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

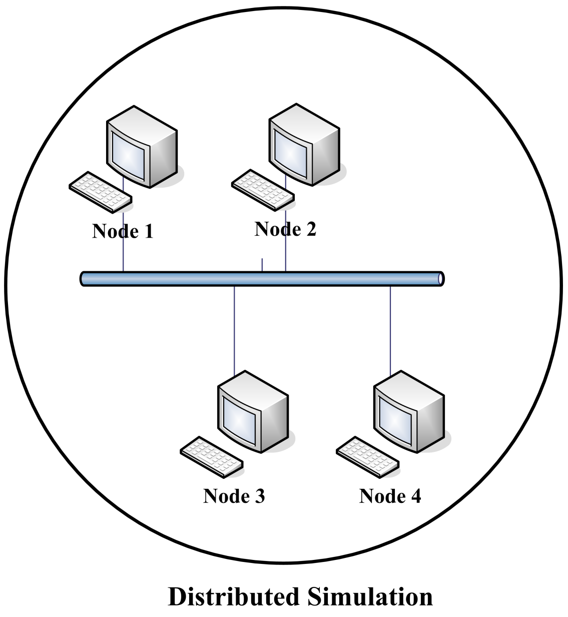

Figure 1.1: The Physical Deployment of a Distributed Simulation.

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.

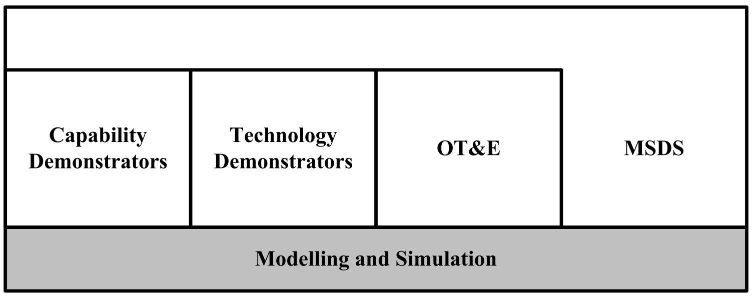

Figure 1.2: The Modelling and Simulation Competencies

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.

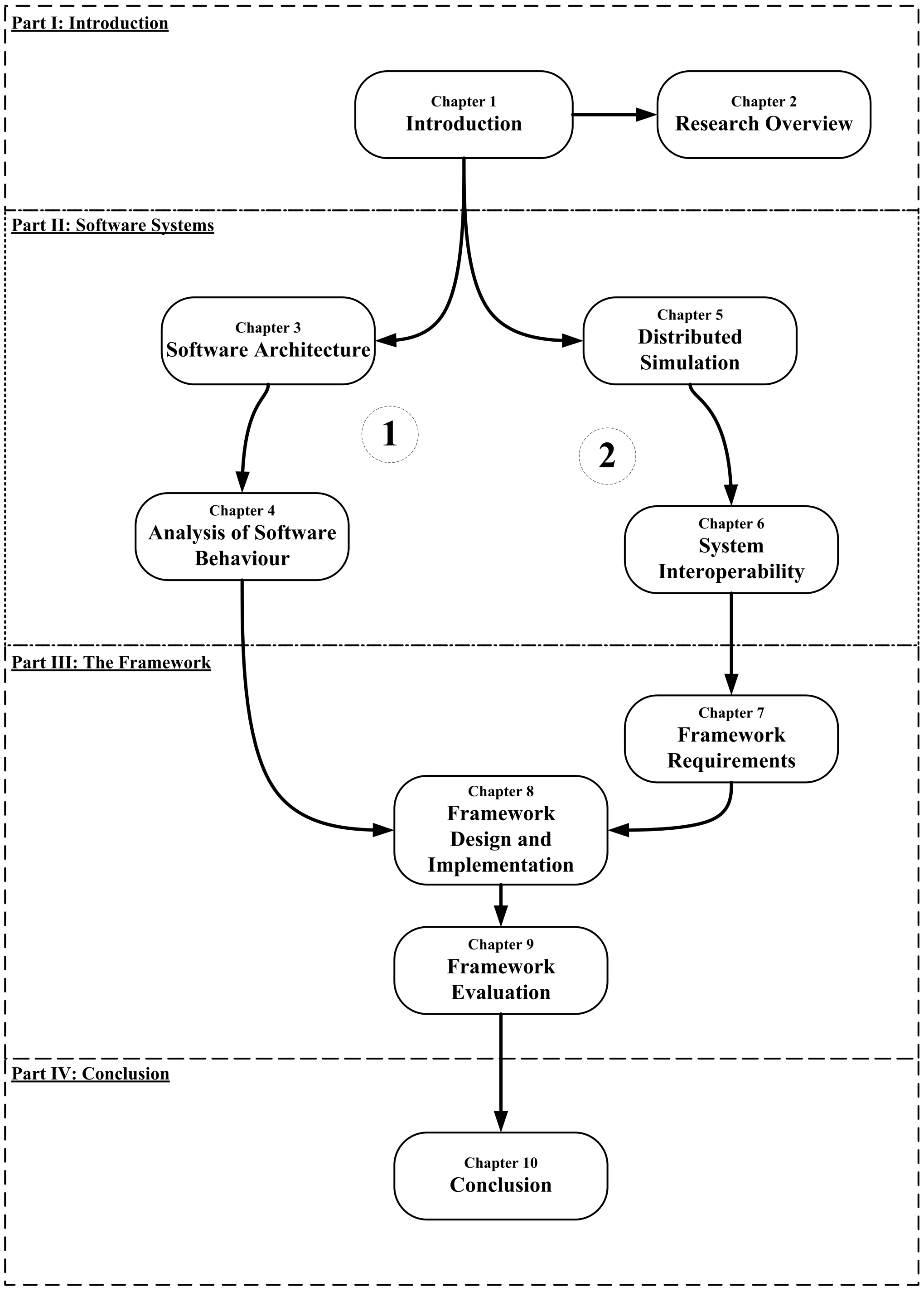
Part I: Introduction
Chapter 1 Introduction
Chapter 2 Research Overview
Part II: Software Systems
Chapter 3 Software Architecture
Chapter 5 Distributed Simulation
1
2
Chapter 4 Analysis of Software Behaviour
Chapter 6 System Interoperability
Part III: The Framework
Chapter 7 Framework Requirements
Chapter 8 Framework Design and Implementation
Chapter 9 Framework Evaluation
Part IV: Conclusion
Chapter 10 Conclusion

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 standalone 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.

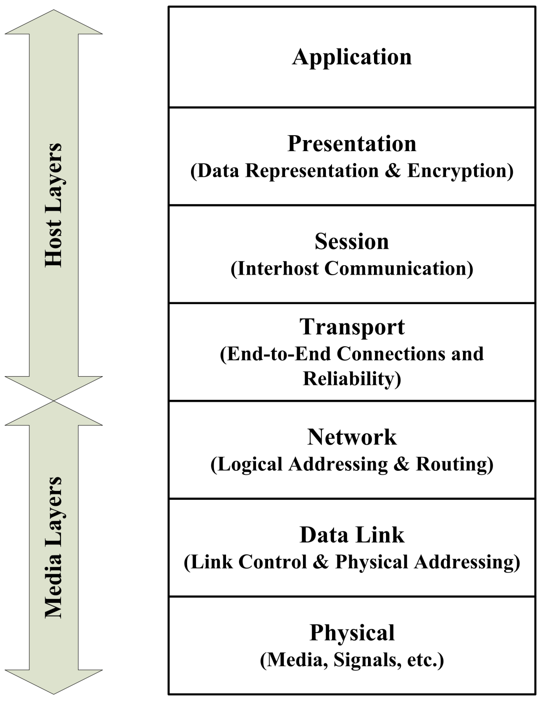
Host Layers
Media Layers
Application
Presentation
(Data Representation & Encryption)
Session
(Interhost Communication)
Transport
(End-to-End Connections and Reliability)
Network
(Logical Addressing & Routing)
Data Link
(Link Control & Physical Addressing)
Physical
(Media, Signals, etc.)

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

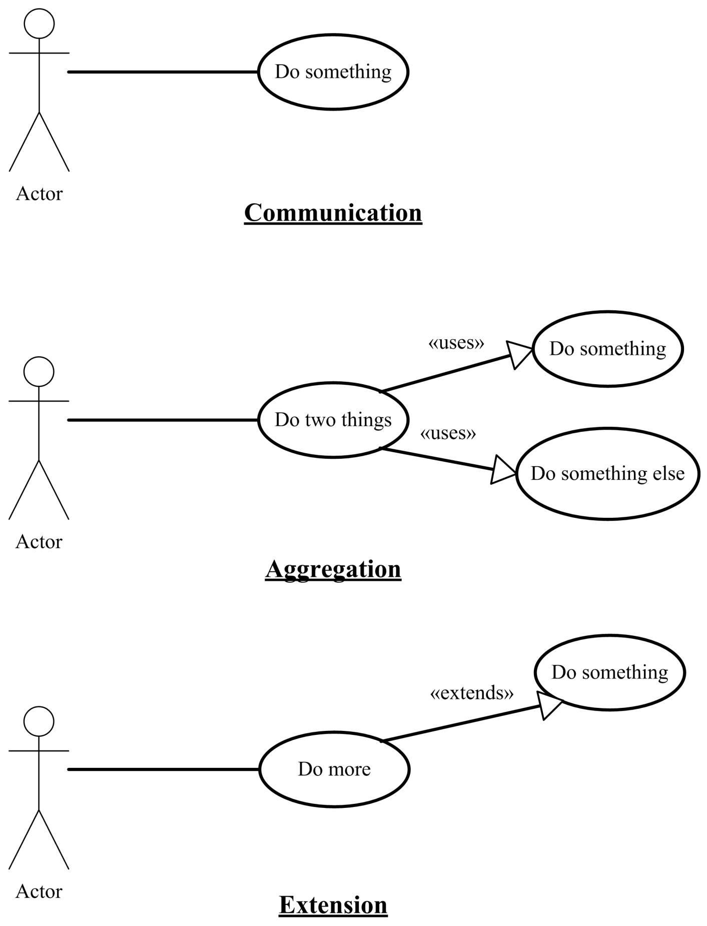

Figure 4.1: UML Use Case Diagrams

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):

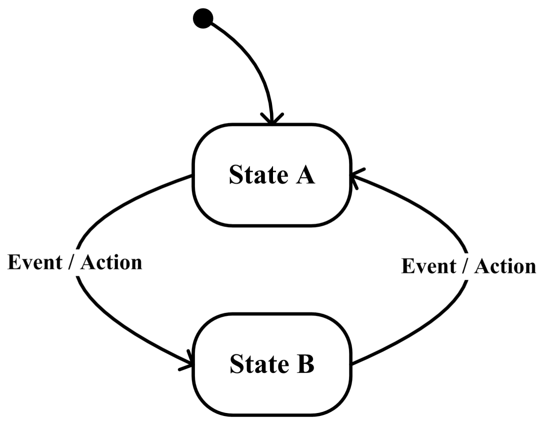

Figure 4.2: UML Finite State Machine

- *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*: $P \,\square\, Q$ allows the environment to choose the first events of P and of Q and then behaves accordingly. Also $(a \rightarrow P) \,\square\, (b \rightarrow Q)$ 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*: $P \,⊀ b ⊁\, Q$ 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.

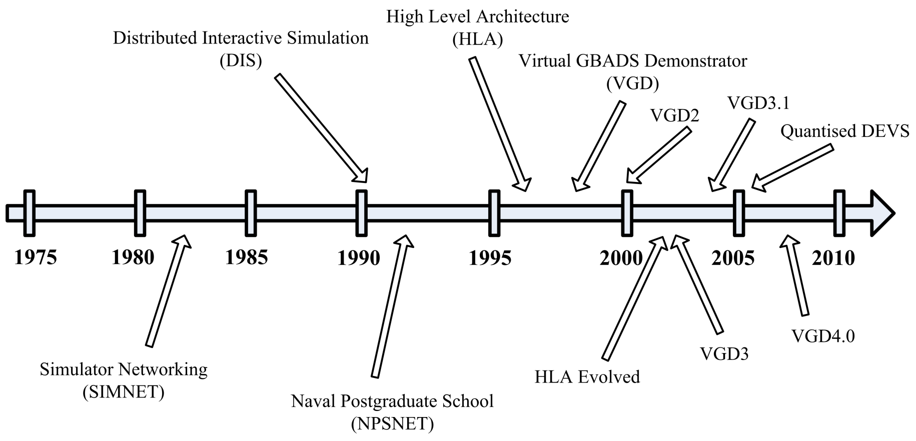
High Level Architecture (HLA)
Distributed Interactive Simulation (DIS)
Virtual GBADS Demonstrator (VGD)
VGD2
VGD3.1
Quantised DEVS
1975
1980
1985
1990
1995
2000
2005
2010
Simulator Networking (SIMNET)
Naval Postgraduate School (NPSNET)
HLA Evolved
VGD3
VGD4.0

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).

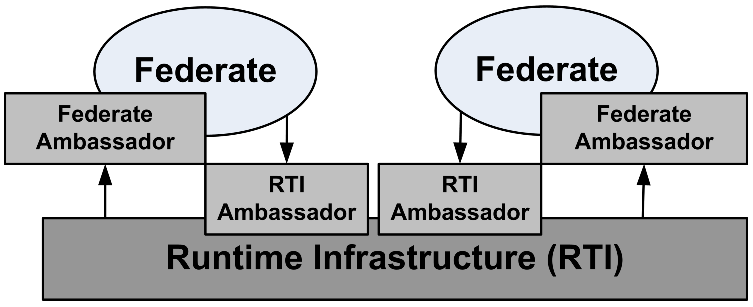

Figure 5.2: The HLA Interface between RTI and Federates

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).

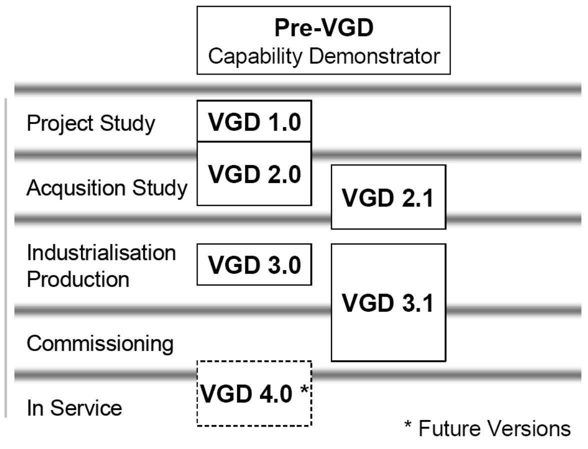

Figure 5.3: The VGD Versions and Acquisition Phases
Taken from (le Roux, 2006), with permission.

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.

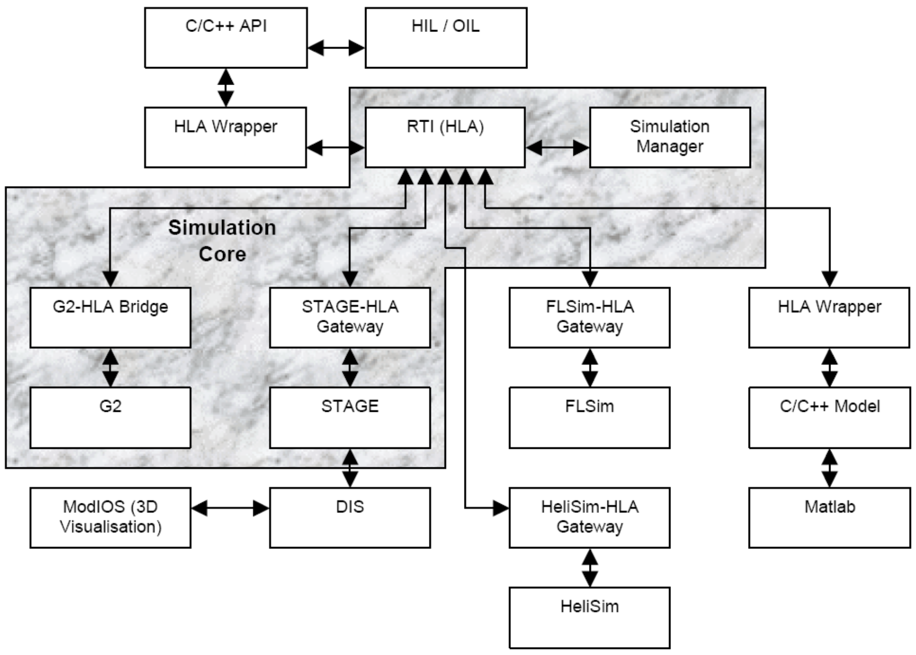

Figure 5.4: The VGD 2.0 Architecture
Taken from (le Roux, 2002), with permission.

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

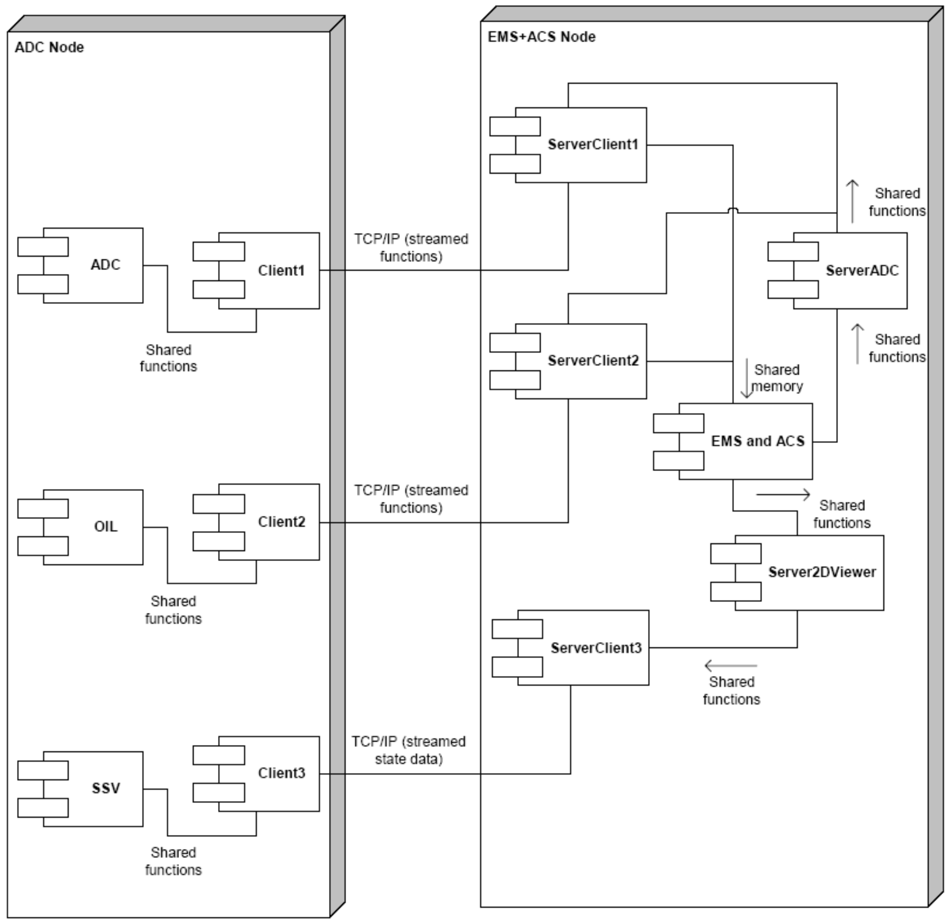

Figure 5.5: The VGD 3.0 Architecture
Taken from (Duvenhage and Senekal, 2004), with permission.

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.

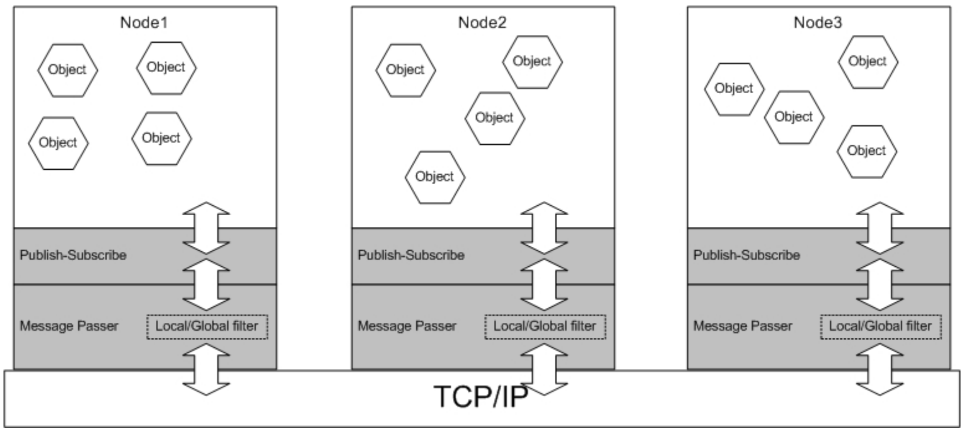

Figure 5.6: The VGD 3.1 Architecture
Taken from (Duvenhage and le Roux, 2007b), with permission.

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.

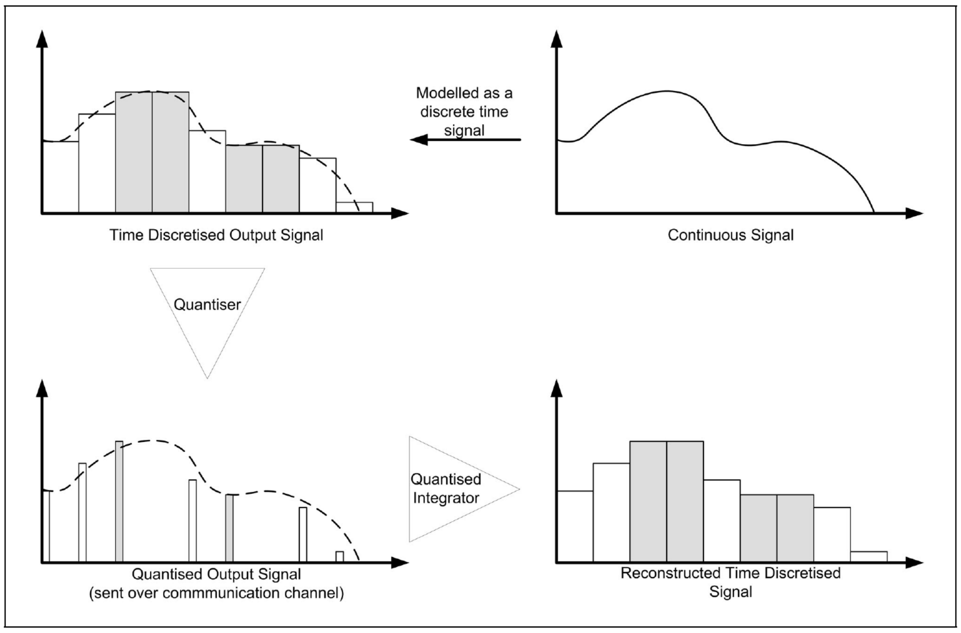
Modelled as a discrete time signal
Time Discretised Output Signal
Continuous Signal
Quantiser
Quantised Integrator
Quantised Output Signal
(sent over commmunication channel)
Reconstructed Time Discretised Signal

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).

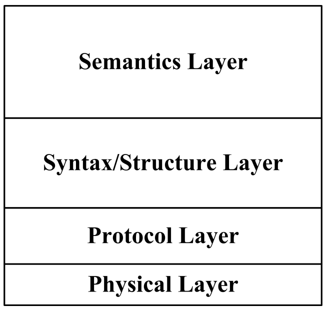

Figure 6.1: The Interoperability Layers

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).

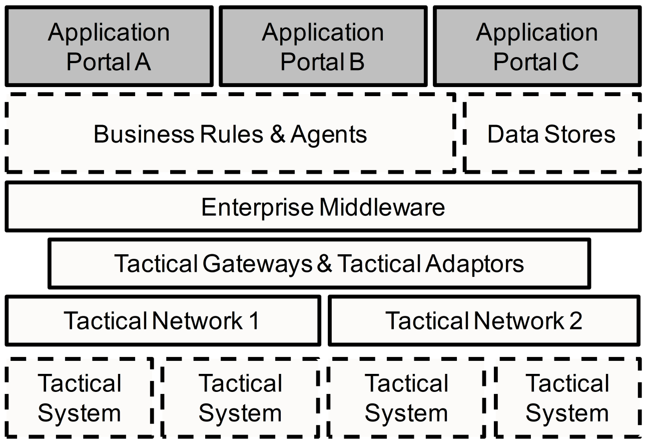

Figure 6.2: The Tactical Network and C2 Enterprise

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.