the binary data (received and sent) to a file. These *raw* logs can then be used for playback at a later stage by using a file interface with the relevant coder.

The network coders run on separate threads to ensure that interface creation or slow data transfer do not interfere with the backbone object execution. The interoperability layer uses extended network coders referred to as *protocol coders* that operate on various data formats and protocols. Protocol coders are discussed in more detail later in this chapter. The network coder thread also includes mechanisms to recover and possibly re-connect when a network interface fails.

The backbone contains components that can measure the performance of the backbone object execution. This helps to optimise the object execution and distribution. The performance measures look at the following:

- the overall application load, which provides an indication of how well the application in running in general,
- the backbone overhead, which shows how much of the time is spent on modelling versus time spent on reading and writing titles,
- the hub bandwidth throughput, which gives an indication of the utilisation of the underlying transport medium when running in distributed mode, 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.

The overhead is an indication of the amount of data transported over the backbone and gives an indication of how successfully an application could be distributed. Objects that interact closely, exchanging a lot of data, should typically be located on the same node to minimise inter-node bandwidth usage.

8.3 The Infrastructure Layer

The *infrastructure* layer extends the capability of the backbone layer from simple object execution to the simulation of virtual environments. The infrastructure layer is also responsible for the simulation time management and synchronisation between different nodes.

The backbone objects and basic object titles are extended for modelling and simulation of spatial, time-based phenomena. The backbone objects are also extended to allow saving and loading object attributes in a XML format. The framework uses this to read and write XML scenario files which specify what object are loaded into the backbone.

8.3.1 Spatial Reference and Environment Models

This layer adds spatial reference models for coordinate representation and translation. The models support Meridian and Cartesian coordinates and vectors as well as orientation. This layer also includes the relevant coordinate conversion operations.

The infrastructure layer also adds the environment model for terrain. The environment model can also include things like atmosphere and sun position. The terrain gives objects the ability to get the terrain altitude as well as the ability to test for line of sight. The terrain components are designed to be easily extendable to support different terrain formats. The terrain is loaded as a set of terrain tiles.

8.3.2 The Bootloader

The bootloader introduces the backbone object XML interface. It allows backbone object attributes to be loaded (and saved) from XML scenario files. The bootloader identifies and creates objects from the object hierarchy based on the type of the object. The XML element names correspond to the relevant object class names as defined in the backbone object hierarchy.

Any backbone object that is in the object hierarchy and inherits from the XML interface can be loaded by the bootloader. This, along with the use of the backbone object factory, allows the bootloader to support an arbitrary number of objects without having to modify or recompile the infrastructure layer.

8.3.3 The Node Hub

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 through the XML scenario file.

The hub specifies the type of inter process communication (IPC) used. It controls the inter-node communication, synchronisation, node addressing and inter-node connection setup. This makes it possible to change the backbone infrastructure from a distributed peer-to-peer TCP scheme to a parallel memory-mapped scheme or even a web-based scheme by using different hub implementations.

The current framework implementation includes two hub implementations. The first is a very simple *single node* hub implementation that just delivers everything that was published in the previous frame to the local node. It is very fast and ideal when all the backbone objects are run on one node (no distribution).

The second hub implementation is a *peer-to-peer TCP/IP node* hub that allows two or more nodes to be connected (i.e. distributed backbone object execution). The hub implementation creates a mesh network with every node connected to all nodes except itself (see Figure 8.6). The hub implementation is then intelligent enough to only send issues to the nodes that have the relevant subscribers. The *peer-to-peer TCP/IP node* hub can be used to run nodes distributed over multiple hosts, but there can also be more than one node per host.

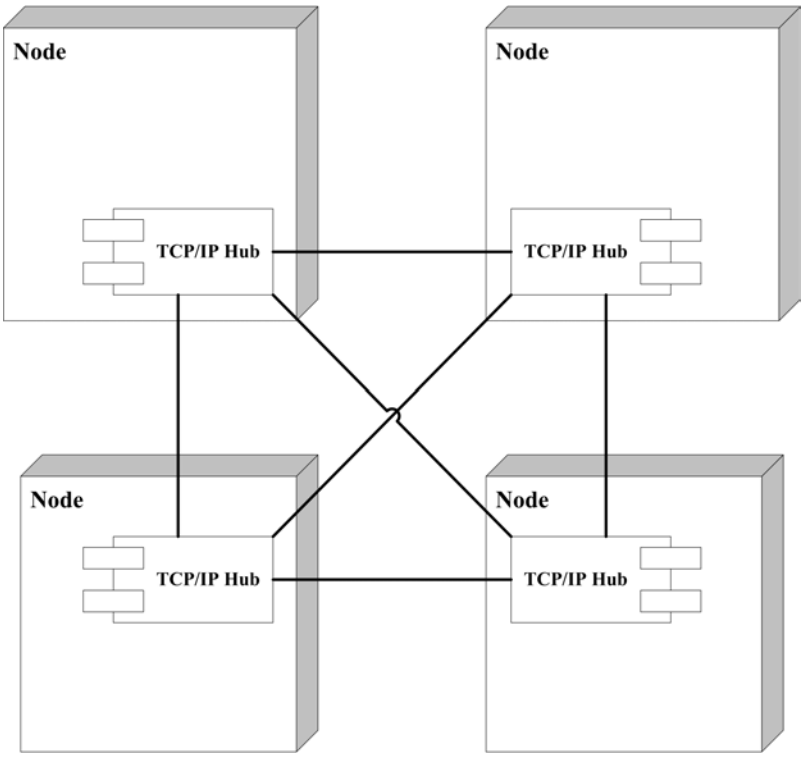

Figure 8.6: The TCP/IP Node Hub Inter-connection

8.3.4 Information Representation and Translation

The data model of a virtual environment implemented with the framework is the specific set of titles published by the simulation objects. A title does however not specify how an object should react when it receives a title through a specific subscription. The data model also includes how, in the military case, objects are classified as units, equipment, weapons, etcetera. For this the current infrastructure layer implements the MIL-STD-2525B warfighting symbology standard.

Real-world systems use tactical data links or proprietary protocols to communicate. The information exchanged by these systems may not match the titles and symbology used by the virtual environment. The internal titles would have to be translated to and from the external data models used by the real-world systems to interoperate with those systems.

Translating titles to and from the various external data models is done by the *protocol coders*, discussed in the next section, but the infrastructure layer contains the components for defining and matching the symbology used by external systems with the internal symbology. For example: the internal data model will identify an fighter aircraft as a *fixed wing military aerial unit* with a *fighter role*; an external system might only have *aircraft* or *bomber* defined.

Tactical data links have very specific ways of addressing different systems in the network, but the backbone uses a single string (the simulation ID) to identify objects. The infrastructure layer also contains components for system address translation, which allow the *protocol coders* to translate to and from external system addresses.

8.4 The Interoperability Layer

The *interoperability* layer adds the required components to interoperate with real-world systems and other simulators. This is where all tactical systems and enterprise services link into the framework. All the relevant protocols and data representations of the legacy and net-centric systems have to be understood and translated. The interoperability layer consists of the following:

- the protocol coders,
- the backbone link objects for the protocol coders, and
- the *gateway* interoperability service.

The *protocol coder* components are extended *network coder* components (see Figure 8.5). Communicating with real-world systems involves creating one or more network coders which are responsible for translating between the *external* systems and the *internal* application components built with the framework. These protocol coders operate on the syntax or structure of the foreign data and only map the information onto titles without understanding the data (i.e. on a syntax level and not on a semantic level).

The *protocol coder* components all run on their own threads to prevent slow or blocking interfaces from holding up the backbone object execution. The coders are responsible for link setup, link tear-down and handshaking (maintaining the link). These functions may be slow depending on the type of interface to the external system and the link requirements. Having the coders run on separate threads makes it very easy to execute these functions without causing the backbone object execution to be delayed—the coder executes independently of the backbone. For example: a coder can wait four seconds for a reply and only then return titles to the backbone with no delay in the object execution.

Normally the coder thread priority is set to *below normal* while the backbone worker thread priority is set to *normal*. This caused the operating system to give the backbone object execution priority above protocol coder execution. This helps to improve the backbone object execution when there are many protocol coders. The drawback of this is that the coder threads might starve (i.e. not be run by the operating system if the backbone object execution is using too much CPU resources).

Some protocol coders have very strict timing requirements and have to run with a very high confidence or at a very high rate. This is typically required when virtualising or emulating systems that normally send messages at very specific intervals. In these cases the developer can set the protocol coder thread priority very high. This commands the operating system to give the coder thread priority above all other threads. The coder execution should however be carefully controlled since high-priority threads can starve the rest of the operating system threads.

The protocol coders are not backbone objects and have to be wrapped inside extended backbone objects, called *link* objects that can be loaded and executed by the framework. The interoperability layer includes a basic *link* object that handles the locking of the coder for thread-safe access; manages the publications and subscriptions; and reports the link-state. The interoperability layer includes many different *link* objects that extend the basic link object.

The basic link also adjusts titles to have the correct time reference: incoming titles are adjusted to be relative to the simulation time while outgoing titles are adjusted to be relative to the clock of the external system. The link buffers incoming titles if the adjusted time reference is in the future. This normally occurs when reading from a file interface (i.e. all the protocol information is available immediately) or when the backbone object execution is too slow (i.e. 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 subscribes to titles from all links and selectively publishes titles back to links based on the configured routes.

8.5 The Simulation and Application Layers

The *application* layer (and to a certain extent the *simulation* layer as well) is very application specific. A typical rapid application development code-base would consist of many different simulation and application layer implementations that could be reused or extended to quickly create new tools and applications. The *simulation* layer allows developers to create unique simulations or tools by adding the required models and services.

The virtualisation (or emulation) of systems happens on this level. A simulation model is created by extending a backbone object to simulate the behaviour of an actual system. Models of aircraft, for example, would simulate the flying qualities of a specific aircraft and then publish certain attributes of the aircraft (like position and velocity).

The simulation layer is built on top of the *infrastructure* layer and the *interoperability* layer. The simulation layer is therefore ideally equipped to create models of systems that interface with the C2 enterprise as the real systems would. The next chapter discusses the *Radar Emulator* application, which is an example of this.

The framework will be used to create many different types of applications, but there are some generic components that most applications will share. This is discussed in the next few paragraphs.

8.5.1 Application Integration

Applications might require specific command-line arguments. For this the framework includes, as part of the core operating system abstractions, a command line argument parser that gives any part of an application access to the command line arguments.

Applications also require the ability to load application specific parameters from disk and save them again. The *application* layer provides an extendable XML parameter loader. It works similarly to the *backbone* object boot-loader and application developers can add additional parameter sets.

Applications need an interface for loading and saving scenarios as well as managing the *backbone* object execution. The *application* layer provides a generic *framework execution thread* that manages scenario loading, object execution, object locking and provides access to the spatial environment. This component frees the application from the complexities

of loading and saving XML files, locking backbone objects for safe access and managing the object execution. All the applications built with the framework have access to this *framework execution* component.

The *application* layer has to provide a means of collating the capabilities of the framework and integrating it with the relevant technologies for building the user applications and tools. The framework uses *control services* which are normal backbone objects that are instantiated by the user application rather than from the XML scenario file. The user application can create the *control service* and add it to the backbone (using the *framework execution* component) as soon as a scenario is loaded.

Control services can be simple or complex, depending on the application requirements. A control service can subscribe and publish like any other backbone object, but it also allows the application to access the title information through an application specific interface. This means that the application still works with titles, but the control services manage how the information gets processed and then distributed to other backbone objects. Applications can dynamically add and remove control services. One restriction is that an application should lock a control service before it uses it—this is to prevent the backbone from accessing a control service while the application is busy using it.

The next chapter evaluates the framework implementation to determine how well the implementation fits the requirements. This evaluation covers a wide range of criteria. Critical components of the framework are also evaluated using the software architecture concepts as well as the behaviour analysis techniques reviewed in part two of this dissertation.

9. Framework Evaluation

In this chapter the evaluation of the framework implementation is discussed. The evaluation covers enough aspects of the implementation to address how well the implementation fits the original requirements discussed in Chapter 7 (interoperability with C2 systems, virtualisation of C2 equipment using M&S, application development, good code quality and performance and portability).

This chapter is divided into three sections. The first section evaluates the performance and scalability of applications created with the framework. The next section discusses the existing C2 applications created with the framework. The third section formally evaluates the framework implementation. The formal evaluation is done using the software architecture concepts and behaviour analysis techniques reviewed in part two of this dissertation.

This chapter uses the terms *host*, *node*, *object*, *title* and *worker thread* extensively. Please refer back to Chapter 8 for a description of these.

9.1 Performance and Scalability Testing

Object execution performance and scalability is important when virtualising C2 equipment. This section discusses the evaluation of the general M&S performance of the framework implementation. A small test application was created with the framework. It is a very simple simulation that runs multiple instances of the same model. The model has a constant execution time of 2.5ms and only publishes one type of title. The number of titles published are however configurable. The title has several attributes and has a size of 135 bytes when streamed.

The simulation frame rate is set at 100Hz with 160 models in total. Each model subscribes to every other model (i.e. each model will receive 159 *sets* of titles per frame—one set of titles from every model). The model execution is distributed over multiple hosts as well as distributed over multiple CPU cores of one host. The test application is run multiple times with the models evenly distributed over a varying number of hosts and CPU cores. This tests both the distributed performance and parallel performance of the object execution.

The fact that the test application only has one type of model and one type of title makes it easy to analyse the distributed and parallel performance of the test application. Normally one would get several different types of models, publishing different types of tiles at lower rates (each model subscribing and publishing to every other model is actually the worst case scenario).

Figure 9.1: The 10 Toshiba Qosmio Laptops

The hosts used for the tests are Toshiba Qosmio laptops. Each laptop runs MS Windows 7 and has an Intel i7 processor with four hyper-threaded cores. The *hyper-threading* effectively provides eight *execution units* per CPU. Each *execution unit* is also referred to as a *core* in the rest of this chapter (the difference between a hyper-thread and a real CPU core is beyond the scope of this discussion). All the hosts are connected using a 1Gbps Local Area Network (LAN). Ten laptops were used for the tests.

9.1.1 Expected Behaviour

The test hosts can schedule eight threads of execution to run concurrently (one per core). An application running on a single thread only has access to one core at a time which amounts to 12.5% of the total CPU resources. An application that can use eight threads *effectively* should be able to fully utilise the host CPU.

The object execution can be distributed among several *worker threads* in the same node (i.e. parallel execution utilising multiple CPU cores of one host). The objects are divided up equally among the worker threads and the worker threads all operate independently of each other on the same set of delivered titles. Something that may affect execution performance is resource *locking*: the framework uses a memory manager that keeps track of all memory allocations within a node and each node has its own memory manager. Allocating and deleting memory involves locking some parts of the the memory manager. The contention caused by backbone objects allocating or deleting memory from different worker threads may then decrease the execution performance. The contention at the memory manager should be the only thing affecting the worker thread performance.

Distributed model execution involves executing backbone objects on different nodes and transferring data between nodes. The amount of data to transfer increases as more nodes are added, for example: 160 models distributed over two nodes adds up to 80 models per

node (i.e. 80 models are publishing to another node, giving 80 sets of titles being received and published per frame by each node); 160 models distributed over four nodes gives 40 models per node (i.e. 40 models are publishing to three nodes, giving 120 sets of titles being received and published per frame by each node). The inter-node data transfer *overhead* in the test application can be controlled by configuring the number of titles each test model publishes per frame. The network throughput is measured as the total amount of bytes sent and received by each node per second.

Nodes can be deployed per host (i.e. distributed) or as multiple processes on the same host. Communication between the nodes occur in exactly the same way in both cases (using TCP/IP). In the first case (the distributed case) the inter-host data transfer and synchronisation may reduce the overall performance and in the second case the overhead should be very low. In both cases the network overhead and host synchronisation should be the only thing affecting the execution performance if there is only one worker thread per node (since resource locking or worker thread contention within the nodes do not affect performance in these cases).

9.1.2 Parallel Performance

The framework object execution can be distributed among several worker threads in the same node (parallel execution utilising multiple CPU cores of one host). The test application was run with varying worker thread counts and detailed results are shown in Figure 9.2. A number of summarised views of this data are presented in subsequent figures.

Each table in Figure 9.2 shows the following:

- the number of titles published per model per frame,
- the number of worker threads,
- the CPU utilisation,
- the time per frame it takes to run all the models,
- the time per frame it takes to read and publish titles (overhead) and
- the speedup compared to using only one worker thread.

The *model exec* time here includes the time it takes to process the received titles.

Figure 9.3 shows how well the model execution can be sped up when the models are not publishing any titles. The figure shows the thread count (worker count) on the horizontal axis and the speed increase and CPU utilisation on the vertical axis. The speed increase and CPU utilisation are very close to linear in the number of threads (i.e the work is distributed very well with almost no overhead). The CPU utilisation shows an increase from 13 to 98 percent which is *almost perfect*. Unfortunately these results are very optimistic since the models are not publishing information.

The effect of the memory manager contention can already be seen in Figure 9.4 and Figure 9.5. As more worker threads are added, the speed increase and CPU utilisation no longer increase linearly in the number of threads: backbone objects that are publishing and processing titles need access to the memory manager and contention caused by the objects being executed by multiple worker threads increases as more threads are used.

Pub Count	Threads	CPU (%)	Model Exec (ms)	Overhead (ms)	Speedup
0	1	13	400	0.01	1.000
	2	25	200	0.02	2.000
	3	37	135	0.02	2.963
	4	50	100	0.02	4.000
	5	63	80	0.02	5.000
	6	75	68	0.02	5.882
	7	87	58	0.02	6.897
	8	98	51	0.02	7.843

Pub Count	Threads	CPU (%)	Model Exec (ms)	Overhead (ms)	Speedup
1	1	13	413	0.14	1.000
	2	25	210	0.19	1.967
	3	37	144	0.23	2.868
	4	49	109	0.23	3.789
	5	59	93	0.24	4.441
	6	68	82	0.24	5.037
	7	76	75	0.24	5.507
	8	84	69	0.25	5.986

Pub Count	Threads	CPU (%)	Model Exec (ms)	Overhead (ms)	Speedup
2	1	12	425	0.25	1.000
	2	25	225	0.34	1.889
	3	36	153	0.38	2.778
	4	48	123	0.42	3.455
	5	54	114	0.42	3.728
	6	58	107	0.42	3.972
	7	61	106	0.42	4.009
	8	65	105	0.41	4.048

Pub Count	Threads	CPU (%)	Model Exec (ms)	Overhead (ms)	Speedup
10	1	12	530	1.1	1.000
	2	24	305	1.5	1.738
	3	28	340	1.8	1.559
	4	30	405	1.9	1.309
	5	31	400	2	1.325
	8	31	395	2.1	1.342

Pub Count	Threads	CPU (%)	Model Exec (ms)	Overhead (ms)	Speedup
50	1	12	1100	6.2	1.000
	2	20	2050	8.6	0.537
	3	20	2500	9	0.440
	4	18	2600	14	0.423

Figure 9.2: Parallel execution results

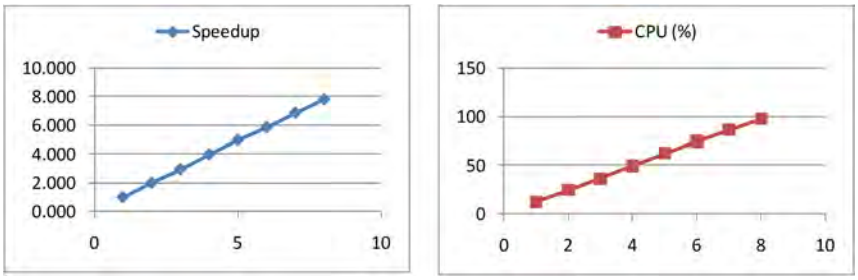

Figure 9.3: Performance with no publications

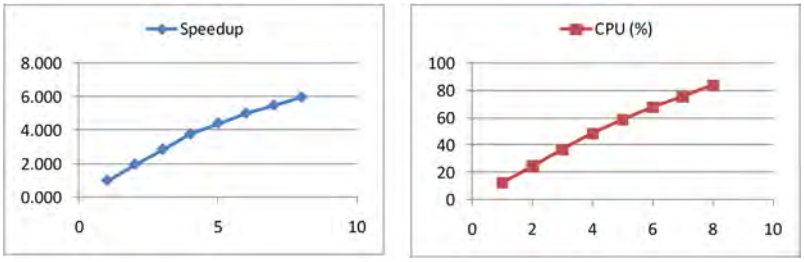

Figure 9.4: Performance with 1 title published per model per frame

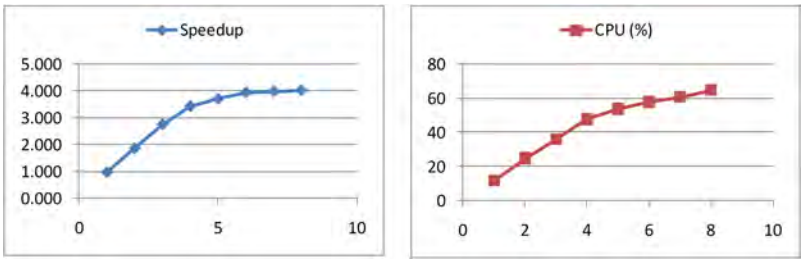

Figure 9.5: Performance with 2 titles published per model per frame

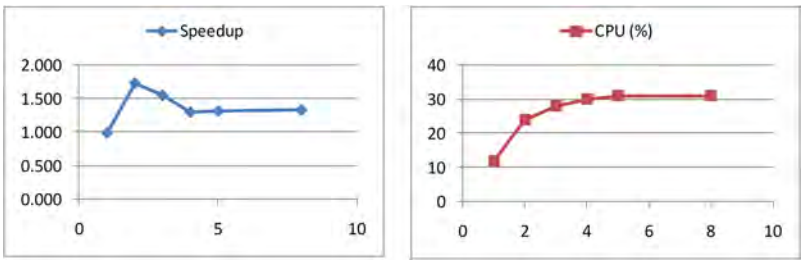

Figure 9.6: Performance with 10 titles published per model per frame

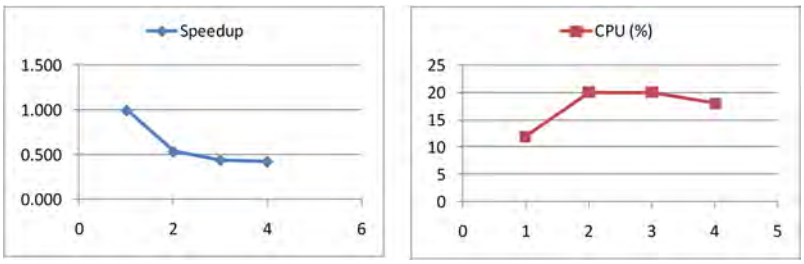

Figure 9.7: Performance with 50 titles published per model per frame

When the models publish even more titles per frame the performance suffers further. Figure 9.6 and Figure 9.7 show a decrease in performance as more worker threads are added. This indicates that the backbone objects actually spend more time waiting for access to the memory manager than they spend executing. The key to good parallel performance is minimising the number of titles published to help alleviate the contention issue between the threads.

9.1.3 Distributed Performance

This subsection discusses two sets of tests relating to distributed object execution. The first set of tests involved running multiple nodes on one of the hosts. The results are given in Figure 9.8. Each table in the figure shows the following:

- the number of titles published per model per frame,
- the number of worker threads,
- the CPU utilisation,
- the time per frame it takes to run all the models,
- the time per frame it takes to read and publish titles (overhead) and
- the total amount of data sent and received by each node (throughput in MBps), and
- the speedup compared to running all the models on only one node (see previous section).

As before, the *model exec* time here includes the time it takes to process the received titles. Each node is configured to use only one worker thread.

Pub Count	Nodes	CPU (%)	Model Exec (ms)	Overhead (ms)	Throughput (MBps)	Speedup
200	1	12	2400	14	0	1.000
	2	26	1500	60	2.6	1.600
	3	38	1200	90	4.2	2.000
	4	51	990	110	5.7	2.424
	5	63	890	200	6.2	2.697
	6	76	860	250	6.5	2.791
	7	86	860	210	6.7	2.791
	8	94	910	200	6.6	2.637

Pub Count	Nodes	CPU (%)	Model Exec (ms)	Overhead (ms)	Throughput (MBps)	Speedup
50	2	25	610	18.6	1.7	1.803
	3	38	480	30	2.7	2.292
	4	50	385	26.3	3.7	2.857
	5	65	370	60	3.8	2.973
	6	78	350	45	4.3	3.143
	7	86	320	60	4.7	3.438
	8	94	306	65	4.9	3.595

Pub Count	Nodes	CPU (%)	Model Exec (ms)	Overhead (ms)	Throughput (MBps)	Speedup
10	2	25	274	3	0.74	1.934
	3	38	195	6	1.37	2.718
	4	50	151	5	2.03	3.510
	5	63	125	20	2.3	4.240
	6	76	108	18	2.71	4.907
	7	88	98	14	3.12	5.408
	8	97	95	12	3.4	5.579

Pub Count	Nodes	CPU (%)	Model Exec (ms)	Overhead (ms)	Throughput (MBps)	Speedup
1	2	25	208	0.5	0.1	2.043
	3	37	138	3	0.19	3.080
	4	50	105	0.5	0.3	4.048
	5	63	85	2	0.38	5.000
	6	76	69	4	0.47	6.159
	7	89	62	1.2	0.57	6.855
	8	100	57	6	0.62	7.456

Figure 9.8: Distributed execution results with all nodes on one host

Figure 9.9 shows the speed increase and CPU utilisation when each model is publishing one title per frame. The figure shows the node count on the horizontal axis and the speed increase

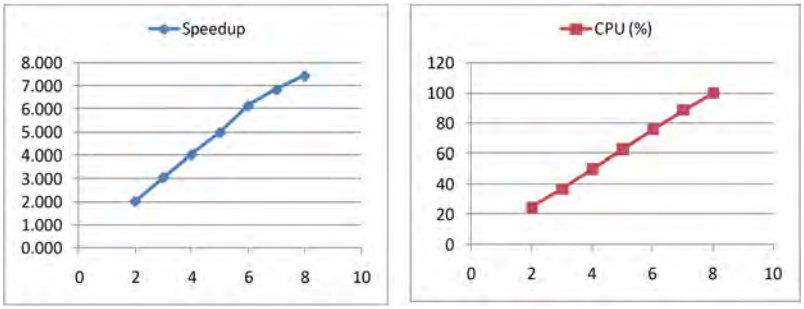

Figure 9.9: Performance with 1 title published per model per frame

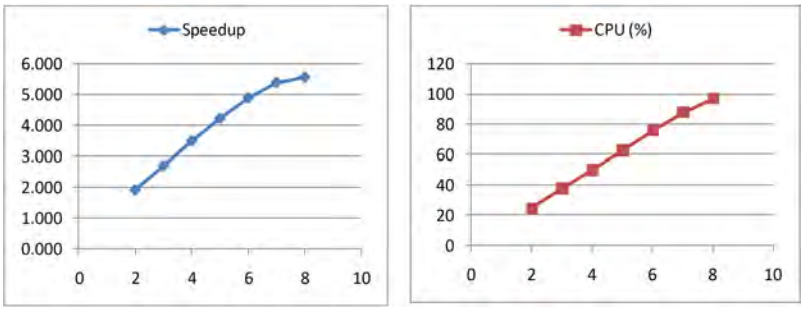

Figure 9.10: Performance with 10 titles published per model per frame

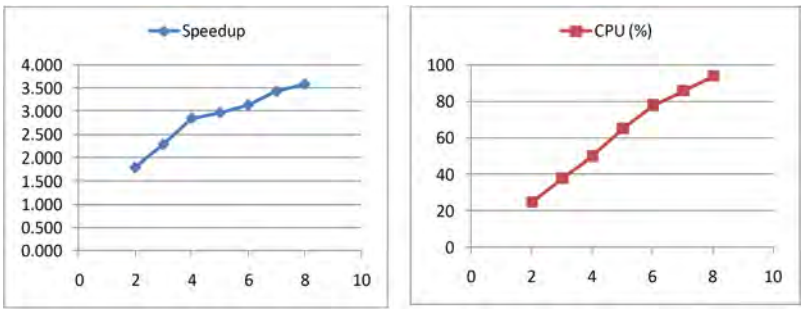

Figure 9.11: Performance with 50 titles published per model per frame

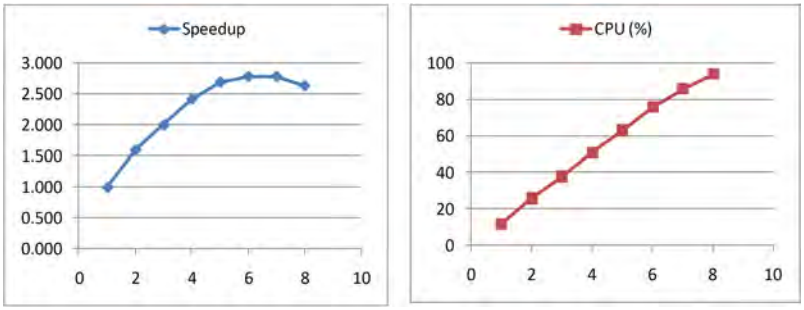

Figure 9.12: Performance with 200 titles published per model per frame

and CPU utilisation on the vertical axis. There is a linear speedup and full CPU utilisation,

which is expected since the overheads are very low.

The execution performance decreases as the overhead increases (Figure 9.10), but the CPU utilisation is however still very good. Figure 9.11 shows that the speedup is less than half of what it was for the first graph, but the CPU utilisation stays very high. This is expected, since the overheads are increasing and there is no contention between the nodes (i.e the nodes are spending more time processing titles, but without having to wait for each other). When the overheads become too big the nodes start spending more time on exchanging data than they do on executing models. This results in a decrease in performance as more nodes are used (Figure 9.12). Nevertheless, the CPU utilisation still stays very high.

The second set of tests for the distributed object execution involves several hosts, running one node each. The results are given in Figure 9.13. Each table in the figure shows exactly the same data as in the previous table, namely:

- the number of titles published per model per frame,
- the number of worker threads,
- the CPU utilisation,
- the time per frame it takes to run all the models,
- the time per frame it takes to read and publish titles (overhead) and
- the total amount of data sent and received by each node (throughput in MBps), and
- the speedup compared to running all the models on only one node (see previous section).

Again, the *model exec* time here includes the time it takes to process the received titles. There is no restriction on the number of hosts that can be used (as opposed to the previous tests where only eight processor cores were available per host). Each node is configured to use only one worker thread and with only one node per host, the CPU utilisation stays constant.

Figure 9.14 shows the speed increase and network throughput when each model is publishing 50 titles per frame. The figure shows the node count on the horizontal axis and the speed increase and network throughput on the vertical axis. Figures 9.14 and 9.15 clearly show that distributing the model execution among several hosts increases the performance. It actually performs better than expected, since it outperformed the *local host* case (see Figure 9.8) which was expected to be faster. It appears that the *local host* TCP/IP capability of the host operating system is slower than expected.

The *model execution* time increases when more data is published (Figure 9.13). This is expected since the model execution time includes the processing of the received titles. The node throughput however changes very little when more data is published. This indicates that the node throughput is at its maximum and that the execution performance is limited by the sending, receiving and processing of titles rather than the actual model execution. Figure 9.17 shows the maximum throughput achieved when the test model is modified to have an execution time of 0ms. The maximum values are very close to what is shown in Figure 9.13, further indicating that the execution performance is limited by the processing of titles rather than the actual model execution. The key here to good distributed performance is minimising the data exchange between hosts.

Pub Count	Nodes	CPU (%)	Model Exec (ms)	Overhead (ms)	Throughput (MBps)	Speedup
400	1	12	4100	50	0	1.000
	2	12	3250	120	2.4	1.262
	3	12	2250	150	4.5	1.822
	4	12	1700	200	6.5	2.412
	5	12	1450	170	7.9	2.828
	6	12	1220	180	9.5	3.361
	7	12	1050	220	10.9	3.905
	8	12	940	230	12	4.362
	9	12	860	240	13.1	4.767
	10	12	780	260	14.2	5.256

Pub Count	Nodes	CPU (%)	Model Exec (ms)	Overhead (ms)	Throughput (MBps)	Speedup
200	2	12	1675	55	2.4	1.433
	3	12	1180	80	4.4	2.034
	4	12	880	110	6.2	2.727
	5	12	735	110	7.7	3.265
	6	12	630	110	9.2	3.810
	7	12	550	130	10.5	4.364
	8	12	490	140	11.5	4.898
	9	12	460	130	12.6	5.217
	10	12	420	140	13.5	5.714

Pub Count	Nodes	CPU (%)	Model Exec (ms)	Overhead (ms)	Throughput (MBps)	Speedup
50	2	12	520	15	1.9	2.115
	3	12	360	30	3.6	3.056
	4	12	280	35	5	3.929
	5	12	230	30	6.3	4.783
	6	12	190	35	7.8	5.789
	7	12	165	40	8.6	6.667
	8	12	145	100	9	7.586
	9	12	135	40	10.5	8.148
	10	12	120	40	11.6	9.167

Figure 9.13: Distributed execution results with one node per host

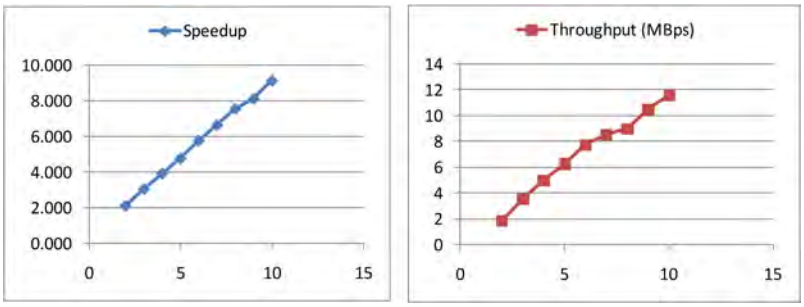

Figure 9.14: Performance with 50 titles published per model per frame

9.2 Application Examples

The previous section discussed the general M&S performance of the framework. This section discusses applications which served to further verify some of the interoperability and virtualisation (i.e. M&S) capabilities of the implemented framework. Four applications within the C2 domain were developed and successfully applied. In addition, three test applications were also developed. These test applications served to verify the behaviour and performance of the *backbone* and *infrastructure* layer implementations of the framework. The C2 applications are more complex and their successful implementation provides additional

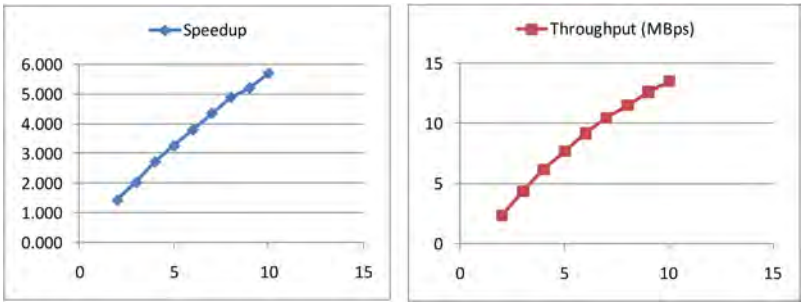

Figure 9.15: Performance with 200 titles published per model per frame

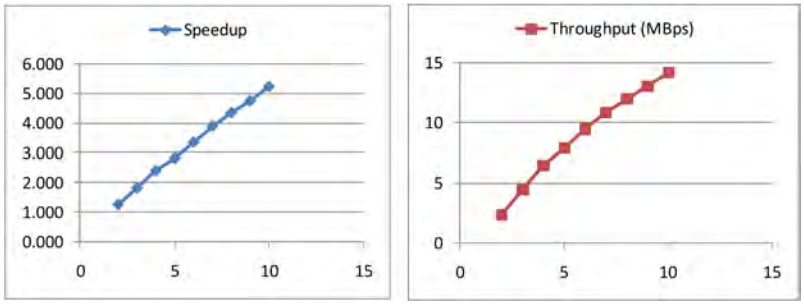

Figure 9.16: Performance with 400 titles published per model per frame

Pub Count	Nodes	Model Exec (ms)	Overhead (ms)	Throughput (MBps)
200	2	1480	51	2.67
	3	1040	70	4.8
	4	780	100	7

Figure 9.17: Performance with a model exec time of 0ms

positive evidence in verifying the *interoperability* and *application* layers of the framework.

The framework and the three test applications were designed and implemented by the author. However, the four C2 applications discussed below were developed by the relevant CSIR project teams in which the author was merely the technical lead[1]. Although the *development* of the C2 applications cannot be regarded as part of the research discussed in this dissertation, the fact that they were successfully implemented using the framework bears testimony to its validity and usability and, for this reason, deserves to be mentioned here.

The test applications are:

- the performance test application discussed in the previous section of this chapter,
- a test application that simulates the flocking behaviour of birds, and
- an implementation of Conway's game of life.

The C2 applications are:

[1]Because these C2 applications deal with restricted information they are only discussed in broad overview. The author can be contacted for additional information.

- an air to air tactics evaluation tool for fighter aircraft,
- a protocol gateway that facilitated air force, navy and military system interoperability during preparatory field exercises for the soccer world cup 2010,
- a radar emulator for adding additional information sources to an air force system, and
- a *joint operations operator console* concept demonstrator.

The C2 applications all use custom user interface and 2D/3D visualisation layers that are integrated with the framework. These layers use technologies like *wxWidgets*, *Qt* and *Open Scene Graph*. The user interface and visualisation layers are application specific and do not form part of the research effort discussed in this dissertation. The previous chapter (Chapter 8) did however discuss how such application specific layers integrate with the framework.

The framework source code has been included on the DVD accompanying this dissertation. The complete source code for the three test applications is also included [2].

9.2.1 A Simulation of Flocking Behaviour

Craig Reinolds developed an artificial life program called *Boids* in 1986 which simulates the flocking behaviour of birds. The test application discussed in this section implements that same *boid* behaviour. The flocking behaviour is simulated by creating multiple instances of a model (or *boid*) that moves around according to a fixed set of rules. The rules define how each boid behaves within its flock: each boid tries to stay close to the center of the flock; each boid tries to move in the same direction as the flock; and, each boid tries to avoid flying into other boids. It is possible to create very realistic flocking behaviour with these simple rules.

In order to test the *infrastructure* layer the flocking behaviour is simulated in a spherical coordinate system (i.e. the boid positions are represented using latitude, longitude and altitude). The model rules however operate within a local cartesian coordinate system and the infrastructure layer contains the components that represent the two coordinate systems and can translate between them.

Each boid subscribes to the state of every other boid and also publishes its own state using a state title. The state title provides the position (in spherical coordinates) and the velocity of a boid. Each boid builds up a set of flock-mate positions (in local coordinates) with the state titles from its subscriptions. Each boid then calculates its own velocity based on the *boid* rules, updates its own state and then publishes it. This type of simulation, with the models operating at discrete time steps, can be referred to as discrete time simulation (DTS).

The boid models follow smooth trajectories that can be approximated by some form of prediction. The framework accommodates techniques like *dead-reckoning* by allowing subscribers to predict the future state of the information received from publishers. The boid model updates the set of flock-mate positions with state titles that come through on the subscriptions or by predicting the last known state for flock-mates that did not update. The boid model does not publish its own state if it knows that the prediction used by other boids

[2] The four C2 applications created with the framework can not be included on the DVD since the applications contain restricted or sensitive information.

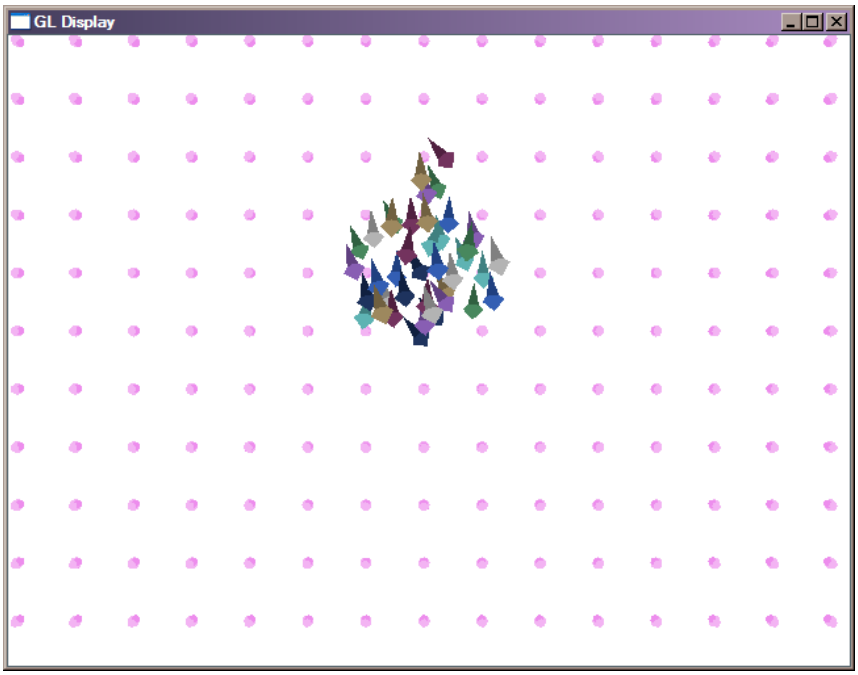

Figure 9.18: A 3D View of 40 Flocking Models in Test Application

would produce an accurate enough result (this functionality is part of the framework). This can be used to make a trade off between accurate boid movement and publishing fewer titles.

Figure 9.18 shows a simple custom 3D view that was created as part of this test application. This instance had 40 models specified in the XML scenario file. The boid model execution can also be distributed and parallelised in exactly the same way as in the *performance test application* discussed in the previous section of this chapter.

This test application simulated the flocking behaviour as expected and ran without problems. The application can be found on the accompanying DVD.

9.2.2 Conway's Game of Life

John Horton Conway developed *Conway's Game of Life* in 1970. The game has no players and takes place on a two dimensional grid of square cells. The cells switch on and off based on specific game rules and the game grid evolves based on the initial cell pattern. The game runs through multiple iterations, each cell changing in each evolution step on the basis of rules relating to the cell states of its immediate neighbours.

Each cell has eight immediate neighbours (cells that are directly vertically, horizontally and diagonally adjacent). The rules are:

- a cell that is off and has exactly three on neighbours switches on in the next iteration of the game;
- a cell that is on and has more than three on neighbours switches off in the next iteration;
- a cell that is on and has less than two on neighbours switches off in the next iteration.

The rules aim to mimic the behaviour of cells living and dying as a consequence of reproduction, overcrowding and under-population. The test application discussed in this section implements the game of life with one model per cell subscribing to the state of its eight neighbours and publishing its own state whenever it changes.

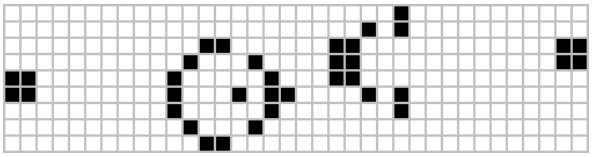

Figure 9.19: The Initial Cell Pattern Used by the Test Application

Figure 9.19 shows the initial pattern used by the test application: the cell models are created in either the on state or in the off state based on this pattern. This particular pattern, when being evolved by the game of life, is called the *Gosper Glider Gun* and the cells switch on and off in such a way that it looks like a gun firing bursts that fly off indefinitely.

Figure 9.20: A View of 2000 Cells in Test Application

Each cell in the test application publishes its initial state and then only publishes its state again when it changes. Each cell also remembers the last state of each of its neighbours in case they do not publish. The test application uses a 50 by 40 cell grid—that is 2000 cell models running in the backbone (shown in Figure 9.20). The performance is still extremely good since each cell only reacts if it gets a new update from one of its neighbours and then only publishes its own state if its state changes. This type of simulation, where the models only react to events and only publish events, is referred to as discrete event simulation.

The cell model execution can also be distributed and parallelised in exactly the same way as in the *performance test application* discussed in the previous section of this chapter. This application can be found on the accompanying DVD. It demonstrates how scalable

the backbone object execution is—it easily runs 2000 models.

9.2.3 An Tactics Evaluation Tool for Fighter Aircraft

In 2008 an early version of the framework implementation was used to develop an air-to-air tactics evaluation tool for the new generation Gripen fighter aircraft acquired by the South African Airforce (SAAF). The Gripen has a higher situational awareness than previous SAAF aircraft and it also has the ability to share information with other aircraft over a tactical data link.

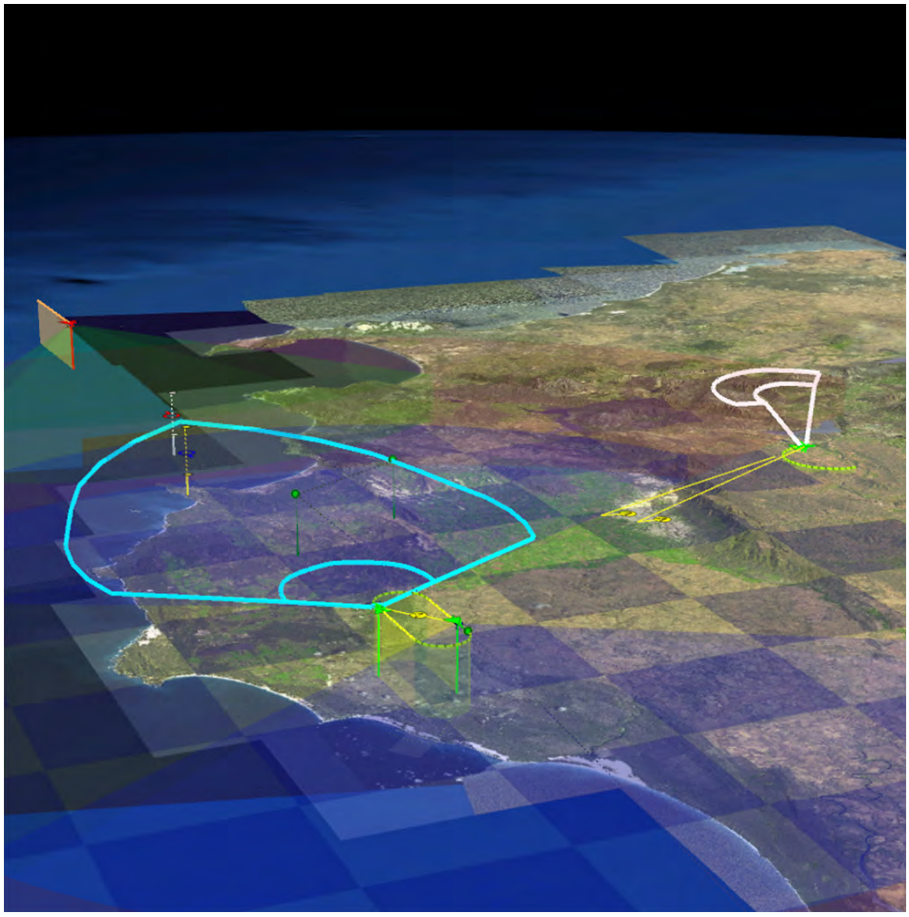

Figure 9.21: The Tactics Evaluation Tool (3D view)

The goal of the tool was to help develop new air-to-air tactics for the Gripen aircraft using modelling and simulation (M&S). The key is simulating accurate aircraft flight paths, correctly simulating the behaviour of the tactical data link between aircraft and correctly simulating the behaviour of the aircraft radar. The tool helps the SAAF to quickly generate information regarding the aircraft performance. This information can then be applied to help the SAAF use the official Gripen mission planning systems more effectively.

The tool is currently still being extended by the Defence Peace Safety and Security, Aeronautics research group within the CSIR. A custom 3D view component, developed by the CSIR, is also integrated into the tool in order to visualise various aspects of the aircraft,

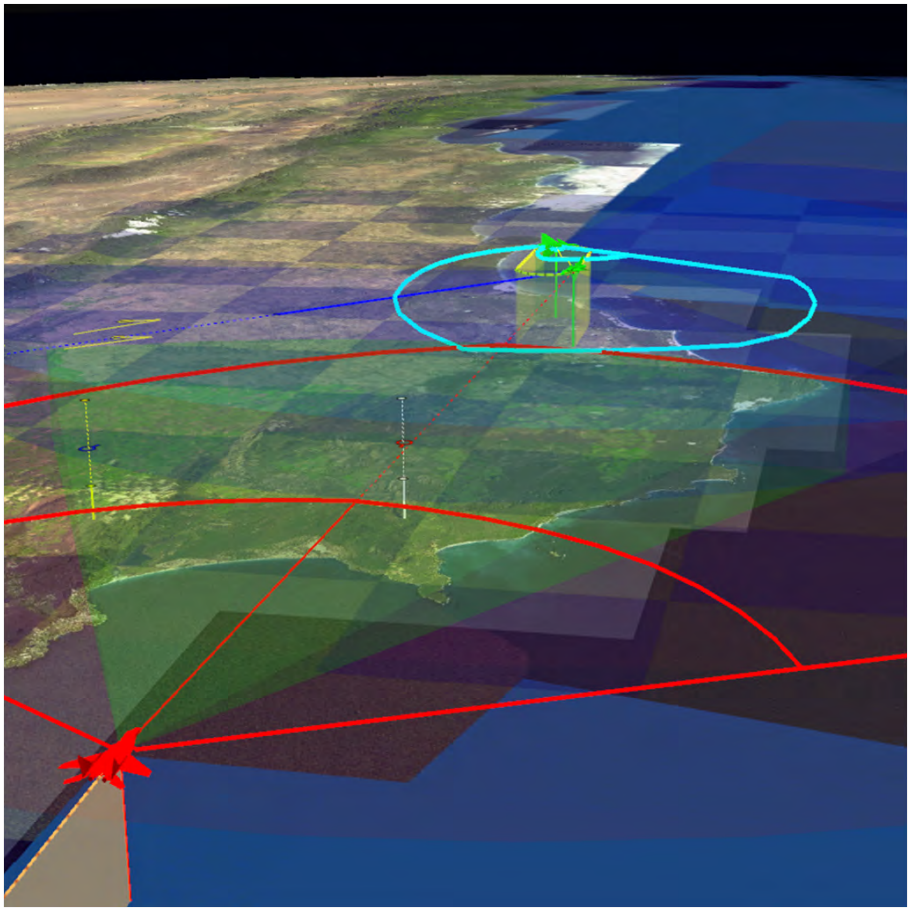

Figure 9.22: The Tactics Evaluation Tool (Closeup of Aircraft Model)

radar and data link models. Figure 9.21 and Figure 9.22 show what the tool looks like.

9.2.4 A Command and Control Protocol Gateway

The framework was used to build a gateway application that can act as a message router for various systems and simulators (Duvenhage and Terblanche, 2008). The gateway implements all the links required to connect to the relevant systems and to exchange information with these systems. The gateway also translates the information to and from an internal representation. This allows the gateway to route information between systems, acting as a C2 hub. Most of the gateway functionality is implemented in the *infrastructure* and *interoperability* layers of the framework, giving all applications access to it. The gateway consists of multiple link objects and one gateway object.

The gateway links are implemented as different backbone objects that publish the information they receive. The link objects are reusable across multiple applications. The links are fault tolerant and translate the information of the various external systems to a unified internal representation that is used throughout the rest of the application.

The gateway object subscribes to all the links; it routes and filters information; and it publishes back to the links. The links also subscribe to information from the gateway. The

subscriptions and publications are registered in such a way that the gateway object can receive information from any link object and publish back to specific link objects based on the routing rules. The gateway implements special *internal* links that enable filtering and exchanging of information with other backbone objects. The internal links allow the gateway to be part of any application created with the framework. The routing rules, links and internal links are setup in the XML scenario file.

This gateway was used extensively to facilitate system interoperability during preparatory military field exercises for the Soccer World Cup 2010—the police, air force, army and navy had to work together and system interoperability was crucial. The gateway was used to relay tactical information between operational air force, navy and army systems. This helped create awareness of the importance of interoperability for joint operations.

9.2.5 A Radar Emulator

The gateway also has the ability to emulate (i.e. virtualise) a specific type of radar system by implementing the same link protocol (in a link object) as an actual radar system does. Several radar systems used by the South African Air Force (SAAF) use this protocol.

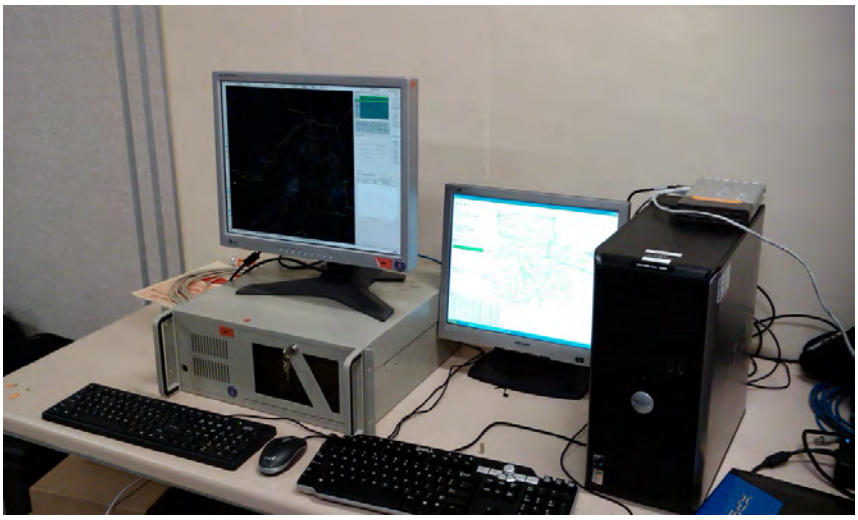

Figure 9.23: The Radar Emulator Test Setup at the SAAF Head Quarters

Figure 9.23 shows a test setup of this *radar emulator* at the SAAF Head Quarters, Pretoria, South Africa. The machine running the gateway/emulator software is on the right and a stand-alone SAAF air picture display system is on the left. The SAAF system accepts radar inputs via an HDLC interface card. The *emulator* machine also has an HDLC interface card and connects to the SAAF system via a serial cable (as the real system would). For all intents and purposes the *emulator* machine then looks like a real radar system to the SAAF system.

The radar emulator has been used during SANDF field trials to integrate additional sensors into a specific SAAF system. The gateway translates information from systems that would normally not function with the SAAF system into something that looks like information from one of the standard SAAF radars. This gives operators access to additional information that would otherwise not have been so readily available.

In the test setup shown in Figure 9.23 the gateway listens in on a SAAF aircraft tactical radio link (using a compatible link radio) and receives reports of each aircraft's own position as well as contact reports from each aircraft's onboard radar. These reports are then sent to the SAAF air picture display system as radar plots. This test setup was built in collaboration with Saab Systems South Africa who provided all the relevant protocol specifications, the HDLC card interface and the radio interface. Saab also assisted with setting up the radio link to the aircraft.

9.2.6 A Joint Operations Operator Console

The Joint Operations Operator Console (JOOC) is a generic platform for technology demonstrators within the Joint Command and Control (JC2) context. The JOOC demonstrates concepts concerning *air picture management*, *multi-sensor fusion* and system interoperability. *Air picture management* concerns sensor tracks of aircraft and the management thereof—an air picture manager can classify or modify existing tracks to be more accurate. Multi-sensor fusion is the process whereby tracks from two or more sensors are associated and combined into a single set of tracks (i.e. only one track for each aircraft in the air).

The JOOC includes the C2 Protocol Gateway capabilities as well as a 3D view for geospatial information and additional services that enable air picture management and multi-sensor fusion. A geospatial view displays the situational picture and allows the user to interact with it.

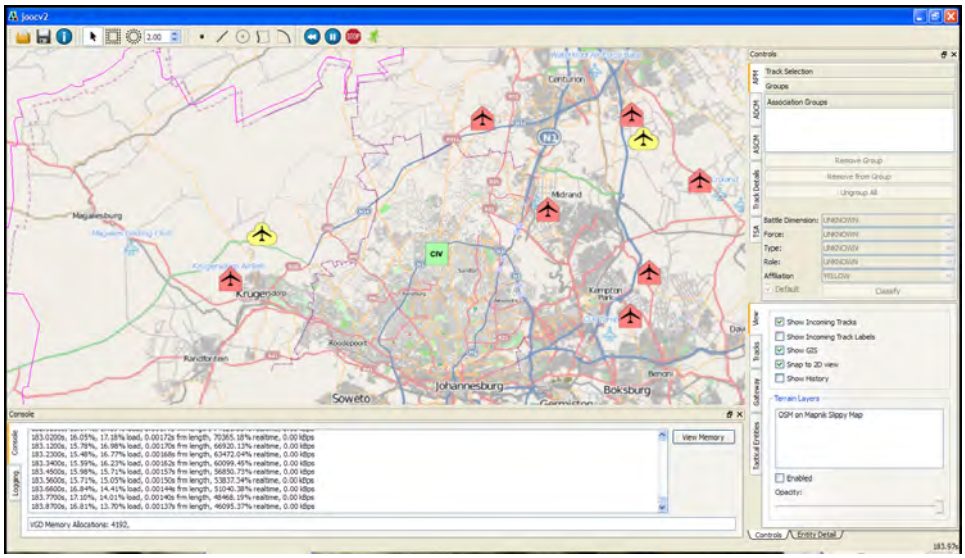

Figure 9.24: The JOOC with Various Air Tracks in the View

The JOOC is currently being extended for the Ground Based Air Defence (GBAD) environment. It will be used to test concepts in regimental-level air defence (a role of the South African Army). Regimental-level air defence concerns the control and management of multiple air defence deployments, optimising air defence by looking at all the deployed equipment and resources in a holistic fashion and collating the air defence efforts.

The extended JOOC (shown in Figure 9.24 and figure 9.25) will allow one to set up different mock-up air defence terminals and have real military personnel experiment with and evaluate

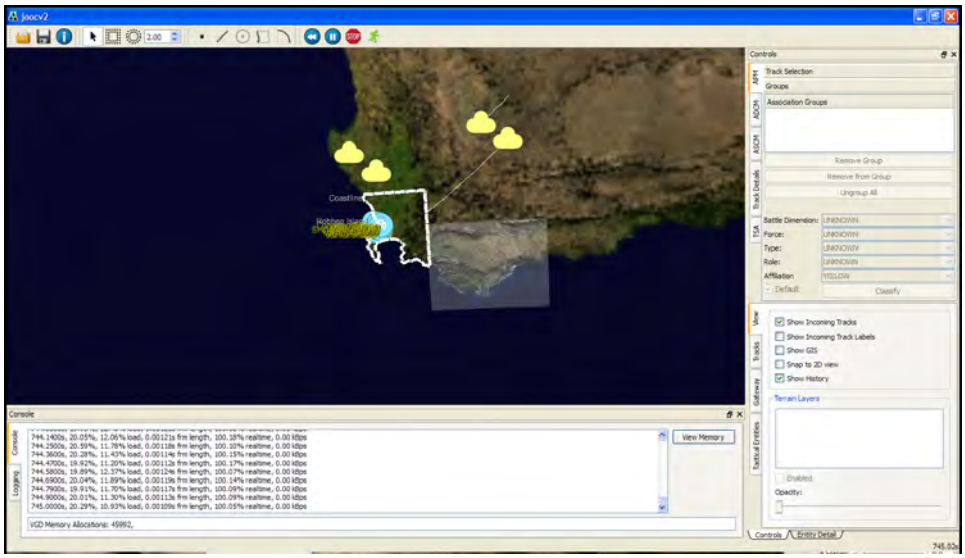

Figure 9.25: The JOOC With Some Air Tracks and GBADS Elements

their operational procedures for regimental fire control. What the role of each air defence operator should be and how different operators should interact with each other, could then be optimised.

Figure 9.24 and Figure 9.25 show the JOOC user interface. In both figures, the geospatial view is in the top-center panel, and the main user interface panels are at the bottom and to the right. The bottom panel contains tabs for logging and a text based console that reports framework event and status information—the M&S capability of the framework is always active. The two panels to the right show the gateway routes and link status, some view controls and several tabs related to air defence control. The panels are user interface widgets that use *control* objects (see Chapter 8) to exchange the relevant information and events with backbone objects.

This extended JOOC will also be used to evaluate possible ways of integrating the GBAD systems with air defence systems from the South African Air Force (SAAF) and the South African Navy (SAN). Integration of the Army, Air Force and Navy air defence capabilities is referred to as Joint Air Defence (JAD).

Figure 9.26 shows an earlier version of the JOOC that was integrated with the base station of a military UAV during a demonstration at the CSIR. The integration was done via the gateway that forms part of the JOOC. The UAV position as well as the positions of targets of interest could be sent from the UAV's base station (via a serial interface) to the JOOC and displayed in 3D. The gateway also has a link that can get civilian air traffic information from local air traffic control centres. This demonstration verified that it is possible to integrate and present military UAV and civilian air traffic information on one view.

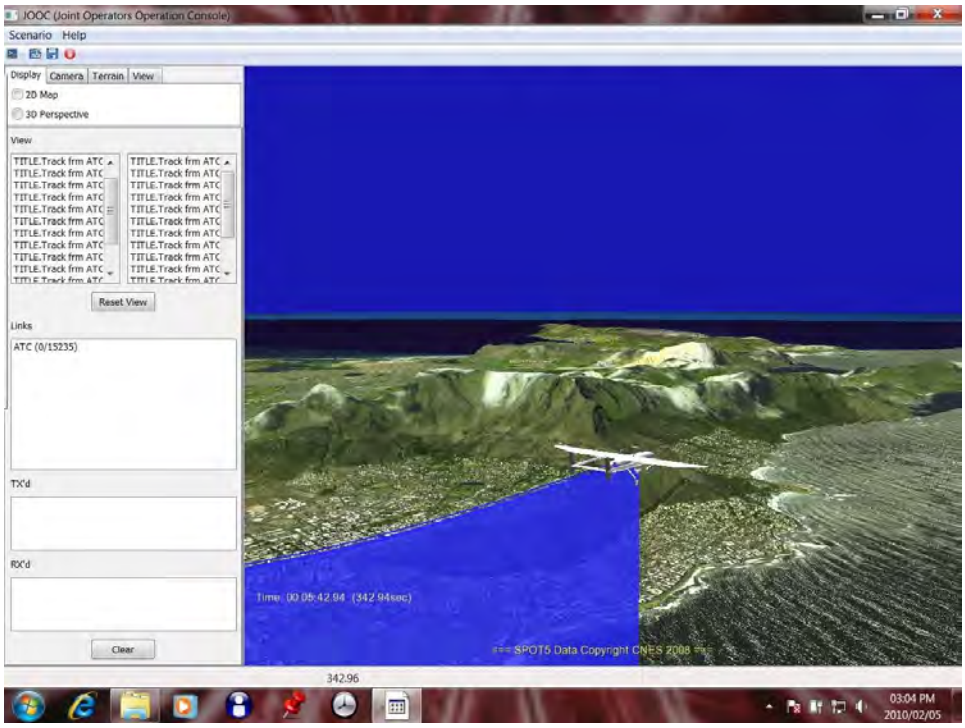

Figure 9.26: The JOOC Integrated With a Military UAV Base Station

9.3 Formal Evaluation

Part two of this dissertation showed how to describe the structure and behaviour of software. In this section, critical components of the software framework design are identified and then formally described and evaluated using the methods discussed in Chapter 4.

9.3.1 Distributed Execution

Distributed model execution involves executing backbone objects on different nodes and transferring data between nodes. The backbone object execution is divided up into frames and the nodes operate in a lock-step fashion (i.e. a node will not start a new frame until all the other nodes have finished the previous frame). Each frame goes through several steps or states which can be summarised as follows:

1. The node receives communications events, generated in the previous frame, from all other nodes (the information from each node is abstracted into one event). The node will not continue until it has received communications events from all nodes.
2. The node processes the communications events (delivering the issues generated in the previous frame to all backbone objects; and then executing the relevant backbone objects).
3. The node sends out the new information published by the objects and generated by the backbone as communications events to all the other nodes.
4. The node goes back to step one and starts waiting for communications events from

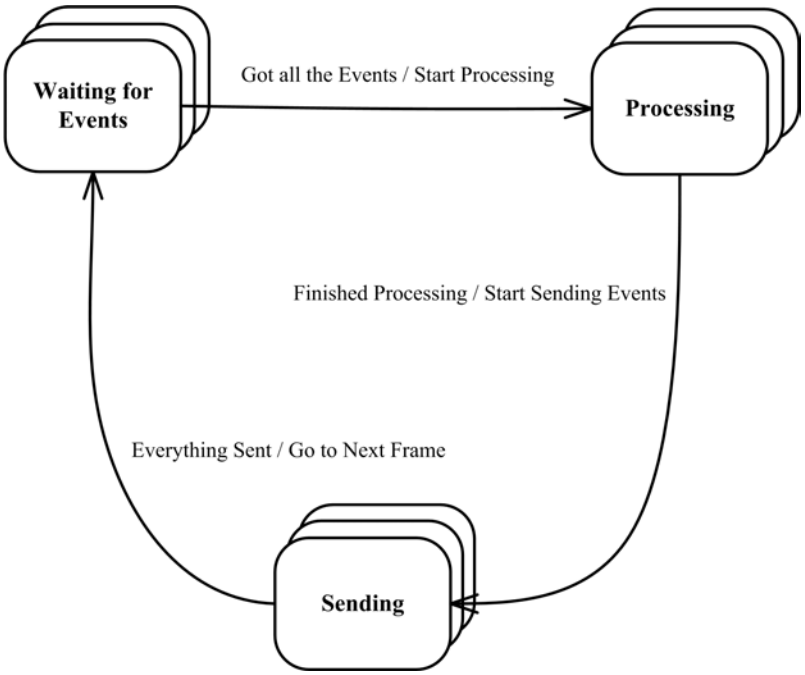

Figure 9.27: A Finite State Machine Showing the States of a Node Frame

other nodes. This ignores the additional wait the framework can make to keep the simulation from running faster than real-time.

These steps match the backbone object execution steps discussed in Chapter 8, but have been adapted for this discussion. Figure 9.27 shows the FSM for a node frame (the layered blocks indicate that multiple nodes can be in that state). The start of the FSM is also not shown, since the very first frame of each node does not include the *read* step—this FSM excludes the first frame.

The following CSP analysis provides an alternative view to the FSM shown in Figure 9.30. Figure 9.28 shows the communication events between three nodes. The nodes are numbered 1 to 3 (*NODE1*, *NODE2*, *NODE3*) and each node can be seen as a process reading and writing communications events. The communication events are *C12, C13, C21, C23, C31 and C32*. To make the analysis simpler the events are reduced to *C1, C2, C3* for the events that are received from the nodes and *D1, D2 and D2* for the events that are sent from the nodes. It is correct to rename the sent events from C to D, since sent events are only read in the next frame. The CSP for the individual processes can be written as follows:

$$\begin{aligned} NODE1 &= (C2 \rightarrow C3 \rightarrow D1 \rightarrow NODE1) \mid \\ &\quad (C3 \rightarrow C2 \rightarrow D1 \rightarrow NODE1) \qquad (9.1) \\ NODE2 &= (C1 \rightarrow C3 \rightarrow D2 \rightarrow NODE2) \mid \\ &\quad (C3 \rightarrow C1 \rightarrow D2 \rightarrow NODE2) \qquad (9.2) \\ NODE3 &= (C1 \rightarrow C2 \rightarrow D3 \rightarrow NODE3) \mid \\ &\quad (C2 \rightarrow C1 \rightarrow D3 \rightarrow NODE3) \qquad (9.3) \end{aligned}$$

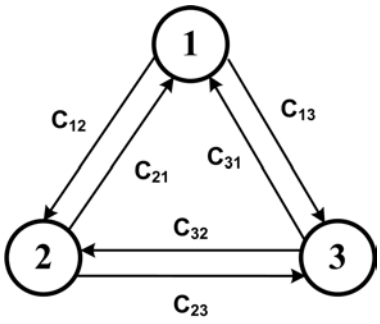

Figure 9.28: The Communication Events Between Three Nodes

The processes run in parallel, but have to synchronise on common events. This can be expressed as:

$$NODES = NODE1 \parallel NODE2 \parallel NODE3 \tag{9.4}$$

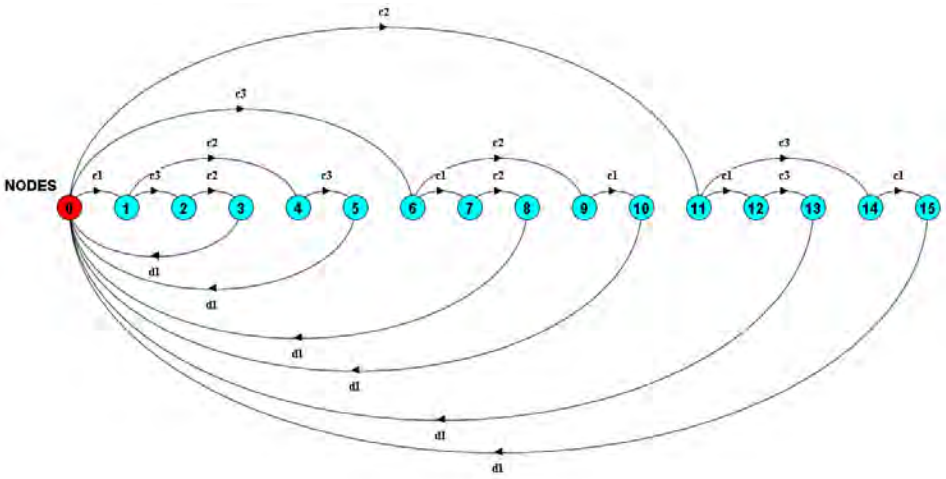

Figure 9.29: The LTSA transition diagram for *NODES*

This CSP model was evaluated using a tool called the *Labelled Transition System Analyser (LTSA)*. The tool uses a Finite State Process (FSP) textual notation to represent the CSP. The tool can analyse the models for deadlock. The CSP model for *NODES* can be expressed in FSP as:

$$
\begin{aligned}
NODE1 &= (c2- > c3- > d1- > NODE1 \mid \\
&\quad c3- > c2- > d1- > NODE1). && (9.5)\\
NODE2 &= (c1- > c3- > d2- > NODE2 \mid \\
&\quad c3- > c1- > d2- > NODE2). && (9.6)\\
NODE3 &= (c1- > c2- > d3- > NODE3 \mid \\
&\quad c2- > c1- > d3- > NODE3). && (9.7)
\end{aligned}
$$

$$
||NODES = (NODE1 \parallel NODE2 \parallel NODE3)/\{d1/d3, d1/d2\}. \quad (9.8)
$$

The events *d2* and *d3* are renamed to *d1* in the FSP model for *NODES*. This is legitimate, since the order in which the different nodes write is not important (the nodes only have to write something after the reads). The model forces synchronisation on the write and on the reads. Using the LTSA tool it was determined that there is no possibility of deadlock. The transition diagram generated for *NODES* by the LTSA tool is also shown in Figure 9.29.

9.3.2 The Frame Execution and Multi-threading

The backbone objects can be executed concurrently (i.e. by multiple worker threads), since backbone objects do not interact directly with each other. The backbone has to interact with the *node hub* to send and received information. This can only happen on a single worker thread, called the *main thread*, since the node hub implementations might not be thread safe (see Chapter 8). Because of this the other worker threads have to block or wait while the main thread does the extra work. The *Distributed Object Execution* is discussed in detail in Chapter 8.

The states of each node frame can be summarised as follows:

1. The main thread receives communications events from all the other nodes via the node hub (these communications events were generated in the previous frame). The main thread will not continue until it has received communications events from all nodes. The worker threads wait for the main thread to continue.
2. The worker threads (including the main thread) process the communications events. The threads are responsible for delivering all issues published in the previous frame and executing the relevant objects. The threads can all work on the same set of issues, since the issues are not modified while delivered.
3. Once all the threads have finished processing, the threads continue. The main thread collects all the new information published by the backbone objects and then sends the information as new communications events to all the other nodes. The worker threads are finished until the next frame and start waiting for the main thread.
4. The node goes back to step one and starts waiting for communications events from other nodes. This ignores the additional wait the framework can make to keep the simulation from running faster than real-time.

These steps also match the backbone object execution steps discussed in Chapter 8, but have

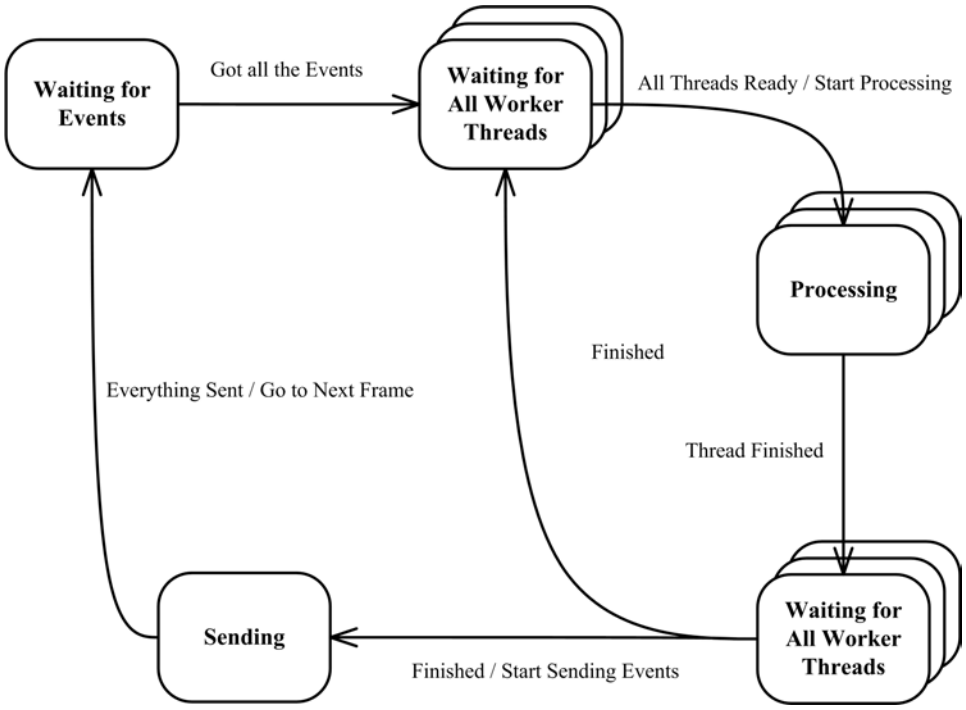

Figure 9.30: A Finite State Machine Showing the States of the Main Thread and Worker Threads

been adapted for this discussion. Figure 9.30 shows the Finite State Machine (FSM) for the *main* thread combined with the FSM for the *worker* threads (the layered blocks indicate that multiple threads can be in that state).

The worker threads are synchronised with the main thread by two *barriers* (see Chapter 8). This is to ensure that worker threads stop executing objects while the published issues are being sent out by the main thread. The worker threads wait at the first barrier for the main thread and the second barrier effectively forces the main thread to wait for all the worker threads to finish executing objects before continuing. Using barriers like this is very efficient, since threads use very little CPU resources when waiting on a barrier.

Having some threads do work while other threads have to wait might decrease the CPU utlisation. The backbone could be configured to have more worker threads than CPU cores. Another, probably better solution, would be to balance the load of the different worker threads in some way.

The following CSP analysis provides an alternative view to the FSM shown in Figure 9.30. There are two types of processes executing objects: the main thread and the worker threads. The main thread has to first read all the information from other nodes, synchronise with worker threads, do some processing, synch with worker threads again and then send out new information to other nodes. This can be expressed in CSP as follows:

$$MAIN = Read \rightarrow Sync1 \rightarrow Sync2 \rightarrow Send \rightarrow MAIN \quad (9.9)$$

Similarly the CSP for the worker thread can be expressed as:

$$WORKER(i) \quad = \quad Sync1 \rightarrow Sync2 \rightarrow WORKER(i) \tag{9.10}$$

The threads run in parallel and synchronise on the two barriers (the *Sync1* and *Sync2* events). This can be expressed as:

$$NODE \quad = \quad MAIN \parallel WORKER(1..n) \tag{9.11}$$

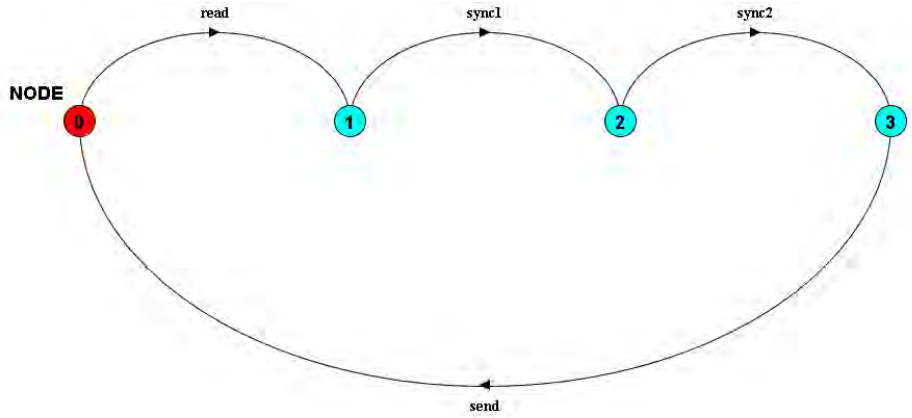

Figure 9.31: The LTSA transition diagram for *NODE*

The CSP model for *NODE* can be expressed in FSP as:

$$\begin{aligned} MAIN &= (read->sync1->sync2->send->MAIN). && (9.12)\\ WRK1 &= (sync1->sync2->WRK1). && (9.13)\\ WRK2 &= (sync1->sync2->WRK2). && (9.14)\\ WRK3 &= (sync1->sync2->WRK3). && (9.15) \end{aligned}$$

$$||NODE \quad = \quad (MAIN \parallel WRK1 \parallel WRK2 \parallel WRK3). \tag{9.16}$$

The FSP model for *NODE* has three worker thread processes (*WRK1, WRK2 & WRK3*). All the worker threads (and the main thread) must jointly synchronise on *sync1* and *sync2* (i.e. every worker thread will wait for all the others and for the main thread to execute *sync1*, and similarly for *sync2*). Using the *LTSA* tool it was found that there is no possibility of deadlock in this model. The transition diagram generated for *NODE* by the LTSA tool is also shown in Figure 9.31.

9.4 General Discussion

This chapter discussed the framework implementation in three sections: the first evaluated the performance and scalability of applications created with the framework; the second discussed the existing C2 applications created with the framework; and, the third formally evaluated the framework implementation. This section provides some additional comments on the framework implementation.

Successful distribution of the object execution depends on how much information each model is publishing. This applies to both the parallel and distributed cases:

- When the object execution is distributed among several worker threads within one node (parallel execution) the execution speedup decreases as threads start waiting for each other. This is clear from the CPU utilisation that decreases as more objects are published. The data exchange overhead is also very small in this case since all the objects are running on one node.
- When the object execution is distributed among several nodes (on one or more hosts) the execution speedup decreases as more time is spent on exchanging data. The CPU utilisation stays high, even though the speedup decreases.

The backbone is designed for distributed object execution. Parallel object execution augments the distributed execution by utilising more of the host resources. The key to good parallel performance is CPU utilisation and solving the contention issue between the multiple threads. The key to good distributed performance is load balancing and optimising the data exchange between hosts. In general the number of titles published should be kept to a minimum.

The infrastructure and interoperability layers add to the functionality of the backbone by giving applications access to spatial simulation and interoperability capabilities. The object execution and inter object communication is however not compromised by the additional layers: each model or interoperability link is also a *backbone object*. The framework is also flexible enough to allow for discrete time and discrete event based simulation.

Each framework layer extends the capabilities of the layer below it (for example, the infrastructure layer adds capabilities to the backbone layer and the interoperability layer adds capabilities to the infrastructure layer—see Figure 8.1). Each layer does however depend on the layer that it extends. This means that the infrastructure, interoperability and simulation layers can be modified or replaced with new implementations, but the dependencies need to be managed (for example, modifying the information model defined in the infrastructure layer might affect the link implementations in the interoperability layer).

The application layer serves as a set of templates or rules that help guide developers on how to develop the relevant user interfaces and then integrate those user interfaces with the simulation and interoperability capabilities. Many of the application layer components are reusable across multiple applications. The application examples discussed in this chapter give a clear indication of the virtualisation and interoperability capabilities of the framework. The user interface and visualisation layers are application specific and do not form part of the research effort discussed in this dissertation.

Good code quality and portability is hard to measure in isolation. In general the acceptance of the framework and the way in which it is used to create applications can be taken as

an indication of the code quality. Each new application developed with the framework also provides the opportunity to identify and solve potential deficiencies in the framework and to add more capabilities to the relevant layers.

The next part of this dissertation provides the lessons learned and possible future work on the framework.

CONCLUSION

The current status of the framework, the lessons learned and possible future work on the framework are discussed in this final part of the dissertation. The work discussed in this dissertation also contributes to a larger vision of unified system development within the command and control environment.

10. Conclusion

This chapter provides the lessons learned and possible future work on the framework. The bigger vision of unified system development within the Command and Control environment is also discussed.

10.1 The Framework Implementation

Early versions of the framework have been in use since March 2008 and this has already resulted in four applications that have successfully been applied within the C2 domain (see Chapter 9):

- an air to air tactics evaluation tool for fighter aircraft,
- a protocol gateway that facilitated air force, navy and military system interoperability during preparatory field exercises for the soccer world cup 2010,
- a radar emulator for adding additional information sources to an air force system, and
- a *joint operations operator console* concept demonstrator.

Software requirements can change often and more command and control systems might have to be supported—the success of the applications created with the framework depends on the quality of the framework design and implementation. From the application examples, discussed in Chapter 9, it should be clear that the current implementation of the framework is successful. The test applications created with the framework (see Chapter 9) also show that the framework is flexible enough to allow for discrete time and discrete event based simulation.

The code-base has however not undergone the rigorous testing and validation required to qualify it for use in safety critical systems. Creating operational systems is in fact outside the scope of the framework and this should rather be done by the local defence industry. For now the framework remains part of the support services provided to the defence force.

The framework source code has been included on the DVD accompanying this dissertation. The complete source code for the three test applications, discussed in Chapter 9, is also included. The four C2 applications created with the framework can however not be included on the DVD since the applications contain restricted or sensitive information. The source code for the user interface and visualisation layers used by the C2 applications is not included on the DVD, since it is not directly part of the software framework discussed in this dissertation.

Essentially all the components of the framework can be made open source (i.e. the source code would be publicly available): making the framework open could advance the development of the framework as well as increase the number of applications created with it. Unfortunately, at this stage, contractual complications with the armaments industry prevents this from happening and the source code remains the property of the CSIR.

10.2 Future Work

Distribution of the applications over wide area networks is becoming more important and the current simulation time management and information distribution will have to be updated. The node hub implementation would have to be updated to support nodes entering and leaving the simulation on the fly. This has the added benefit of improving the fault tolerance of the framework by allowing backup nodes to take over when primary nodes fail.

It is worthwhile mentioning that the use of the framework in no way negates the use of something like the High Level Architecture (HLA) for simulation interoperability. The framework can be applied to enhance the capability and quality of HLA federates and could very well be extended to be a federate development environment. The framework also has the potential to parallelise a federate's internal model execution.

10.3 An Open Unified Architecture for System Development

The current framework addresses system virtualisation and system interoperability. The work presented in this dissertation however also contributes to a bigger vision of unified system development within the context of command and control. There is a need for a *unified software architecture for system software development* that enables modular C2 systems with reusable sub-systems. The current framework implementation can be used to create the software for systems within the C2 environment, but falls short when it comes to operational systems since it is not qualified for safety critical applications. An M&S capability is also not necessarily required by operational C2 systems and equipment.

Any system that would function within the C2 environment would have to be compatible with this unified architecture for to be a success. Sub-systems bought from international vendors would also have to be comply with this architecture. This would lead to an open middleware implementation for creating the software for all local C2 systems. One would need buy-in and acceptance of the unified architecture (and the middleware implementation) from the local defence industry. This might seem excessive, but it might also prove to be a *necessary evil* in achieving truly modular systems. A unified architecture will also reduce the required skill set of system developers, potentially extending the operational lifetime of the systems.

10.4 Final Thoughts

This now concludes the final part of this dissertation. This dissertation discussed the design, implementation and evaluation of a *software framework for supporting distributed Command*

and Control applications. The work represents an *unique hybrid approach* that combines M&S and system interoperability to build distributed C2 support software.

The work was put into perspective by an extended literature review and it was shown that the current design and implementation of the framework is of a high quality and is successful. The research outcomes include the framework implementation as well as the key requirements for providing interoperability and M&S support to the C2 enterprise. These research outcomes will contribute to further research in system interoperability, M&S and unified system development within the C2 environment.

Bibliography

Alhir, S. (2003). *Learning UML*, O'Reilly & Associates Inc., California, USA.

Capps, M., McGregor, D., Brutzman, D. and Zyda, M. (2000). NPSNET-V: A new beginning for dynamically extensible virtual environments, *IEEE Computer Graphics and Applications* pp. 12–15.

Chaum, E. and Lee, R. (2008). Command and control common semantic core required to enable net-centric operations, *Critical Issues in C4I*, AFCEA-GMU C4I Center, George Mason University, Fairfax, Virginia Campus.

Crane, S., Campbell, C. and Scannell, L. (2008). Bridging the digital devide with net-centric tactical services, *Critical Issues in C4I*, AFCEA-GMU C4I Center, George Mason University, Fairfax, Virginia Campus.

Daly, J. and Tolk, A. (2003). Modeling and simulation integration with network-centric command and control architectures, *SISO Fall SIW*, Simulation Interoperability Standards Organization, pp. 40–49.

Duvenhage, A. and Duvenhage, B. (2008). An alternative to dead reckoning for model state quantisation when migrating to a discrete event architecture, *ECMS*, The European Council for Modelling and Simulation.

Duvenhage, A. and le Roux, W. (2007a). A state estimation approach for live aircraft engagement in a C2 simulation environment, *SISO Fall SIW*, Simulation Interoperability Standards Organization.

Duvenhage, A. and Terblanche, L. (2008). The evolution of a command and control protocol gateway, *SISO Euro SIW*, Simulation Interoperability Standards Organization, pp. 51–58.

Duvenhage, B. and Kourie, D. (2007). Migrating to a real-time distributed parallel simulator architecture, *2007 Summer Computer Simulation Conference, California.*

Duvenhage, B. and Kourie, D. (2008). *Migrating to a real-time distributed parallel simulator architecture*, Master's thesis, Department of Computer Science, University of Pretoria, South Africa.

Duvenhage, B. and le Roux, W. (2007b). A peer-to-peer simulation architecture, *In Proceedings of the 2007 High Performance Computing and Simulation Conference (HPC&S 2007)*, European Council for Modelling and Simulation, pp. 684–690.

Duvenhage, B. and Senekal, F. (2004). VGD 3 architecture review, *Technical report*, Council for Industrial and Scientific Research, South Africa.

Eugster, P., Felber, P., Guerraoui, R. and Kermarrec, A.-M. (2003). The many faces of publish/subscribe, *ACM Computing Surveys (CSUR)* **35**(2): 114–131.

Fujimoto, R. (2000). *Parallel and Distributed Simulation Systems*, Wiley Interscience.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (2004). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley.

Hamilton, J. and Catania, G. (2003). A practical application of enterprise architecture for interoperability, *International Conference on Infomation Systems and Engineering*, pp. 183–188.

Harless, W. and Roose, K. (1999). Considerations for the inclusion of the gateway in the long term HLA interoperability tool suite, *SISO Fall SIW*, Simulation Interoperability Standards Organization.

Keen, M., Acharya, A., Bishop, S., Hopkins, A., Milinski, S., Nott, C., Robinson, R., Adams, J. and Verschueren, P. (2004a). *Patterns: Implementing an SOA Using an Enterprise Service Bus*, WebSphere software, IBM, chapter 3, p. 55.

Keen, M., Acharya, A., Bishop, S., Hopkins, A., Milinski, S., Nott, C., Robinson, R., Adams, J. and Verschueren, P. (2004b). *Patterns: Implementing an SOA Using an Enterprise Service Bus*, WebSphere software, IBM, chapter 4, p. 73.

Kuhl, F., Weatherly, R. and Dahmann, J. (1999). *Creating Computer Simulation Systems, An Introduction to the High Level Architecture*, Prentice Hall, Upper Saddle River, NJ.

Larsen, P. (2006). Coalition C2 interoperability challenges, *The 11th Command and Control Research and Technology Symposium*, DOD Command and Control Research Program.

le Roux, W. (2002). VGD 2.0 architectural design considerations, *Technical report*, Council for Industrial and Scientific Research, South Africa.

le Roux, W. (2006). Implementing a low cost distributed architecture for real-time behavioural modelling and simulation, *SISO Euro SIW*, Simulation Interoperability Standards Organization.

le Roux, W. (2008). Interoperability requirements for a south african joint command and control test facility, *SISO Euro SIW*, Simulation Interoperability Standards Organization, pp. 87–96.

Macedomia, M., Zyda, M., Pratt, D., Brutzman, D. and Barham, P. (1995). Exploiting reality with multicast groups: a network architecture for large-scale virtual environments, *Virtual Reality Annual International Symposium*, pp. 2–10.

Miller, D. and Thorpe, J. (1995). Invited paper - SIMNET: The advent of simulator networking, *Proceedings of the IEEE* **83**(8): 1114–1123.

Möller, B., Morse, K., Lighter, M., Little, R. and Lutz, R. (2008). HLA evolved - a summary of major technical improvements, *2008 Fall Simulation Interoperability Workshop*.

Moller, B. and Olsson, L. (2004). Practical experiences from HLA 1.3 to HLA IEEE 1516 interoperability, *SISO Fall SIW*, Simulation Interoperability Standards Organization.

Morse, K., Drake, D. and Brunton, R. (2004). Web enabling HLA compliant simulations to support network centric applications, *Command and Control Research and Technology Symposium*.

Morse, K., Lighter, G., Lutz, R., Saunders, R., Little, R., Möller, B. and Scrudder, R. (2005). Evolving the high level architecture for modeling and simulation, *The Interservice/Industry Training, Simulation & Education Conference (I/ITSEC)*.

Naidoo, S. and Nel, J. (2006). Modeling and simulation of a ground based air defense system and associated tactical doctrine as part of acquisition support, *SISO Fall SIW*, Simulation Interoperability Standards Organization.

Nel, J., le Roux, W., van der Schyf, O. and Mostert, L. C. M. (2007). Modelling joint air defence doctrinal issues with a LinkZA-based integration of two c2 simulators - a case study, *Military Information and Communications Symposium of South Africa (MICSA)*, Armscor, Command and Management Information Systems (CMIS).

Nel, J., Roodt, J. and Oosthuizen, R. (2007). The design of the M&S acquisition support effort of the SANDF GBADS acquisition programme, *SimTecT*, Simulation Industry Association of Australia (SIAA).

Olsson, J. and Michalski, R. (2008). *Serious games—integrating games in military training*, Master's thesis, LTH School of Engineering at Campus Helsingborg, Lunds University, Sweden.

Pokorny, T. (2005). Practical XMSF: Open source tools for enabling web based simulation, *SimTecT*.

Roscoe, A. (2005). *The Theory and Practise of Concurrency*, Prentice Hall.

Schmidt, D., Stal, M., Rohnert, H. and Buschmann, F. (2000). *Pattern-oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Vol. 2, Wiley.

Schulte, R. (2002). Predicts 2003: Enterprise service bus emerge, *Predicts 2003: SOA Is Changing Software*, Gartner, Inc.

Shaw, M. and Garlan, D. (1996). *Software Architecture, Perspectives on an Emerging Discipline*, Prentice Hall, Upper Saddle River, NJ.

Straßburger, S. (2000). *Distributed Simulation Based on the High Level Architecture in Civilian Application Domains*, PhD thesis, Otto-von-Guericke University MagdeBurg.

Tanenbaum, A. and van Steen, M. (2007). *Distributed Systems, Principles and Paradigms*, 2 edn, Prentice Hall, Upper Saddle River, NJ.

Zimmermann, H. (1980). OSI reference model - the ISO model of architecture for open systems interconnection, *IEEE Transactions on Communications*, Vol. COM-28, pp. 425–432.

Appendix A: Papers Published Related to Framework

This appendix contains four papers, authored or co-authored by Arno Duvenhage, that discuss research related to the work presented in this dissertation. All four papers were also presented by Arno Duvenhage.

- The Evolution of a C2 protocol gateway, *The Simulation Interoperability Standards Organization (SISO) Euro SIW 2008 Conference, Edinburgh, Scotland, 16-19 June 2008.*
- Effectively Utilizing a 3rd Party 3D Visualization Component in a Discrete Event Simulation Environment for Joint Command and Control (JC2), *Fall Simulation Interoperability Workshop 2009, Orlando, Florida, 21-25 September 2009.*
- Experiences From Constructing Command and Control Simulations Using a Tactical Data Link Standard, *Fall Simulation Interoperability Workshop 2009, Orlando, Florida, 21-25 September 2009.*
- A Layered Distributed Simulation Architecture To Support The C2 Enterprise, *The Simulation Interoperability Standards Organization (SISO) Fall SIW 2009 Conference, Orlando, Florida, 21-25 September 2009.*