Performance Analysis of Live-Virtual-Constructive and Distributed Virtual Simulations: Defining Requirements in Terms Of Temporal Consistency

DISSERTATION

Douglas D. Hodson, Civilian, USAF

AFIT/DCE/ENG/09-25

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this dissertation are those of the author and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the United States Government.

AFIT/DCE/ENG/09-25

Performance Analysis of Live-Virtual-Constructive and Distributed Virtual Simulations: Defining Requirements in Terms Of Temporal Consistency

DISSERTATION

Presented to the Faculty

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

Douglas D. Hodson, B.S. Physics, M.S. Electro-Optics, MBA

Civilian, USAF

December 2009

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT/DCE/ENG/09-25

Performance Analysis of Live-Virtual-Constructive and Distributed Virtual Simulations: Defining Requirements in Terms Of Temporal Consistency

Douglas D. Hodson, B.S. Physics, M.S. Electro-Optics, MBA

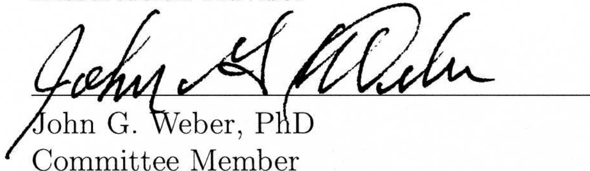
Civilian, USAF

Approved:

11/2/09

Rusty O. Baldwin, PhD
Dissertation Advisor

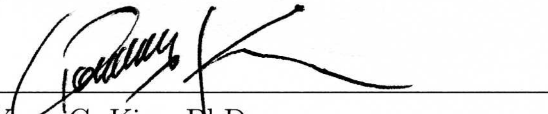
Date



10/26/2009

John G. Weber, PhD
Committee Member

Date



2 Nov 09

Yong C. Kim, PhD
Committee Member

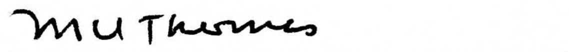
Date

2 Nov 09

Dursun A. Bulutoglu, PhD
Committee Member

Date

Accepted:



12 Nov 09

M.U. Thomas, PhD
Dean, Graduate School of
Engineering and Management

Date

AFIT/DCE/ENG/09-25

Abstract

In a live-virtual-constructive (LVC) environment, people and real system hardware interact with simulated systems. Introducing these real-world elements into the simulation environment imposes timing constraints which, from a software standpoint, places the design of LVCs into the class of real-time systems.

A distinguishing characteristic of LVCs is the relaxation of data consistency to improve the interactive performance and geographic scalability of the simulation. Relaxing consistency improves interactive performance since the simulation continues executing and responding to inputs without waiting for the most current shared data values. Scalability improves since live and simulated entities from distant geographic locations can be interconnected through relatively high latency networks.

LVCs are characterized as a set of asynchronous simulation applications each serving as both producers and consumers of shared state data. In terms of data aging, an LVC system is a first order linear system and the rate a consumer uses state data is irrelevant to the aging itself. Because of this, simple analytic models to estimate data aging based upon system architecture can be derived. An algorithm to compute, in real-time, the temporal consistency of state data for an LVC in operation is developed and the relationship between validity intervals and an LVC's systems parameters is defined.

To develop simulations that reliably execute in real-time and accurately model hierarchical systems, two real-time design patterns are developed: a tailored version of the model-view-controller architecture pattern along with a companion Component pattern. Together they provide a basis for hierarchical simulation models, graphical displays, and network I/O in a real-time environment.

Finally, the relationship between consistency and interactivity is established by mapping threads created by a simulation application to factors that control both interactivity and shared state consistency throughout the distributed environment.

This research extends the knowledge of LVCs and distributed virtual simulations (DVS) through detailed analysis and the characterization of the underlying computing architecture's effect on shared state consistency and interactive performance. System performance is quantified via two opposing factors; the consistency of the distributed state space, and the response time or interaction quality of the autonomous simulation applications. A framework is developed that defines temporal data consistency requirements such that the objectives of the simulation are satisfied.

Table of Contents

Page

Abstract iv

List of Figures ix

List of Tables xi

I. Introduction 1
- 1.1 State Space Consistency 3
- 1.2 Interaction Quality 3
- 1.3 Summary 4

II. Background 5
- 2.1 Terminology 5
- 2.2 Parallel and Distributed Systems 7
- 2.3 Analytic and Virtual Simulations 7
- 2.4 Distributed Virtual Simulation 8
- 2.5 Real-Time Systems 10
  - 2.5.1 Real-Time Communication 11
- 2.6 Consistency Models 12
  - 2.6.1 Temporal Consistency 13
- 2.7 Performance Analysis 15
  - 2.7.1 Models 16
- 2.8 Petri Nets 17
  - 2.8.1 Colored Petri Nets 19
  - 2.8.2 Simulation 20
- 2.9 Related Work 20
  - 2.9.1 CAVE Automatic Virtual Environment 20
  - 2.9.2 Narrative Immersive Collaborative Environment 23
  - 2.9.3 Soft Real-Time Database Systems 25
  - 2.9.4 Analysis of a Simulated Computer Network . . 28
  - 2.9.5 Consistency in DVS Applications 30
- 2.10 Summary 31

Page

III. LVC/DVS System Characterization 33
3.1 Modeling Time 33
3.2 Time Flow Mechanisms 34
3.3 System Under Study 36
3.3.1 Interaction with the Real World 37
3.3.2 Inputs and Outputs 38
3.4 Distributed Simulation 38
3.5 Dynamic Shared State 40
3.6 Performance vs Consistency 42
3.7 Sources of Inconsistency 44
3.7.1 Simulation Applications 44
3.7.2 Interoperability Communication 45
3.8 Temporal Consistency Model 47
3.8.1 Derived Data Objects 48
3.9 Classifying State Data 48
3.10 Summary 49

IV. State Space Consistency Model 50
4.1 Startup Dynamics 53
4.2 Analysis and Results 54
4.3 Analytic Model 57
4.4 Measuring Consistency 60
4.5 Generalized System Model 62
4.6 Relationship to Validity Interval 64
4.7 Application 64
4.8 Aerial Combat Example 66
4.8.1 Candidate System Design 66
4.8.2 Evaluation 66
4.9 Summary 67

V. Real-Time Design Patterns 69
5.1 Real-Time Concepts 69
5.1.1 Jobs 70
5.1.2 Periodic Task Model 71
5.1.3 Reliability 72
5.1.4 Utilization 72
5.1.5 Foreground/Background Systems 73
5.1.6 Rate Monotonic Analysis 73
5.1.7 Threads as Tasks 74

Page

5.2 Model-View-Controller Pattern 75
5.3 Multi-Threading 77
5.4 Component Pattern 78
5.4.1 Hierarchical Modeling 78
5.4.2 Partitioning Code 80
5.4.3 Scheduling Jobs 82
5.4.4 Modeling a Player 84
5.4.5 Graphics and Input/Output 84
5.5 System Abstraction 85
5.6 Estimating Performance 86
5.7 Consistency and Utilization 88
5.8 Summary 89

VI. Conclusion 90
6.1 Future Research 92
6.1.1 Determination of Validity Intervals 92
6.1.2 Data Consistency Monitoring 92

Appendix A. Petri Net Simulator 95
A.1 General Features 95
A.2 Software Organization 96
A.3 Execution and Analysis 97

Appendix B. Application of Design Patterns 99
B.1 Frameworks, Toolkits and Applications 100
B.2 An Object-Oriented Real-Time Framework 101
B.2.1 Object 102
B.2.2 Component 103
B.3 Simulation Architecture 104
B.4 Graphics Architecture 108
B.5 Device I/O Architecture 110
B.6 Fighter Cockpit 111
B.7 MQ-9 Ground Control Station 113
B.8 Group Command Post 114
B.9 Summary 115

Bibliography 116

List of Figures

Figure		Page
1.1.	Simulation Classification Framework	2
2.1.	Classes of Parallel and Distributed Computers	6
2.2.	Simple Graph	17
2.3.	Petri Net Example	18
2.4.	CAVE Automatic Virtual Environment	21
2.5.	A Real-Time Database System	26
2.6.	Absolute and Relative Consistency	27
2.7.	Simulated Computer Network	29
3.1.	Time Flow Mechanisms	34
3.2.	Time-Stepped State Space (adapted from Fujimoto [Fuj00])	35
3.3.	Event-Stepped State Space (adapted from Fujimoto [Fuj00])	35
3.4.	Distributed Synchronous State Space Diagram	38
3.5.	Synchronous Distributed Simulation	39
3.6.	Asynchronous Distributed State Space Diagram	39
3.7.	Distributed State Space	43
3.8.	Multi-Threaded MVC Pattern	45
4.1.	LVC Model	50
4.2.	Producer Model	51
4.3.	Network Model	51
4.4.	Consumer Model	52
4.5.	Mean Worst-Case Age (ms) (T1=50Hz)	59
4.6.	Standard Deviation (ms) (T1=100Hz, T3=5ms)	60
4.7.	Distributed State Space Data	60
4.8.	Latency Classification & OSI Model	63
4.9.	HLA-based Communication	63

Figure | Page

4.10. Computing System Latency . . . 65
5.1. Release Time and Deadline Relationships . . . 70
5.2. A Periodic Task with 3 Jobs . . . 71
5.3. Example Usefulness Function . . . 72
5.4. Model-View-Controller Pattern . . . 75
5.5. Simulation Pattern . . . 76
5.6. Hierarchical Player Model . . . 79
5.7. Structural Composite Pattern [GHJV95] . . . 80
5.8. Component With Partitioning Support . . . 81
5.9. Example Component Models . . . 81
5.10. Cyclic Scheduler Structure . . . 82
5.11. Component with Scheduling Support . . . 83
5.12. Graphic and Network Classes . . . 84
6.1. Experiment Planning Flowchart (adapted from [BCE+06]) . . . 93
A.1. PT Workbench . . . 96
A.2. Petri Net Editor . . . 97
A.3. Software Organization . . . 98
B.1. OpenEaagles Packages . . . 100
B.2. Component Tree . . . 104
B.3. Simulation Pattern . . . 105
B.4. Player Pattern . . . 106
B.5. Interoperability Pattern . . . 107
B.6. Graphics Class Hierarchy . . . 108
B.7. Device Class Hierarchy . . . 110
B.8. Generic Heads Down Display . . . 111
B.9. MQ-9 Ground Control Station . . . 112
B.10. Group Command Post . . . 113

List of Tables

Table Page

2.1. Analytic and Virtual Simulations 8

4.1. 4-Factor, 2-Level Design 54

4.2. 4-Factor, 2-Level Results 55

4.3. 4-Factor, 2-Level ANOVA 56

4.4. 3-Factor, 3-Level Design 57

4.5. 3-Factor, 3-Level ANOVA 57

4.6. Computing System Worst-Case Analysis 67

Performance Analysis of Live-Virtual-Constructive and Distributed Virtual Simulations: Defining Requirements in Terms Of Temporal Consistency

I. Introduction

Live-virtual-constructive (LVC) simulations and distributed virtual simulations (DVS) are software systems that create an environment where multiple users interact with each other in real-time, even though they may be located around the world. In this context, real-time means time with respect to the simulation's progress and is synchronized with "wall clock" time. "Distributed" in this context refers to a number of heterogeneous computers located in different geographic locations connected by a network.

Participants interacting with the simulated environment could include pilots flying fighter aircraft, operators controlling an early warning radar system in an Integrated Air Defense System (IADS), or even a person playing the game HALO where the objective of the simulation (game) is less about representing an accurate picture of the real world and more about providing an exciting "virtual world" for entertainment. For simulations that include live assets, participants can also include real system hardware.

LVC and DVS systems include assets or entities from three distinct classes of military simulations: live, virtual, and constructive. In a live simulation, real people operate real systems. For example, a pilot launching weapons from a real aircraft at real targets for the purpose of training, testing, or assessing operational capability is a live simulation. In a virtual simulation, real people operate simulated systems or simulated people operate real systems. For example, a pilot flying a simulated

Human \ System	Real	Simulated
Real	Live	Virtual
Simulated	Virtual	Constructive

Figure 1.1: Simulation Classification Framework

aircraft, launching simulated weapons at simulated targets is a virtual simulation. In a constructive simulation, simulated people operate simulated systems.

Figure 1.1 provides a conceptual framework to classify these simulations based upon the types of entities they include. Entities in a live simulation include real people and real systems. Entities in a virtual simulation include simulated systems operated by real people. The entities in a constructive simulation are completely simulated by computer models and are often referred to as "computer generated forces."

While categorizing simulations into three distinct classes is useful, in practice it is problematic because there is no clear division between these categories – the degree of human participation in the simulation is variable, as is the degree of system realism [DoD97]. Because of this, many simulations are actually hybrid systems that contain a mix of entity types. This is particularly true for virtual simulations which routinely include both virtual and constructive entities. LVC simulations are typically assumed to include a broader scope of entities than DVS systems by directly incorporating live assets into the interactive environment.

To create a context for the environment, a hybrid simulation is assembled from a collection of autonomous distributed simulation applications which we refer to as an "LVC" or an "LVC simulation." Within the LVC, individual entities, vehicles and weapon systems are generated by specific simulation applications responsible for sharing current state information through a network.

In an LVC, the "system under study" is often a "system of systems" which includes humans and/or operational system hardware. Because these real-world elements are present, timing constraints are imposed on the simulation environment

which, from a software standpoint, places the simulation into the class of real-time systems.

1.1 State Space Consistency

LVCs operate by passing state data between distributed simulation applications. As a result, a fundamental conflict arises in LVCs; simulation applications require state data that is not locally managed to produce correct outputs. A conflict arises because, in many situations, the application cannot wait for the most current data and still meet real-time interactive response time constraints. If the distributed processes are connected via a network infrastructure with a relatively high latency, data transmitted by one application might be considered inconsistent or "too old" by the time it is received. This inconsistency in state data is a distinguishing characteristic of LVCs which not only must be recognized but also managed to harness the realism and power LVCs can provide. This research characterizes the consistency of the distributed state space and provides a framework to define consistency requirements relative to the objectives of the system.

1.2 Interaction Quality

Both LVC and DVS systems include people or real system hardware interacting with a simulated system. In either case, the software system (the simulation) interfaces and interacts with driving functions (input signals) [CK06] generated by a person or hardware component and responds by producing outputs. For a typical flight simulator, interaction includes input from stick and throttle devices and output in the form of graphical displays.

Because of this, the performance characteristics and requirements of LVC and DVS systems differ from discrete-event and parallel discrete-event simulations as the former places a much greater emphasis on interaction. As a result, each have different performance parameters and metrics to gauge efficiency. Each also provides an efficient solution for different kinds of simulation and modeling objectives. For virtual

simulations, it is not sufficient to simply consider performance characteristics such as speedup and throughput; rather, emphasis is placed on response time or interaction latency. Interaction latency is the time delay between a user providing input to the system and experiencing the result of that input. In virtual simulations, the response time is a hard constraint due to the modeling requirements and the characteristics of the system under study. This stands in stark contrast to non-real-time constructive simulations, where response time is not an issue as there are no human or hardware interactions. Software systems designed to meet latency requirements due to real-world interactions fall into the class of real-time systems which has several accepted software organization paradigms.

1.3 Summary

This research quantifies the performance of LVC and DVS systems in terms of two opposing factors; the consistency of the distributed state space, and secondly, the response time or interaction quality of the autonomous simulation applications. Furthermore, the performance of individual autonomous distributed simulation applications is considered by abstracting the essential architectural features of the distributed applications into well-defined object-oriented design patterns. The design patterns are then used as a basis to estimate performance using rate-monotonic principles.

II. Background

Designing and building reliable high quality LVC and DVS systems is challenging due to the number of disciplines a simulation engineer needs to understand. This includes programming, operating systems, networks, real-time system development and simulation. The following sections cover the domains relevant to this discipline followed by a section on related work.

2.1 Terminology

The terminology associated with simulation systems can be confusing as there are subtle differences between the use of certain terms. This section defines terms as they are used throughout this document.

- Model - a physical, mathematical, or otherwise logical representation of a system, entity, phenomenon, or process [DoD97].
- Simulation - a method for implementing a model over time [DoD97].
- Live Simulation - a simulation involving real people operating real systems [DoD95].
- Virtual Simulation - a simulation involving real people operating simulated systems. Virtual simulations inject human-in-the-loop in a central role by exercising motor control skills (e.g., flying an airplane), decision skills (e.g., committing fire control resources to action), or communication skills (e.g., as members of a C4I team) [DoD95].
- Constructive Model or Simulation - models and simulations that use simulated people operating simulated systems. Real people provide inputs to such simulations, but are not involved in determining outcomes [DoD95].
- Networked Virtual Environment (net-VE) - a software system in which multiple users interact with each other in real-time, even though those users may be located around the world [SZ99].
- Distributed Interactive Simulation - a time and space coherent synthetic representation of world environments designed for linking the interactive, free-play

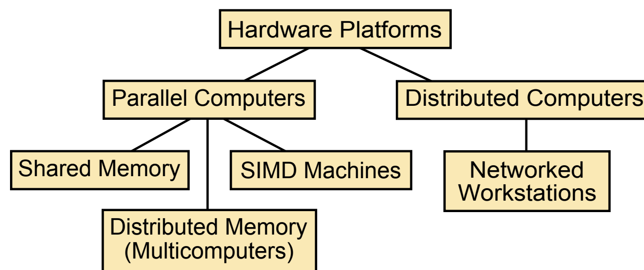


Figure 2.1: Classes of Parallel and Distributed Computers

activities of people in operational exercises. The synthetic environment is created through real-time exchange of data units between distributed, computationally autonomous simulation applications in the form of simulations, simulators, and instrumented equipment interconnected through standard computer communicative services. The computational simulation entities may be present in one location or may be distributed geographically [IEE95].

- Collaborative Environment - a space in which multiple users share and modify the state of a set of common objects (information) in real-time [Kol03].

The difference between virtual simulations, networked virtual environments and distributed interactive simulation is subtle. Each consists of people interacting with a real-time system that provides a context in which to participate. The term Distributed Interactive Simulation can be used generically, but it usually associated with simulations built using the Distributed Interactive Standard (DIS) standard.

The term "constructive" implies a simulation without interactive participation by a human. These systems might be designed to run in real-time or designed to run "as fast as possible" and generate output results from a set of input files. This research considers the execution of distributed simulation systems in real-time.

Collaborative environments often are not simulations at all. The term can mean a system designed for multiple users to interact using a common set of data.

2.2 Parallel and Distributed Systems

There is a distinction between parallel and distributed systems. As the taxonomy in Figure 2.1 suggests, distributed simulation typically involves a set of heterogeneous workstations connected through a network, interacting to create a simulation system. The workstations are heterogeneous because they may be using different operating systems and computing platforms. Parallel computers are typically more homogeneous in design and are usually connected through higher speed, lower latency networks. Whereas parallel computers are typically located together in the same room or building, distributed computers are often located at different geographic locations around the world. This research is primarily concerned with distributed computers connected through a network.

2.3 Analytic and Virtual Simulations

Historically, two classes of simulation applications have received the most attention: analytic simulations and virtual environments [Fuj00]. Characteristics that distinguish these different domains are summarized in Table 2.1.

While the central goal of analytic simulations is to capture detailed quantitative data concerning the system being simulated, the goal in most virtual-environment simulations to date has been to give users the look and feel of being embedded in the system being modeled [Fuj00].

Analytic simulations are intended to study the system being simulated. Human interaction, in any form, ranges from limited to none.

Virtual simulations have typically been oriented towards studying the interactions of the operators with the system. In some cases, however, the purpose of the simulation is to train the operator to perform some task using simulation as a means of interacting with a virtual environment or to make the simulation look and feel real [Ney97]. As such, it is not always essential for these simulations to exactly emu-

Table 2.1: Analytic and Virtual Simulations [Fuj00]

	Analytic Simulations	Virtual Environments
Execution pacing	Typically as-fast-as-possible	Real-time
Typical objective	Quantitative analysis of complex systems	Create a realistic and/or entertaining representation of an environment
Human interaction	If included, a person is an external observer to the model	People integral to controlling the behavior of entities within the model

late the actual system. If the differences between the simulated world and the actual world are not perceptible to human participants, this is usually acceptable.

2.4 Distributed Virtual Simulation

The origins of DVSs can be traced back to 1983 and the development of SIMNET (SIMulator NETworking) [MT95]. Originally developed for the Defense Advanced Research Projects Agency (DARPA), SIMNET was delivered to the U.S. Army in March 1990. At that time, the SIMNET network software architecture was proved scalable with some 850 objects (mostly semi-automated forces) at five sites [SZ99]. It's architecture has three basic components [SZ99]:

- An object-event architecture
- A notion of autonomous simulation nodes
- An embedded set of predictive modeling algorithms called "dead reckoning"

SIMNET served an important role in the development of distributed virtual simulations, but needed further refinement. For example, the packet formats and network software architecture was not documented sufficiently so others could use it. It also lacked generality.

These shortfalls were addressed by the creation of the Institute of Electrical and Electronics Engineers (IEEE) Distributed Interactive Simulation (DIS) network

software architecture standard. The standard provides all the information needed to build DIS-compliant simulations.

The DIS standard defines the Protocol Data Units (PDU) or the data messages passed between cooperating simulations. For example, a typical message in a DIS compliant simulation transmits an entity state PDU containing position, orientation, and entity velocity changes. With the advances in network bandwidth, latency, and computing power, it is not uncommon to implement large scale distributed simulations that involve thousands of entities using DIS protocols.

The principle goal in most DVSs is to achieve a "sufficiently realistic" representation of an actual or imagined system as perceived by the participants embedded in the environment [Fuj00]. What "sufficiently realistic" means depends on the underlying requirements of the system.

In many cases, requirements focus on the training activities of the participants. To improve the performance of the system, the "state" (i.e., data) of the simulation is replicated in a way that limits network activity thereby reducing the consistency of the data. Purposely allowing inconsistencies to enhance scalability is sometimes called a "dynamic shared state" [SZ99]. This inconsistency allows the system to scale so a larger number of entities can be represented and included within the simulation itself.

Consider, for example, two flight simulators each being flown by a pilot connected through a network. Further, assume an aerodynamics model samples pilot inputs and computes a new aircraft position and orientation at 50Hz. According to the DIS standard, calculated aircraft position would not have to be transmitted to the other simulator at 50Hz. In fact, it might be much less depending upon what maneuvers the pilot is engaging in, for example, if one pilot is flying without maneuvering, a new position need only be transmitted every few seconds.

One of the responsibilities of each simulator is to represent the environment in a manner sufficient to satisfy the requirements of the operator. So in the case

above, each simulation might estimate the other's position using "dead reckoning" algorithms. This calculated position might not be perfect (or even consistent with the true state), but it is likely accurate enough depending upon the underlying requirements of the system. This loosening of consistency allows more entities to interact over the same network.

It is useful to distinguish between *analytic* and *virtual* simulations because they have different objectives which leads to different requirements and constraints. Having stated this clear distinction does not account for the fact that simulation engineers routinely use systems designed for one domain in another to conduct simulation studies. For example, the best behavioral model of a pilot is a real pilot — no computer algorithm can match the real thing. So for some analytical studies in which the system under test involves a person, the constructs used to build virtual simulations might be interleaved with constructs used to build purely analytic simulations.

2.5 Real-Time Systems

Real-time systems have been studied extensively [Liu00, Lap04]. These systems differentiate themselves from other systems by not only completing tasks correctly, but also completing them within a certain time. In other words, they have the additional burden of ensuring tasks are executed in a manner that produces both correct results and meets timing deadlines.

Consider a pilot immersed in a virtual environment flying an aircraft. As the pilot is controlling the aircraft through stick and throttle inputs, the simulator must process those inputs, update the simulation state and possibly update visual displays within 100ms [IEE95]. If the simulator took, on average, considerably longer to respond to pilot inputs, the quality of the simulation would degrade, and certainly not "feel" like the real system. If the average response time of the system was 100ms, it would probably be considered acceptable. This means that on occasion, the system might take more time to respond to input changes, say 115ms. Timing requirements

in this form, where average response time is considered acceptable are said to be "soft".

Requirements in which a violation of a timing constraint is considered unacceptable, are considered "hard." For example, consider the release of a "dumb" bomb. A timing requirement might be specified such that the bomb must be released within, say 80ms, of button press. If it should release later than that a catastrophic event might result.

A central issue in the design of real-time systems is the scheduling of software tasks to ensure each task is executed in a manner such that timing constraints of all the tasks are met. To do this, tasks are classified into categories. A well-known deterministic workload model is the periodic task model [Liu00]. In this model, tasks are classified as:

- Periodic - a task where a computation or data transmission is executed at regular or semi-regular time intervals on a continuing basis. Periodic task timing deadlines are usually considered hard.
- Aperiodic - a task generated in response to unscheduled events. Work associated with aperiodic tasks have the same statistical behavior and the same timing requirements. The timing deadlines are soft.
- Sporadic - similar to aperiodic tasks except the timing deadlines are hard.

Recall that distributed virtual simulations are real-time systems because the human operator imposes timing requirements on the design of such systems as the example above illustrates. Fortunately, timing requirements associated with a human-in-the-loop tend to be soft.

2.5.1 Real-Time Communication. Communications in real-time distributed systems is different from communications in other distributed systems. While performance is always welcome, predictability and determinism are the real measures of success [Tan95]. LAN protocols whose performance is inherently stochastic, such as

Ethernet, are unacceptable because they do not have a fixed upper bound on transmission time [Tan95].

Since their advent, the transport protocols TCP and UDP, and the Internet protocol IP have served non-real-time applications well [Com06]. Yet these protocols are unsuitable for real-time applications for many reasons [Liu00]. The primary issue is the determination of an upper bound for data transmission. This requirement is met by using networks designed to provide these bounds such as token ring, or the use of protocols such as Time Division Multiple Access (TDMA) that inherently avoid collisions [Tan95] (i.e., they avoid what gives rise to the stochastic behavior of some networks). In addition, much work has also been done to generalize or extend rate monotonic scheduling theory to distributed systems that utilize these networks [SS93, SS95].

Despite the stochastic nature of Ethernet and the non-real-time characteristics of TCP and UDP, it is very common to implement distributed virtual simulations using them. In fact, the DIS standard assumes UDP is used to pass messages throughout the network. In many cases, the timeliness and reliability of UDP is considered "good enough" to meet requirements.

For example, the DIS standard specifies if a entity state packet arrival exceeds 300ms the receiving simulation should disregard it. As long as this does not occur frequently, the quality of the simulation is considered acceptable. More stringent consistency requirements for correct operation might demand other network structures for implementing a system design.

2.6 Consistency Models

One of the first steps in characterizing a distributed virtual simulation is the identification of the proper consistency model. A consistency model is a contract between software and memory [Tan95]. A wide spectrum of contracts have been defined, each with a different level of consistency.

In a single CPU system, the contract is inherently strict. In fact, a single CPU system implements "strict consistency". Formally this means that any read to memory location x returns the value stored by the most recent write operation to x [Tan95]. This ideal programming model is problematic to implement in multiprocessor systems, and strict consistency is virtually impossible to implement in a distributed system. To achieve it would imply a perfectly synchronized global clock and instantaneous updates to memory for all read and write operations.

A slightly weaker memory model than strict is "sequential" consistency. This form of consistency relaxes the notion of a global clock and simply states the result of any execution must be in some arbitrary but agreed upon sequential order [Tan95].

An even weaker memory model is called "causal" consistency. In this model, writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines [Tan95].

Implementing stronger forms of consistency involves considerable overhead due to the complexities of managing and coordinating access to shared memory or distributed shared memory. However, weaker consistency models increase the performance of parallel shared memory machines and the benefits increase as memory latency increases [Mos93]. In loosely-coupled systems, such as distributed computers connected through a network, intermachine message latency is considered large [Tan95]. This is why distributed virtual simulations implement what appears to be very weak forms of data consistency. Consistency models and their performance have been formally analyzed for distributed shared memories [Yan05] using read/write operations. Distributed virtual and collaborative environments often only update distributed data [Kol03]. This notion of consistency is little studied [Kol03].

2.6.1 Temporal Consistency. Temporal consistency models [SL92, SL95, KLA+03] have been used to evaluate the performance of soft real-time database systems. They offer a promising framework to characterize LVC and DVS systems.

Temporal consistency is defined in terms of the "age" and "dispersion" of data [SL95]. That is, the timing characteristics of data objects being read and written to by tasks. As such, it is an extension of the periodic task model presented earlier. In this extended model, each periodic task is either a read-only, write-only or update (read and write) transaction.

Consider a *write-only* transaction that models the periodic reading of a sensor (or the external environment) along with the updating of sensor values. The sensor values themselves are called "image" objects. These are also sometimes referred to as "base" data. Another example is the reading of stick and throttle inputs as commanded by a pilot.

An *update* transaction reads a set of data objects (which could include image or base data), computes, and writes to "derived" objects. A *read-only* transaction retrieves the values of a set of data objects but does not write to any data object.

As inputs are sampled, a sample time is associated with the image data. As a new value of an image is written, the older value of the image read by other transactions "ages". To capture the effect of aging, an image is viewed as having multiple "versions". Naturally, the faster the sampling, or the higher the sampling rate, the faster the image ages.

The age of data item x can be characterized by an aging function $a_t(x)$. The dispersion of two data objects is the difference between their ages. For example, if $a_t(x)$ and $a_t(y)$ are the ages of the objects x and y at time t, then the dispersion $d_t(x, y)$ would be $d_t(x, y) = |a_t(x) - a_t(y)|$.

Given a set Q of images and derived objects, Q is absolutely temporally consistent at time t if $a_t(x) \leq A$ where $A \geq 0$ for every x in Q, where A is an absolute threshold [SL95]. Q is relatively temporally consistent at time t if $d_t(x, y) \leq R$ where $R \geq 0$ for every two objects x and y in Q, where R is a relative threshold [SL95]. A set of data objects is temporally inconsistent if the objects are either absolutely or relatively inconsistent.

The thresholds A and R reflect the temporal requirements of the application, that is, how current and close in age the data must be for the results of computations based on them to be considered correct [SL95].

2.7 Performance Analysis

Assuming temporal model threshold requirements A and R meet different requirements of the system, there needs to be a way to evaluate the overall performance of a system design. Performance in this context quantifies how well a system meets its temporal requirements. In other words, given temporal thresholds or bounds, to what extent does the dynamic system stay within those bounds?

There are three ways to evaluate the performance of a system: measurement, simulation, and analytic modeling [Jai91]. Direct measurement could be done, but in this domain it would be rather expensive depending upon a number of factors and requirements. Even for a completely new system design it would be expensive to design and partition the software system into logical processes, and to assemble the necessary networks and hardware systems for a test. In other cases, direct measurements using simple tools like "ping" might be sufficient to estimate the performance of an existing network infrastructure. For example, if the temporal requirement is such that 300ms delays can be tolerated across a network connection (this is the DIS standard for "loosely coupled" interactions between entities [IEE95]), and a ping test on an existing network with a representative workload shows a maximum latency of 40ms, no further investigation might be deemed necessary. This is often the case for DVS systems designed for operator training. In fact, during the course of a simulation exercise, "ping" as well as other tools are routinely used to assess network performance.

Analytic modeling is another approach. Certainly "back of the envelope" estimates can be calculated, using network bandwidth and latency values, message transmission rates, and so on. But analytic models, of necessity, simplify the system. Thus, important characteristics of the system might be abstracted away such as the

asynchronous nature of LVC and DVS systems. In fact, asynchronous real-time systems are quite difficult to analyze [Liu00]. It has been said that analytical modeling of complex systems requires so many simplifications and assumptions that if the results turn out to be accurate, even the analysts are surprised [Jai91]. Since simulations can incorporate more details and require fewer assumptions than analytical modeling they are often closer to reality [Jai91].

This research uses simulation to estimate the performance of a new system design. That is, a system design in which no predetermined partitioning of software into autonomous applications has taken place. While simulation might be the preferred approach in a number of situations, it should not be used to the exclusion of direct measurement or analytical models. Each approach to evaluating performance has its merits. Depending upon requirements, one approach or another might be the most convenient or efficient at solving the problem. The use of multiple methods facilitates validation of performance estimates.

2.7.1 Models. To simulate a system design, a model of the system must be built. The model itself is a physical, mathematical, or otherwise logical representation of a system, entity, phenomenon, or process. The construction and validation of a system model offers a number of benefits including, insight into the design and operation of the system, a better understanding of the system under study, and it also reveals errors and ambiguities in the system design [Uni07].

After a model has been built, properties of the system can be evaluated. These properties tend to fall into the categories of functionality or performance. Functional properties include characteristics such as the absence of system deadlocks, whereas performance properties characterize some aspect of a system in operation.

Models themselves are described using particular languages. Most modeling languages can only be used to analyze either functional/logical properties or the performance properties of a model. To evaluate the performance properties, simulation

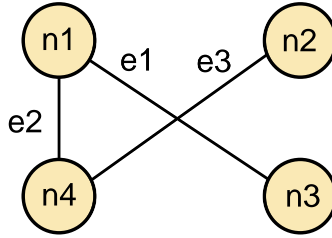


Figure 2.2: Simple Graph

is typically used. A simulation is a dynamic representation of a system model and models the execution of a system over time.

Simulations rarely provides exact answers, but it is possible to calculate how precise the estimates are. Simulation-based performance analysis of a model includes a statistical investigation of output data, the exploration of large data sets, the appropriate visualization, and the verification and validation of simulation experiments.

2.8 Petri Nets

Petri nets are a graphical and mathematical tool to model, analyze and simulate discrete-event systems and discrete distributed systems [Pet77]. They originated from the doctoral dissertation of Carl Adam Petri in 1962 [Pet62]. In a relatively short period of time, Petri nets were used extensively in practice as well as seeing continuing theoretic development.

A Petri net is a graph. That is, it is a set of nodes, edges and rules associating edges and nodes. Formally, a graph is defined as a triple $G = (N, E, \varphi)$ consisting of a set N of nodes, a set E of edges and a mapping φ of the elements of E to a pair of elements in N. Figure 2.2 shows a simple graph where $N = \{n1, n2, n3, n4\}$, $E = \{e1, e2, e3\}$ and a mapping function φ where

$$e1 \to (n1, n3),$$
$$e2 \to (n1, n4),$$
$$e3 \to (n2, n4).$$

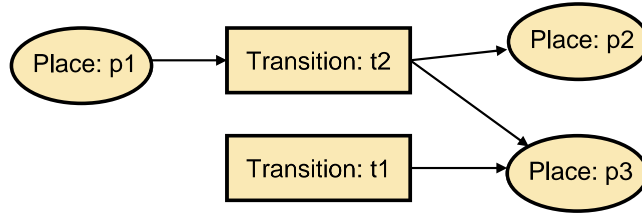


Figure 2.3: Petri Net Example

An undirected graph models symmetric relationships while a "directed" graph or digraph models asymmetric relationships. For a directed graph, the first node of the ordered pair is the tail of the edge, and the second is the head; together they constitute endpoints. We say that an edge is an edge "from" its tail "to" its head [Wes01]. The terms "head" and "tail" come from the arrows used to draw digraphs.

A Petri net has two types of nodes: places and transitions. Places are graphically represented by ellipses and transitions by rectangles. Petri net edges are referred to as arcs and are always directed (i.e., they have a head and tail and are drawn as an arrow). Formally, a Petri net is a 4–tuple $PN = \{P, T, I, O\}$, where P is the set of places, T is the set of transitions, $I(p, t)$ is mappings from $P \times T$ and $O(t, p)$ is mappings from $T \times P$. The element $I(p1, t1)$ is 1 if the Petri net has an arc from $p1$ to $t1$ and 0 otherwise. Likewise, the element $O(t1, p1)$ is 1 if the Petri net has an arc from $t1$ to $p1$ and 0 otherwise.

Figure 2.3 is a Petri net illustrating these definitions. In Figure 2.3, $P = \{p1, p2, p3\}$, $T = \{t1, t2\}$ and the elements of I and O are

$$
\begin{array}{lll}
I(p1,t1) = 0 & I(p2,t1) = 0 & I(p3,t1) = 0 \\
I(p1,t2) = 1 & I(p2,t2) = 0 & I(p3,t2) = 0 \\
O(t1,p1) = 0 & O(t2,p1) = 0 & \\
O(t1,p2) = 0 & O(t2,p2) = 1 & \\
O(t1,p3) = 1 & O(t2,p3) = 1. &
\end{array}
$$

The marking of the Petri net is a specification of how many tokens there are at each P. Formally, it is a mapping of $P \rightarrow \{0, 1, 2, ...\}$. Markings represent the state of a Petri net. A transition associated with inputs P_i is enabled if there is at least one token in each P_i. When a transition fires, one token is removed from each P_i and one token is added to each output place, P_o. That is, state changes are produced by the firing of transitions. Representing a system as a Petri net is a straightforward way of analyzing system properties using formal mathematics without becoming "bogged down" in the details of what the places and transitions represent [WCPW05].

2.8.1 Colored Petri Nets. Colored Petri nets (CP-nets or CPN) provide a complete language for the design, specification, simulation, validation and implementation of large software systems [Jen97b]. It is, in particular, well suited for systems in which communication, synchronization and resource sharing are important. Typical application areas include communication protocols, distributed systems, embedded systems, automated production systems, work flow analysis and VLSI chips [Jen97b].

The development of CP-nets has been driven by the desire to develop a modeling language – at once theoretically well-founded yet versatile enough to be of practical use in systems of the size and complexity found in typical industrial projects. To achieve this, CP-nets combine the strength of Petri nets with the strength of programming languages. Petri nets provide primitives for the description of the synchronization of concurrent processes, while programming languages provide primitives for the definition of data types and the manipulation of data values [Jen97b].

CP-nets were introduced by Jensen [Jen97a, Jen97b, Jen97c, Jen97d] as an extension to the basic Petri net definition. They broaden the range of problems that can be described and analyzed graphically. Petri net places contain tokens that are indistinguishable, and it is only the number of tokens in a place that is important. Colored Petri nets introduce distinguishable (colored) tokens which reduces the size of the model by reducing redundant Petri net structures to distinguishable tokens in a common structure.

CP-net places have a data type (color set) and all of the tokens in a place have the data type of the place. The values (colors) of the data type distinguish one token from another. A place has a multi-set of tokens, which means the tokens in a place do not have to have different values. Arc expressions dictate the number and values of the tokens removed from the input places and the number and values of the tokens created in the output places. There is no requirement that tokens be conserved, although in many cases tokens represent physical objects so conservation of tokens is modeled. It is also clear that the input and output places for a transition may be different, and so the output tokens may differ from the input tokens in both data type and color.

What is crucially important is that CP-nets are a type of graph, that they have a formal definition and that an architecture described as a CPN can be analyzed using graph theory [WCPW05]. As such, the CPN modeling language can investigate both functional/logical properties and performance properties of a model [Uni07].

2.8.2 Simulation. A large body of performance analysis research uses a variety of Petri net and Petri net-related formalisms [Wel02]. Most of this research solves analytical models that are automatically generated from the Petri net models. However, the size and complexity of CP-nets make the generation and solution of analytical models from CPN models prohibitive [Wel02]. Therefore, performance analysis of CP-nets uses simulation to determine performance.

2.9 Related Work

The first part of this chapter presented an overview of the domains relevant LVC and DVS systems. This section presents related work from published papers and dissertations.

2.9.1 CAVE Automatic Virtual Environment. The CAVE Automatic Virtual Environment (better known by the recursive acronym CAVE), shown in Fig-

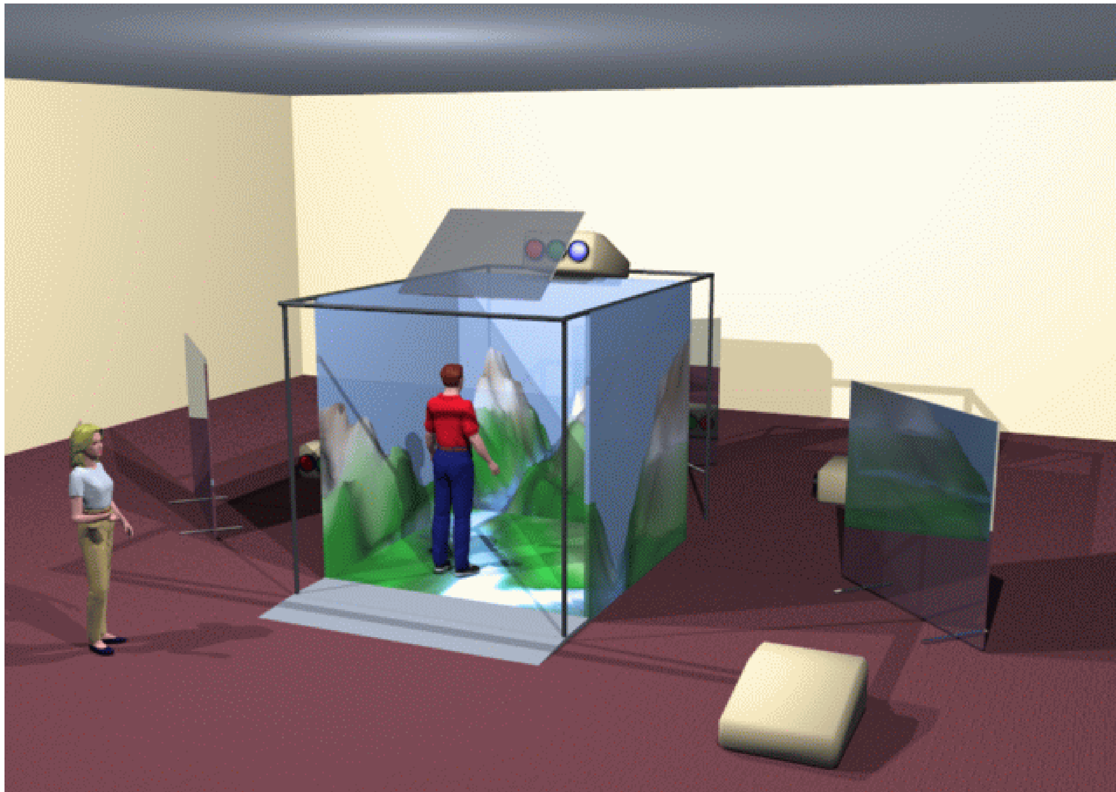


Figure 2.4: CAVE Automatic Virtual Environment [Wik07]

ure 2.4, is a surround screen, projection-based virtual reality environment system [Wik09]. The actual environment is a 10x10x10 foot cube, where images are rear-projected in stereo on 3 walls (front wall, left wall, and right wall), and down-projected onto the floor. (The floor can be considered a floor wall for a total of 4 walls.) The 4 walls display computer generated stereo images of the virtual world in real-time based on the position and orientation of the users head and hand in the CAVE. The viewer wears LCD shutter glasses to mediate the stereo images. The viewers head and hand position and orientation are tracked through sensors on the shutter glasses and on the CAVE input device. The viewer can grab and move objects in the virtual world with the wand [ZMD99].

The CAVE system is composed of multiple hardware and software components that operate asynchronously, such as sensors, image computation and rendering processes, and analog-to-digital converters. Mascarenhas [MKBK98] used a timed exten-

sion of Petri nets to model and analyze the CAVE virtual environment. At the time (1998), numerous techniques using Petri nets for the automatic analysis of general concurrent and real-time systems were in use. However, these techniques and tools had not been applied to modeling and analysis of virtual reality systems [MKBK98].

Mascarenhas wanted to gauge the usefulness of Petri nets for modeling concurrency and the real-time performance of virtual environments. Time was modeled by adding a static delay interval $\tau = [a, b]$ to each transition $t \in T$. A static delay is bounded by two numeric constants, a, and b, with $0 \leq a < +\infty$ and $a \leq b \leq +\infty$. State changes occur by firing "fireable" transitions. A transition is said to be "enabled" when all its input places have at least one token. A transition with delay interval $\tau = [a, b]$ is fireable if it is continuously enabled for at least a, but not more than b, time units.

Of the 48 places and 35 transitions used to model CAVE as a Petri net, only a few of the transitions included a non-zero delay to account for time. These transitions modeled the time to determine a persons head and wand position. Head and wand position were determined by a system that pulsed receivers mounted on the head tracker and wand and communicated results via a 33.6 Kbaud serial line. Time to compute the images to be projected on the screens was also modeled with non-zero transition delays.

After the CAVE model was built, simulation and automatic verification experiments were performed. Using automatic verification, deadlock avoidance was established. However, this automatic verification result could only be performed for experiments of 40ms or less. Automatic verification of the model beyond 40ms became problematic due to "state space explosion." State space explosion occurs when trying to evaluate concurrent asynchronous systems. As time advances, the potential number of system states increases rapidly, thus making it difficult to evaluate all possible states in a timely manner. This is the principle reason for resorting to simulation and statistical analysis to evaluate modeled systems.

A series of experiments in which the delay associated with different transitions (for example, the delay associated with reading the head tracker or the time it takes to render a new image) was modified and simulated to observe the effect on system performance. These experiments uncovered a flaw in the way a particular shared buffer was used by CAVE processes. One of the main conclusions drawn from this work is that Petri net-based tools can effectively support the development of reliable virtual environments. Another result is the realization that automatic verification of models that incorporate time might not be possible due to the state space explosion problem.

2.9.2 Narrative Immersive Collaborative Environment. The Narrative Immersive Constructionist/Collaborative Environments (NICE) project at the Electronic Visualization Laboratory at the University of Illinois at Chicago, is a collaborative learning environment: a virtual garden, where children learn and garden cooperatively. In NICE, children located in distributed virtual environments (e.g., CAVEs), can take care of a virtual garden together in the center of a virtual island. The children, represented by avatars, collaboratively plant, grow, and pick vegetables and flowers. They make sure plants have sufficient water, sunlight, and space to grow, and they keep hungry animals away from sneaking in the garden and eating the plants [ZMD99, RJL+97].

NICE is a network of CAVE systems [YZD00] using a central server to simulate the garden and maintain consistency across the participating virtual environments, and a repeater to broadcast avatar state information. Each virtual environment (VE) sends local avatar information (the local tracker data) to the repeater via UDP, as well as the information about the local childs world-changing activities to the central garden via TCP. The central server receives the world-changing messages from each client, updates the world state and sends the new world information (the information about the garden) to each client via TCP so that all clients have the same world information [ZMD99].

Petri nets have been used as a formal modeling and analysis technique to evaluate the NICE system. Standard practice for net-VEs design is basically trial and error, empirical, and lacks any formal foundation [YZD00, ZMD99]. By applying formal modeling techniques such as Petri nets, design principles for net-VEs could be established, and the usefulness of formal validation and verification techniques demonstrated.

To model and analyze real-time systems, various timed extensions of Petri nets have been proposed. However, many real-time systems have temporal uncertainty. For example, the time to render an image in a VE system varies based on the complexity of the geometric objects in the image and network delays in net-VEs vary widely [YZD00]. To model temporal uncertainties in real-time systems, Murata [Mur96] proposed Fuzzy-Timing High-Level Petri nets (FTHNs) using fuzzy set theory. FTHNs model temporal uncertainties in real-time systems, and provides possibility distributions of events. Thus, FTHNs can capture all temporal uncertainties in CVEs and would be suitable models for CVEs.

Using this time model, a model of the CAVE system was built which improves on earlier work. A model of the NICE system, a network of two CAVEs connected through several UDP and TCP channels was also built. The UDP protocol was modeled as a single transition associated with a fuzzy time. TCP had a more detailed model. This protocol evaluated the response time for an avatar's movement in one client to show up on the other client's display. Other simulation tests changed the frequency of updates being sent through TCP channels to update the central server.

This work validated Petri nets usefulness for studying real-time behavior, network effects, and performance (latency and jitter) of net-VEs. Furthermore, the TCP protocol, while reliable, greatly increases the average network latency and jitter. This research recommended the design of a new transport layer protocol, which transmits shared state information with less latency and jitter than TCP. This is not surprising,

as much research was found proposing real-time protocols more suitable for CVEs (or distributed virtual simulations).

2.9.3 Soft Real-Time Database Systems. A real-time database system (RTDB) is often used in a dynamic environment to monitor the status of real-world objects and discover "interesting" events [KLA+03]. For example, a program trading application monitors the prices of various stocks, financial instruments, and currencies, looking for trading opportunities. A typical transaction might compare the price of Euros in London to the price in New York and, if there is a significant difference, rapidly executes a trade.

The state of a dynamic environment is often modeled and captured by a set of base data items within the system. Changes to the environment are represented by updates to the base data. In a dynamic environment, an entity changes its state in either a continuous or a discrete fashion. Changes to an entity are continuous if the state of the entity is constantly changing. Base items that model continuous entities must be periodically updated. On the other hand, changes to an entity are discrete if the changes occur at distinct instants of time. Maintaining the temporal consistency of discrete objects in soft real-time database systems has been studied by Kao [KLA+03].

Figure 2.5 shows the relationship between the dynamic environment and updates to a set of base items. When a base data item is updated to reflect external activity, the related views need to be updated or recomputed as well. Application transactions generate the ultimate actions taken by the system. These transactions read the base data and views to make their decisions [KLA+03].

Temporal consistency refers to how well data in a RTDB models the actual state of the environment. Temporal consistency consists of two components: absolute (or external) consistency and relative consistency. A data item is absolutely consistent (fresh) if it accurately reflects the state of an external object that the data item models.

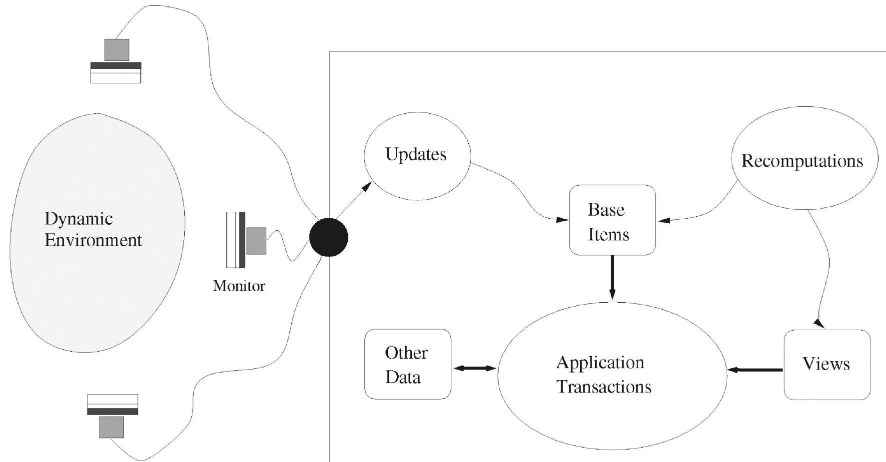


Figure 2.5: A Real-Time Database System [KLA+03]

Data items are relatively consistent if they are temporally correlated to each other. A data object is temporal if its value changes with time. Based on how the value changes, we can classify temporal data objects as continuous or discrete objects. Most previous work on temporal consistency maintenance concentrates on systems with continuous objects [KLA+03].

With discrete objects, the value of an entity remains unchanged until the next update arrives. The update arrives at a discrete point in time and the arrival pattern is sporadic. Unlike continuous objects, it is difficult to suitably define aging for a discrete object since the object changes its state at an unpredictable rate. In other words, it is difficult to define an aging function $a_t(x)$ because the value of the data might not in fact be old.

To formally define a notion of temporal consistency, Kao [KLA+03] introduced the concepts of the "version" and "validity" interval which are defined below.

Definition 1 (Version). A version x of a data item d is a value of the external object that d models. Every time the external object changes its value, a new version of d is generated. Each version x is thus associated with a time interval that specifies when the version is valid. This time interval is the validity interval of x denoted by $VI(x)$. $VI(x)$ has a lower bound $LTB(x)$ and an upper bound $UTB(x)$. $LTB(x)$

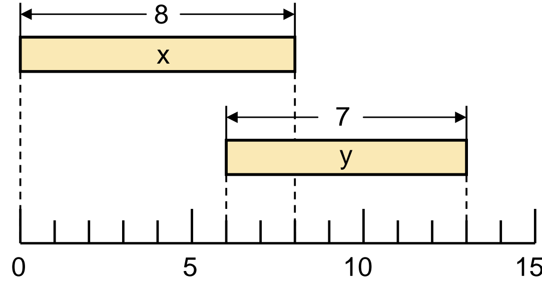


Figure 2.6: Absolute and Relative Consistency [KLA+03]

is the instant an update of d with $x's$ value arrives. $UTB(x)$ is the instant the next update of d arrives.

Definition 2 (Current version). The current version of an item is a version x_i such that its validity interval contains the current time instant t_c, i.e., t_c is in $VI(x_i)$.

Definition 3 (Absolute consistency). A discrete data item d is absolutely consistent if, at any time instant, a current version for d can be found in the system.

Definition 4 (Relative consistency). Given a set of item versions R, the versions in R are said to be relatively consistent if $\bigcap \{VI(x_i) \,|x_i \in R\} \neq \emptyset$.

To clarify this terminology, consider two discrete data objects x and y shown in Figure 2.6. Data item x arrives at $t = 0$ and y at $t = 6$. Data items x and y become stale at times 8 and 13 respectively. Using the notation of validity intervals, assume the current version of x is x_m and the current version of y is y_m. Then, if $VI(x_m) = [0, 8]$ and $VI(y_m) = [6, 13]$, then x is absolutely consistent during $[0, 8]$ and y is absolutely consistent during $[6, 13]$. Also notice that x_m and y_m are relatively consistent in the time interval $[0, 8] \bigcap [6, 13] = [6, 8]$.

The reason for defining discrete data objects and their temporal correctness criteria is that the entities in Kao's research could not be represented by continuous objects since he needed to maintain a particular level of consistency in real-time databases and considered the scheduling and efficiency of executing update transactions (so called "application transactions") and their impact on the database.

This is relevant to LCS and DVS systems because some of the data passed between applications is discrete or sporadic in nature. For example a DIS "fire" or "detonation" event. This data is not transmitted on even a quasi-periodic basis such as DIS entity state PDUs, but rather, is transmitted in a sporadic manner. Thus, the temporal requirements of discrete events need to be considered.

2.9.4 Analysis of a Simulated Computer Network. In 2002, the U.S. Army Simulation, Training and Instrumentation Command (STRICOM) and the Computer Engineering Department of the School of Electrical Engineering and Computer Science at the University of Central Florida (UCF) began a joint project to assess the "Bandwidth and Latency Implications of Integrated Training and Tactical Communication Networks." The research evaluated the network requirements for conducting mission planning and rehearsal while enroute to deployment [VDGG04]. OMNeT++ [OMN09] was used to evaluate network bandwidth and latency characteristics for different system designs and to quantify the amount of traffic that could be expected in a rehearsal mission. DIS PDU data was generated and captured by running a predefined "vignette" with the OneSAF Testbed Baseline (OTB).

The vignette consisted of a network that linked 8 airplanes, a ground station, a satellite, and 3 wireless channels as shown in Figure 2.7. The first wireless link connects the routers in all the planes to each other. A second wireless link connects the routers in the planes to the satellite, and the third connects the satellite to the ground station. In this way, each router is connected to three different links, and the satellite is connected to two. During an actual mission rehearsal, the airplanes, satellite and ground station are not simulated, but real physical assets are used.

Input data for the OMNet++ [OMN09] simulation comes from the data collected by the OneSAF logger (i.e., the expected network data generated during a real rehearsal). Timestamps on each DIS PDU is the time the entity generated the PDU and put it into the output queue for transmission.

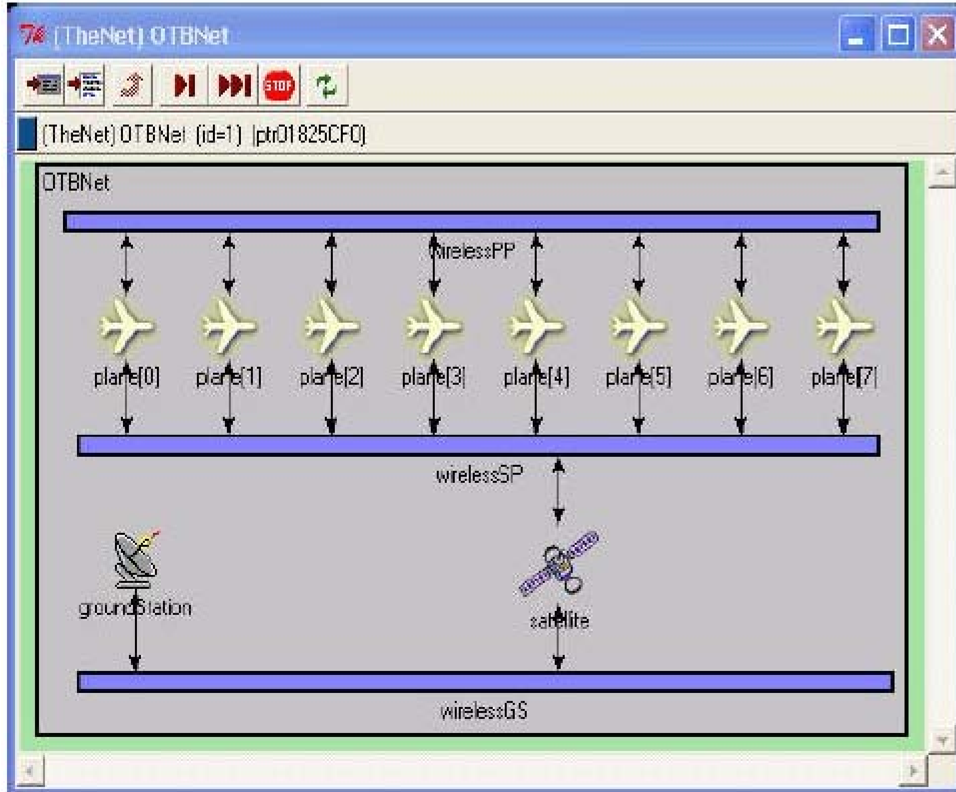


Figure 2.7: Simulated Computer Network

To estimate bandwidth requirements, a separate program calculates the minimum instantaneous bandwidths by dividing the total simulation time into smaller time intervals of 2 seconds each and computes the ratio of volume of data transmitted in each interval to the length of the interval. In this static analysis, overhead due to retransmissions, packet losses, or collisions is not considered. Therefore, the resulting bandwidth estimates can be interpreted as an absolute lower bound for the actual required bandwidth. This approach is a simple yet effective method for estimating bandwidth requirements.

Slack time analysis determines if the channel bandwidth is enough to transmit the required PDUs without delay. The slack time for each node generator is defined as the difference between the timestamp of each PDU and the current simulator time the instant the PDU is read from the input file. If the difference is positive, the generator is ahead of the planned schedule, otherwise it is behind. Thus, a negative

slack time indicates channel bandwidth is insufficient to transmit the required PDUs without delay [VDGG04]. Results for this study indicated that the 64 Kbps wireless channel connecting the ground station to the satellite was in fact insufficient.

Travel time analysis looks at the total latency to transmit a PDU from a source to a sink node. Travel time is the difference between the sending time of a PDU from a node generator and the arrival time at a node sink. All the transmission times, propagation times and waiting times in router queues are part of the travel time. The travel times of most of the PDUs on the 64 Kbps channel were completely unacceptable. Some PDUs took more than 100 seconds to arrive.

A queue length analysis determines the number of messages waiting to be transmitted. As expected, the size of the queues associated with transmission across the 64 Kbps channel was unacceptable, with as many as 3000 messages awaiting service. Also expected was the unacceptable number of collisions from nodes attempting to gain access to the 64 Kbps channel.

This research is important from a number of perspectives. The use of a simulator to generate expected traffic for an estimate of bandwidth requirements is practical and useful. This directly relates to tradeoffs on how a software system could be partitioned. Performing "first-order" estimates of the bandwidth requirements using the expected traffic could be applied in the domain of LVC and/or DVS simulations. Using either OMNeT++ or Petri nets to evaluate more precisely the latency characteristics of a particular design also confirms a later recommendation about the design of LVC simulations that indicate which system designs should be considered "candidates" for further analysis before deployment.

2.9.5 Consistency in DVS Applications. The issue of consistency in distributed interactive applications and its effect on entity position has been studied [ZCLT04, ZCLT01]. Zhou's work defines a metric to measure the time-space inconsistency of entities within a distributed virtual environment (DVE). The metric evaluates the time-space consistency property of a DVE considering clock asynchrony,

message transmission delay, the accuracy of a dead reckoning algorithm, the kinetics of the moving entity, and human factors. The quality or goodness of the DVE is based upon a human characteristic related to visual perception time for spatial information. While this work is important and the analysis impressive, it's limited to a single metric concerning spatial entity position consistency and its impact on the DVE relative to human response traits.

Improving consistency by delaying or purposely degrading the response time of individual simulation applications in a DVE has also been studied [Qin02]. A new consistency model named the "delayed consistency model" provides a framework to evaluate the tradeoff between consistency and response time. An acceptable compromise between consistency and response time is hard to determine – poor responsiveness with few inconsistencies or a large number of inconsistencies with a short response time [Qin02].

The preceding work was leveraged to develop a conceptual model for consistency maintenance in DVE/DVS/LVC environments [Hl04]. This conceptual model is based upon the human nature of the participants (i.e., human perceptual limitations, area of interest management, and visual and temporal perception).

2.10 Summary

The architecture for LVC and DVS systems can be traced back to 1983 and the development of SIMNET. While there are more options now in terms of interoperability protocols, fundamental limits of sharing data between a set of autonomous simulation nodes remains the same. How to improve the consistency of shared entity state data using predictive modeling dead reckoning algorithms and other techniques has been studied. No general underlying framework to specify consistency requirements was found.

In the domain of real-time databases, methodologies to evaluate the performance of soft real-time database systems using temporal consistency of stored data has

been studied. The fundamental notion that the performance of these systems can be improved by relaxing the consistency of the stored data has application in the domain of real-time distributed simulation. This work provides a basis for a general framework to describe LVC and DVS data requirements.

Petri nets provide a means of studying the temporal properties of CAVE and NICE environments and a sound methodology to study the temporal properties of LVC and DVS systems more generally. The essential architectural features of LVC/DVS systems that affect shared state consistency are modeled so that system properties and factors can be studied.

III. LVC/DVS System Characterization

Many characteristics about "what" a LVC is, and "how" LVCs operate are known; however, no research was found that provides a formal characterization of these systems. This is likely the result of the application domain of interest, namely, training systems where the quality of the the simulated environment is judged by more subjective measures such as a "good enough" look and feel. Human factors provides measures of "look and feel" that are derived from experimental data. This chapter provides a detailed characterization of important properties of an LVC.

3.1 Modeling Time

Simulations have several notions of time. Fujimoto [Fuj00] provides the following useful definitions.

- Physical time - time with respect to the physical system being simulated. For example, consider a simulation of a battle that took place during the Civil War in 1864. The time associated with a specific scenario as executed in this case, the year 1864, is the physical time.
- Wallclock time - refers to time during the execution of the simulation program. In other words, this is the actual time the simulation is executing, for example, from 4 pm on May 5th to 5 pm on May 5th or precisely 3600 seconds.
- Simulation time - an abstraction used by the simulation to model physical time. It represents the total elapsed time since the simulation started. For example, a double floating-point variable could be used to measure time where 0.0 corresponds to the start of simulation execution and 1.0 represents one second of simulation time. One "second" of simulation time does not necessarily correspond to one second of wallclock time.

From the perspective of real-time system theory, wallclock time is *real-time.* It is continuous and advances at a constant rate. In a virtual (or real-time) simulation, the goal is to ensure that *simulation time* advances and stays in sync with the wallclock.

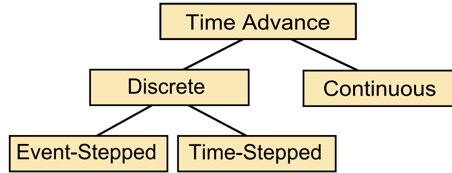


Figure 3.1: Time Flow Mechanisms

Additionally, a virtual simulation must respond to inputs within prescribed *response times*. Levying these time constraints on the simulation system effectively places the software design of virtual simulations into the class of real-time system design.

3.2 Time Flow Mechanisms

Modeled systems can be viewed as the collection of variables necessary to describe the system state at a particular time relative to the objectives of a study [LK00]. As simulation time advances, state variables change depending upon the time advance mechanism used. State variable change can be represented by a state space diagram. There are several different time advancement mechanisms used in simulations, commonly referred to as the *time flow mechanism*. The general relationship between time advancement mechanisms is shown in Figure 3.1.

In a continuous simulation, the simulation is executed on an analog computer and the state space changes continuously. A discrete simulation can be executed on a digital computer and its state space changes at discrete points. The two most common types of discrete simulations are *time-stepped* and *event-stepped* (or *event-driven*).

In a time-stepped simulation, state space variables are updated at fixed intervals as shown in Figure 3.2. Not every state variable is modified at each time step, but when a state variable is modified, it occurs at a fixed interval. As the figure shows, some variables are updated every other step, or even every fourth step depending upon modeling requirements. The duration of the *step size*, as shown in Figure 3.2, is based on the nature of the specific activity or activities a system developer considers important [GL00].

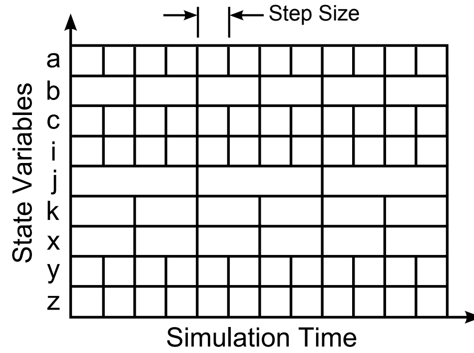


Figure 3.2: Time-Stepped State Space (adapted from Fujimoto [Fuj00])

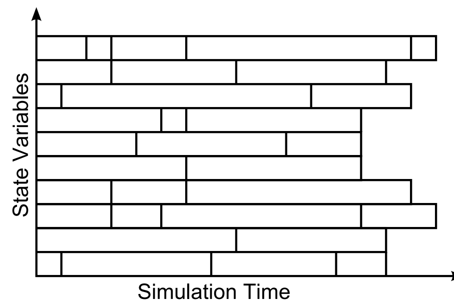


Figure 3.3: Event-Stepped State Space (adapted from Fujimoto [Fuj00])

In an event-stepped simulation, state space variables are updated only when "something interesting" occurs as defined by a model developer. This is sometimes referred to as an asynchronous update and is depicted in Figure 3.3. For a discrete-event simulation, events for a given process or system are based on the activities the model developer deems important [GL00]. See Narayanan [NSP+97] for additional simulation types defined on the basis of state space and timing considerations.

Both event-stepped and time-stepped simulations are used extensively. The choice of which time advance mechanism to use, and which is most efficient, is determined by the *system under study*.

The computational overhead associated with discrete-event simulations arises from detecting and recording events that determine the next time step. However, this overhead is more than compensated for by not having to execute the model at every

timestep [GL00]. Time-stepped simulations, on the other hand, have computational overhead whether state variables are modified or not.

In general, a time-stepped simulation constitutes the logical choice for processes with activity distributed over every timestep; discrete-event simulation is most efficient for activity that is sparsely distributed over time [GL00]. More information can be found in [GL00].

3.3 System Under Study

Understanding the nature of the *system under study* frames the modeling requirements and the challenges virtual simulations must address. Specifically, it determines the most efficient choice between implementing an event-stepped or a time-stepped simulation. Information on this topic, however, is widely scattered in literature and thus, not available in a concise and consistent location [CK06]. Below, we focus on key system characteristics that contrast the usefulness of both time advance mechanisms.

Consider an example that expands upon Fujimoto's [Fuj00] simulation of an aircraft flying from New York to Los Angeles. Suppose the arrival of the aircraft at Los Angeles is part of the *system under study.* For this requirement, a discrete-event simulation might compute the total flight time of the aircraft, and advances the aircraft's position and simulation time immediately to the time the aircraft reaches Los Angeles since this is where "something interesting" occurs. Computing intermediate aircraft positions are not needed (i.e., they are deemed uninteresting and irrelevant). Modeling this system using a discrete-event paradigm is quite efficient because the simulation itself can make relatively large jumps in *simulation time* to the next "interesting" event.

Now consider a virtual simulation in which a pilot is inserted into the simulated aircraft flying from New York to Los Angeles. In this case, the *system under study* might include not only the arrival at Los Angeles, but also an evaluation of how well

the pilot flies the route from New York to Los Angeles. In this case, the pilot is part of the system under study.

Inserting a pilot in the loop creates several additional requirements: the simulation must execute synchronously with wallclock time (i.e., in real-time) and must respond to pilot inputs in a timely manner. To complicate matters further, the simulation will likely be required to simulate the flight dynamics of the aircraft so the simulation feels correct to the pilot. This, in turn, requires real-time flight dynamics calculations be performed at a relatively high frequency, with a bounded deterministic execution time.

3.3.1 Interaction with the Real World. A key characteristic of any virtual simulation is its interaction with the *real-world*. In the above example, the pilot is part of the *system under study*. There are a variety of reasons to insert an operator or pilot into the simulation. The most common is operator training. Another is to conduct human factor studies, such as an evaluation of an aircraft instrument display. In other instances, the insertion of a well-trained pilot adds fidelity to the simulation rather than a computerized, possibly simplistic behavioral model of a pilot.

In virtual simulations, placing a person or hardware into the system under study puts an additional burden on the simulation as it must respond to inputs and generate outputs in a timely manner. An important attribute that describes this relationship is the *response time*. Response time is the time between the presentation of a set of inputs to a system (release) and the realization of the required behavior (completion).

Inputs are generated by a wide variety of devices including control stick changes, keyboards, touch screen displays and even the reception of data via a network interface. Responses come in the form of generating audio, video, or motion cues. Responses that affect hardware assets come in many forms.

It can not be over-emphasized that if these response times are not adequately satisfied, the person or hardware included in the *system under study* might not respond correctly, thus degrading the validity of the simulation itself.

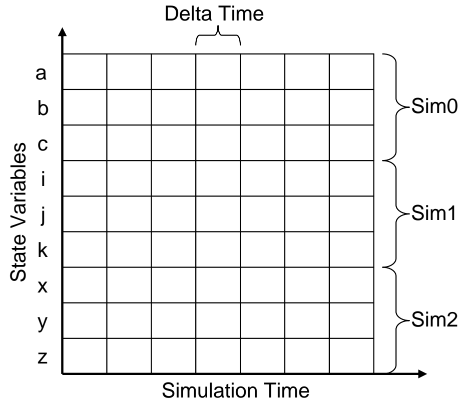


Figure 3.4: Distributed Synchronous State Space Diagram

3.3.2 Inputs and Outputs. Responding to simulation inputs by generating correct outputs in a timely manner is specified by the response time parameter. For example, the Federal Aviation Administration (FAA) requires a response time of less than 125 ms for flight simulators. If this requirement is not met, the simulator will feel sluggish to the operator. Additionally, a response time greater than 125 ms may contribute to pilot induced oscillation (PIO) effects in aircraft performance.

To meet this requirement, real-time systems sample inputs and generate outputs. The process of sampling itself fits with the organization of time-stepped simulations. The rate at which sampling takes place is application dependent. Because a human is *in-the-loop* controlling the vehicle as simulation time advances, the forces affecting position, velocity and acceleration are not, indeed cannot, be known in advance. In other words, where the vehicle will be in the future is unknown. Because of this, the modeled activity is distributed over time which provides a solid rationale for advancing time in a time-stepped manner as most LVC simulators and simulations do.

3.4 Distributed Simulation

Distributing a time-stepped simulation across multiple computers involves dividing and partitioning the state space into several individual simulation applications that communicate and share information. Each application is responsible for main-

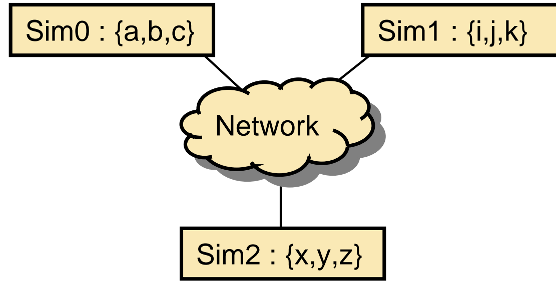


Figure 3.5: Synchronous Distributed Simulation

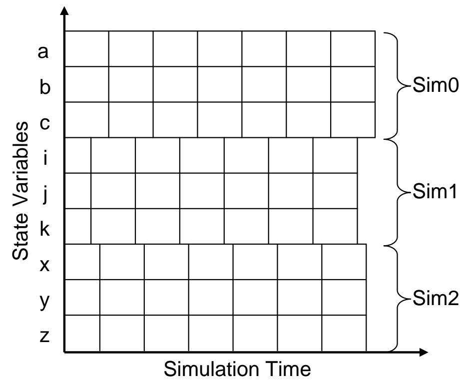


Figure 3.6: Asynchronous Distributed State Space Diagram

taining its own partitioned state variables. If a synchronization mechanism advances time in a coordinated manner, the resulting state space of the entire distributed simulation state space can be viewed as shown in Figure 3.4. In this diagram, the three simulation applications (Sim0, Sim1, Sim2) individually maintain three state space variables of interest.

Physically, the simulations themselves are hosted on different computers and communicate via a network infrastructure like that shown in Figure 3.5. In reality, most LVC simulations do not execute synchronously as shown in Figure 3.4, rather, they run asynchronously with respect to each other as notionally shown in Figure 3.6.

Each simulation in Figure 3.6 is executing a time-stepped simulation with its own state space. Typically these simulations are “loosely coupled” so they can execute as autonomously as possible (which also improves the performance and scalability of

the system). Some degree of interaction will always take place in the form of sharing state space information. If this were not the case, the simulation would no longer be considered distributed, it would simply degenerate into a stand-alone independent simulation. This sharing of state space information is achieved by each process by sending messages relevant to other processes through the network as needed. The message transmission delays are non-deterministic and can be on the order of hundreds of milliseconds. Because of this large delay, the totally distributed architecture used in LVC simulations is considered a good architectural choice to increase scalability [DG99]. Unfortunately, this choice also increases the chance of inconsistent states being shared between simulation nodes.

3.5 Dynamic Shared State

Distributing a time-stepped simulation across multiple computers can be viewed as a partitioning of the state space among two or more autonomous simulation applications, where each application is responsible for maintaining its own local state and replicating state space data required by and managed by other applications. This "dynamic shared state" [SZ99] constitutes the information that multiple simulation applications must maintain about the distributed environment. This sharing of locally managed state space information is achieved via messages sent through a network infrastructure.

Communication between applications is facilitated by a number of interoperability protocols and Application Programming Interface (API) standards to so-called "middleware." Middleware is connectivity software with a set of enabling services that allow multiple simulation applications to interact across a network. The Distributed Interactive Simulation (DIS) [IEE98] is a good example of a well established protocol, whereas, the Data Distribution Service (DDS) [OMG09], the High-Level Architecture (HLA) [IEE00] and the Test and Training Enabling Architecture (TENA) [DoD02] are representative middleware solutions.

Communication is facilitated by a number of defined protocols or standards. Two well defined standards are listed with a short description below.

- Distributed Interactive Simulation (DIS) - In DIS, simulation state information is encoded in formatted messages known as Protocol Data Units (PDUs) and exchanged between hosts using existing transport layer protocols. Normally broadcast UDP is used. The current version of the DIS application protocol defines 67 PDU types arranged into 12 families. Frequently used PDU types are listed below for each family.
  - Entity information/interaction family - Entity State, Collision, Collision-Elastic, Entity State Update
  - Warfare family - Fire, Detonation
  - Logistics family - Service Request, Resupply Offer, Resupply Received, Resupply Cancel, Repair Complete, Repair Response
  - Simulation management family - Start/Resume, Stop/Freeze, Acknowledge
  - Distributed emission regeneration family - Designator, Electromagnetic Emission, IFF/ATC/NAVAIDS, Underwater Acoustic, Supplemental Emission/Entity State (SEES)
  - Radio communications family - Transmitter, Signal, Receiver, Intercom Signal, Intercom Control
  - Entity management family
  - Minefield family
  - Synthetic environment family
  - Simulation management with reliability family
  - Live entity family
  - Non-real time family

- High Level Architecture (HLA) - HLA is a general purpose architecture for distributed computer simulation systems. Communication between simulations is managed by a runtime infrastructure (RTI). HLA consists of the following components:
  - Interface specification. The interface specification document defines how HLA compliant simulators interact with the RTI. The RTI provides a programming library and an application programming interface (API) compliant to the interface specification.
  - Object Model Template (OMT). The OMT specifies what information is communicated between simulations and how it is documented.
  - HLA Rules. Rules that simulations must obey to be compliant to the standard.

Regardless of the standard used, state information from logical processes are replicated throughout the simulation system, and that state data usually is inconsistent compared to the true system state.

3.6 Performance vs Consistency

A fundamental conflict arises in LVCs when executing simulation applications require state data not locally managed, yet must also respond to inputs and produce correct outputs in real-time based, in part, on that data. The conflict arises because of network latency. If the network has a relatively high latency, data transmitted by an application might be inconsistent or "too old" when received.

Even so, to improve the performance and scalability of such systems, the state space of simulation applications is purposely allowed to become inconsistent. Performance is improved because each application continues executing and responding to local inputs without waiting for "consistent" data to arrive. As data consistency is relaxed, scalability improves since, in general, this allows more applications from more distant geographic locations to be interconnected. Thus, portions of the state

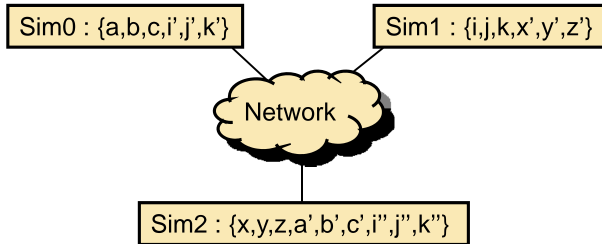


Figure 3.7: Distributed State Space

space within any particular simulation, at a given point in time, will not necessarily be consistent with the true state.

Figure 3.7 shows an LVC composed of three simulation applications where Sim0 locally manages state space variables {a,b,c} and replicates the state space variables managed by Sim1, namely {i,j,k} as data is received from the network. Simulation Sim2 manages its own state space {x,y,z} and replicates the state space of both of the other applications.

Because the system is executing in real-time and synchronized to the wall clock, replicated state spaces often contain data that is older or “aged” compared to the most current value. For example, since Sim0 is receiving updates via a network, its perception of Sim1’s state space is inconsistent with Sim1’s true state (i.e., it lags behind) because simulation time advances in lock step with the wall clock and cannot be stopped or paused to wait for an update to ensure consistency. As the local state variables of Sim1 change, it takes a finite amount of time to update Sim0.

Depending upon modeling and simulation requirements, this inconsistency might be acceptable. In fact, a DIS compliant simulation almost always works with aged data. As an example, the position of dynamic entities moving in the simulation rarely corresponds with the true position. The DIS standard specifies that for “loosely” coupled interactions, entity position (state) updates received within 300ms of when they are sent are acceptable. To reduce network traffic, updates are not sent on a regular basis; they are only sent when certain error thresholds are exceeded. The “old” or

aged data is considered good enough for the receiving application to estimate the true state values.

The level of inconsistency tolerated is based upon the accuracy of the estimation that can be made with older data, and secondly, the underlying requirements of the simulation itself.

3.7 Sources of Inconsistency

For a typical LVC simulation, sources of data inconsistency are introduced by the architecture of the distributed simulation applications and the interconnecting network infrastructure. The combination of the simulation architecture and the communication architecture is called the *system design*. The architectural characteristics of both the simulation applications and the communication mechanism to quantify and estimate their effect on data aging.

3.7.1 Simulation Applications. The model-view-controller (MVC) architectural pattern is a well established structure representative of the design of a typical simulation application. A central feature of the design pattern is the separation of user interface logic from the "domain logic" which performs calculations and stores data.

In the context of virtual and constructive simulation applications, the domain logic are the simulated systems and state variables, while the graphical displays and I/O functionality represent the view and controller components as shown in Figure 3.8. For live simulations, the domain logic often represents the periodic sampling of state information from live assets for the distributed environment.

The controller in the MVC pattern is typically associated with processing and responding to events that induce state changes in the model. Using this approach, the construction of a simulation application would, ideally, consist of a loop that, for each frame, sequentially reads inputs, executes system models, and generates outputs (i.e., updates graphics and processes network activities). In other words, this pattern is

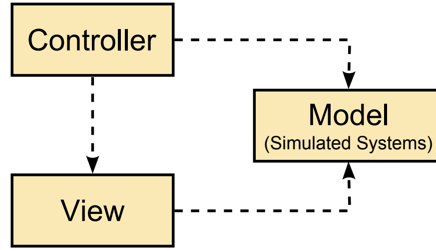


Figure 3.8: Multi-Threaded MVC Pattern

typically implemented as a single-threaded application whereby only the model (i.e., domain logic) makes state changes in response to controller events.

Doing this in real-time, however, is limited by processing power. This limit becomes ever more problematic as frame rates increase, thus reducing the amount of time available to complete all tasks (i.e., model updates, graphic drawing and the processing of network activity). To further complicate matters, in the domain of an LVC, the latency associated with state data moving through the network imposes an upper bound on frame rates if data is to have some level of consistency across applications.

To resolve this fundamental conflict, a multi-threaded variant of the MVC pattern is often used [RHJ+09] whereby the processing of models, graphics, and network I/O is done in separate threads, either synchronized with each other, or purposely allowed to execute asynchronously at assigned priorities. Typically, system state is executed by a high priority thread, while graphics and network threads are set to a lower priority. This multi-threaded variant MVC pattern improves system response time with respect to a human participant, but at the cost of data consistency.

3.7.2 Interoperability Communication. LVC simulations are often implemented using dedicated networks to control utilization and provide enhanced data security. Controlling utilization is an important aspect of characterizing LVCs because it is assumed network utilization has little or no effect on network latency. This

is true as long as the network is not operating at high utilization rates exceeding 50–60%.

Since LVCs are time-sensitive applications, they often transmit state data using connectionless, efficient, transmission protocols, such as the User Datagram Protocol (UDP). Because state space data updates are broadcast regularly, reliability is sacrificed to reduce overall latency. Furthermore, UDP and other connectionless protocols support the simultaneous distribution of state data using unicast, multicast or broadcast addressing schemes.

Because the interaction of LVCs is affected by network latency, and communication latency is increased by software complexity, middleware is often not the best solution to share state data in these environments [Hl04]. Therefore we initially restrict our attention to LVCs that share state data through well-defined protocols such as DIS. This restriction is relaxed later.

The DIS standard specifies a number of factors and performance requirements to minimize both network delay and network delay variance [IEE98]. For example, simulation state data is encoded into formatted messages, known as Protocol Data Units (PDUs) and exchanged using existing transport layer protocols; normally broadcast or multicast UDP. The size of DIS PDUs range from 80 to 200 bytes which is significantly smaller than the Maximum Transmission Unit (MTU) of 1,500 bytes for Ethernet and prevents packet fragmentation at the link layer as well.

Given these network environment characteristics, we model a dedicated LVC network as transporting state data according to well defined latency distributions with adjustable mean and shape parameters. Specifically, the message transmission delay between two nodes in the network may be modeled as a random variable which obeys the exponential distribution. Though the worst case transmission delay may be very large, it occurs with little probability, and the average delay is generally very close to the minimum delay between the two nodes [BG92, KSG99].

3.8 Temporal Consistency Model

Because LVC simulations often execute with inconsistent data, it is useful to characterize consistency in terms of correctness since any notion of data quality or correctness depends on the actual use of the data. In other words, data sufficient or accurate enough for one application might be insufficient for another.

We first define absolute consistency, then apply temporal consistency concepts developed by [KLA+03], [KS97], [Ram93] and [XLLG06] to the domain of LVCs as they provide a useful framework for defining system requirements in terms of correctness.

Definition *(Absolute Consistency)* Given a shared data object, θ, the state is absolutely consistent at any time t, if and only if $\forall i, j$, $1 \leq i, j \leq n$, $\theta_i(t) = \theta_j(t)$, where n is the number of nodes (i.e., simulation applications) in the LVC.

We say that the LVC is absolutely consistent if and only if every θ is consistent. In other words, an LVC is absolutely consistent if and only if the value of the replicated data objects as managed by each simulation application within the distributed system is consistent at all times.

Temporal consistency relaxes absolute consistency by defining the correctness of a shared data object, θ, as replicated by autonomous simulation applications as a function of a time interval. That is, the value of a shared data object is accurate or valid for a period of time after being updated.

Definition *(Temporal Consistency)* A shared data object, θ, is temporally consistent if its creation timestamp, θ_{TS}, plus the *validity interval*, θ_{VI}, of the data object is greater than or equal to current time t, i.e., $\theta_{TS} + \theta_{VI} \geq t$.

This notion of consistency is generalized for a distributed simulation consisting of n nodes, where each node defines a specific validity interval, $\theta_{i,VI}$, so that $\forall i$, $1 \leq i \leq n$, $\theta_{TS} + \theta_{i,VI} \geq t$. Because LVCs are connected via non-deterministic

networks, validity intervals include an acceptable reliability statistic (e.g., 95% of the time).

3.8.1 Derived Data Objects.

Another notion of temporal consistency is "relative consistency." It defines the accuracy or validity of *derived data* in terms of the relative creation times of the set of data used to produce it.

Definition *(Relative Consistency)* The set of data objects used to derive a new data object, θ, form a *relative consistency set*, R. Each set R has a positive validity interval, denoted by R_{VI}. A derived data object, θ, is relatively consistent if $\forall\ \theta \in R, |\theta_x - \theta_k| \leq R_{VI}$ where k is the cardinality of R.

Thus, temporal consistency is viewed as a *freshness* constraint and relative consistency is a *correlation* constraint [GHS95].

By defining LVC correctness requirements in terms of validity intervals for the shared data objects, we address the inconsistency in shared state data directly. Since the inconsistency in shared state data is the distinguishing characteristic of LVCs affecting performance and scalability, the validity intervals of state data directly relate to system performance. Furthermore, relaxing data consistency by increasing validity intervals improve both performance and scalability.

3.9 Classifying State Data

To define validity intervals, it is useful to classify state data as either "continuous" or "discrete." This classification is based upon what the data represents or *what* is being modeled, not the mechanics of how it is updated or processed by a digital computer. For example, the state describing the position of an aircraft in Cartesian coordinates would be considered continuous data, even though it is updated by a producer at some fixed frequency. The state data describing the position of a light switch would be considered discrete. After classifying state data, the determination of an appropriate validity interval defining LVC correctness is made.

For continuous objects, validity intervals establish correctness by bounding the difference (or accuracy) between a producers value and a consumers value [Ram07]. For discrete objects, validity intervals establish correctness based on timeliness; the interval specifies that a consumer may never be out of sync with the producer by more than a validity interval time [Ram07]. In other words, if the modeled system changes state, a consumer will receive the change no later than the time specified by the validity interval. Until that time, the replicated state within the consumer is simply incorrect. The impact of temporally incorrect discrete state data is especially important and should be carefully considered as the specified validity establishes "how long" incorrectness can be tolerated.

3.10 Summary

This chapter provided a detailed characterization of an LVCs important properties. While most of the material is known, it tends to be scattered throughout the literature. The definition of temporal consistency as applied to LVC and DVS systems is new and offers a promising approach for describing and evaluating system requirements.

IV. State Space Consistency Model

To understand the properties of an LVC, a conceptual model is built abstracting essential architectural features of simulation applications and networks that affect state space data timeliness. These effects are captured in a colored Petri net, which is particularly well suited for modeling systems in which communication, synchronization, and resource sharing are important. Jensen provides a comprehensive discussion of the theoretical foundations, analysis methods and the practical uses of colored Petri nets [Jen97a].

For the LVC architectural model, "colored" tokens include attributes to represent simulation state data and the timeliness of that data; "places" contain tokens, and "transitions" model the temporal properties of a particular system design. Time associated with transitions is specified by fixed or stochastic distributions. A "snapshot" of all places in the model constitutes the state of the system at a particular point in time.

When a transition creates a token, the token's "creation time" attribute is set. When a token is *moved* to a place, a second time-stamp attribute called "arrival time" is set. Time stamping tokens with both a creation and arrival time captures the temporal dynamics of a particular system design. The temporal properties of transitions constitute fundamental limits in a system design and their effect on data timeliness are of great interest.

Figure 4.1 is an abstract model of an LVC with a producer application distributing state data θ to other applications or consumers of that data. For both producer and consumer (i.e., simulation applications within the LVC), we assume the archi-

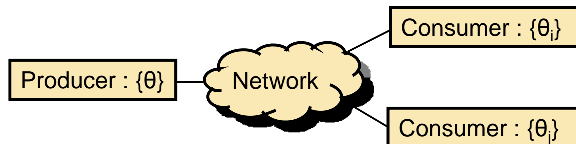


Figure 4.1: LVC Model

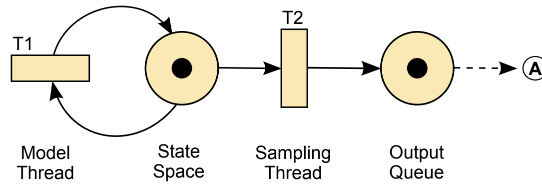


Figure 4.2: Producer Model

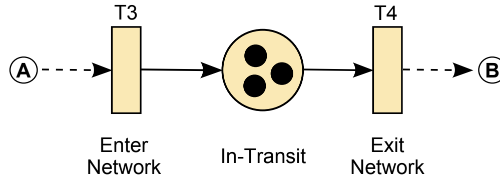


Figure 4.3: Network Model

tectural organization and execution of the simulation architecture is mapped to the multi-threaded MVC pattern.

Figure 4.2 is a Petri net model of the producer, where the simulated systems component from the MVC pattern periodically updates θ while a thread servicing the network periodically transmits updates to consumers. Threads periodically updating and transmitting the various θ's to consumers are asynchronous with respect to each other.

The Model Thread is represented by transition T1 that updates θ by replacing old state data (represented by a token) with new state data. The creation time attribute for this new token is set to the current simulation time. The State Space place holds state tokens. It represents local computer memory that stores θ. The Sampling Thread is represented by transition T2 which models the asynchronous reading of θ for transmission to other applications. This transition copies tokens from computer memory to the Output Queue. The Output Queue holds data to be transmitted by the network infrastructure. This queue could also represent a graphical display in the multi-threaded MVC pattern. The frequencies assigned to transitions T1 and T2 are the factors of most interest in the system design.

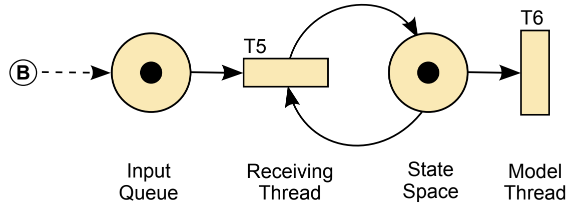


Figure 4.4: Consumer Model

Figure 4.3 models the network which moves θ from producer to consumer with a particular latency. Transition T3 captures the temporal characteristics of a network infrastructure in terms of a distribution, such as exponential or Pareto with location and shape parameters. The location represents the mean latency of moving the data from a producer to a consumer. Each data packet is assumed to be the same size and the latency includes sender overhead to submit and transmit the message, the time of flight (propagation time), the transmission time (message size divided by bandwidth), and receiver overhead.

The Enter Network transition time records the time tokens entering the network will arrive at their destination. This time is stochastically drawn from the above mentioned distribution and added to current simulation time. The Exit Network transition fires as soon as its enabling condition is satisfied – the moment simulation time advances to the earliest arrival time of any token at In-Transit. The distribution assigned to T3 is also a factor of interest in the system design.

Figure 4.4 models a consumer receiving θ and captures the essential feature of the Receiving Thread – the update mechanism for the dynamic shared state.

The Input Queue stores tokens arriving from the network. The Receiving Thread is represented by transition T5 which is enabled when the Input Queue has tokens. In this model, "old" state data already located in State Space place is replaced with "newer" data from the Input Queue the instant it arrives. In practice, transition T5 is implemented as a blocked thread waiting for data. The State Space place is a combination of data received from other producers and data locally managed by

the consumer. The consumer's Model Thread is represented by transition T6 which executes periodically.

Since T5 is enabled and fires immediately when tokens arrive for processing, it is not a factor of interest. Transition T6 on the other hand, determines when state data is available for consumer calculations, and is therefore a factor of interest.

4.1 Startup Dynamics

The temporal characteristics of transitions T1, T2, T3 and T6 affect the consumers replicated state space data timeliness and consistency. Since producers, consumers and their respective threads do not synchronize, but are executed on a periodic basis, there is a phase relationship between each periodically-executed transition. To capture this property of an LVC, phases ϕ_1, ϕ_2, and ϕ_6 are associated with transitions T1, T2, and T6, respectively and are modeled as random variables in the system. Because the phase relationship is relative, the model is simplified by arbitrarily selecting one phase, and setting it to zero. After some analysis ϕ_2 was chosen as the baseline phase since this led to clearer insight and intuition into the roles and relationship each factor contributes to data aging.

Preliminary exploration of the LVC system model with randomly assigned phases yielded important insight that produced a simpler model for simulation and analysis. For example, the sampling of the producer's state space data by T2 for distribution to consumers is equivalent to the periodic creation of a new token in the Output Queue whose age is drawn from a uniform distribution ranging from zero to the period of the producer's Model Thread T1. Also, the randomly assigned phase associated with transition T6 can be simplified by using a uniform distribution to determine *when* the consumer's Model Thread uses State Space data. The time associated with using state space data ranges from zero to the period of the consumer's Model Thread T6.

Table 4.1: 4-Factor, 2-Level Design

Factor	Levels
Producer Model Thread (T1)	50, 100 Hz
Producer Sampling Thread (T2)	5, 20 Hz
Network Latency (T3)	5, 100 ms
Consumer Model Thread (T6)	50, 100 Hz

4.2 Analysis and Results

A simulation of the Petri net-based LVC system model was used to study the factors that affect the temporal properties. This includes the frequency of the producer's Model and Sampling Threads, network latency characteristics as defined by a representative distribution, and the frequency of the consumer's Model Thread.

Exploration of the system model leads to the following two hypotheses concerning system dynamics. The first is that the period of the consumer's Model Thread, T6, does not influence data aging. In other words, the frequency at which the consumer uses replicated state data has no effect on the mean age and variance of that data. The second is, the mean age of the data used by a consumer in an LVC system can be estimated by adding each factor's individual contribution to aging. In other words, each factor's contribution to aging is independent, and there are no interaction effects.

To validate these hypotheses, a preliminary two-level full factorial screening experiment, as shown in Table 4.1, was performed to determine each of the four identified factor's influence on data aging including any second-, third- and/or fourth-order interaction effects. A two-parameter exponential distribution modeled network latency characteristics. For T3, Table 4.1 lists the location parameter for the distribution with a fixed standard deviation of $\pm$1ms.

For each simulation run, the mean age and standard deviation of the replicated state data as used by the consumer's Model Thread was computed. The simulation terminating condition was reached when the mean age was within $\pm$1ms of its true

Table 4.2: 4-Factor, 2-Level Results

Number	T1 (Hz)	T2 (Hz)	T3 (ms)	T6 (Hz)	$\overline{x}$ (ms)	s (ms)
1	100	20	5	100	35	15
2	100	20	5	50	35	15
3	100	20	100	100	130	15
4	100	20	100	50	130	15
5	100	5	5	100	110	58
6	100	5	5	50	110	58
7	100	5	100	100	205	58
8	100	5	100	50	205	58
9	50	20	5	100	40	16
10	50	20	5	50	40	16
11	50	20	100	100	135	16
12	50	20	100	50	135	16
13	50	5	5	100	115	58
14	50	5	5	50	115	58
15	50	5	100	100	210	58
16	50	5	100	50	210	58

age with 95% confidence. Table 4.2 contains the results of the screening design for three replications of the experiment.

Comparing the mean and standard deviation for each pair of runs (e.g., 1 & 2, 3 & 4, ...) in the table indicates the consumer's Model Thread (T6) might not play a role in state space aging. This is confirmed by the analysis of variance (ANOVA) results shown in Table 4.3 for mean data aging. Factors T1, T2, and T3 accounted for nearly all variation in aging with each having statistically significant p-values of zero. Factor T6, and just as importantly, all second-, third- and fourth-order interaction terms played no role in explaining variance.

This validates the first hypothesis and highlights an important characteristic of an LVC. From the standpoint of the consumer, the aging characteristics of the replicated state data is not influenced by the rate at which data is used. In other words, the frequency of the consumer's Model Thread (T6) does not play a role in the temporal characteristics of an LVC.

Table 4.3: 4-Factor, 2-Level ANOVA

Source	DOF	SS	MS	F	P
T1	1	303	303	2477.86	0.000
T2	1	67376	67376	550909.95	0.000
T3	1	108094	108094	883847.66	0.000
T6	1	0	0	0.16	0.695
T1*T2	1	0	0	0.11	0.747
T1*T3	1	0	0	1.44	0.239
T1*T6	1	0	0	0.88	0.356
T2*T3	1	0	0	0.13	0.725
T2*T6	1	0	0	0.14	0.710
T3*T6	1	0	0	0.22	0.645
T1*T2*T3	1	0	0	0.87	0.358
T1*T2*T6	1	0	0	0.19	0.669
T1*T3*T6	1	0	0	2.27	0.141
T2*T3*T6	1	0	0	0.00	0.950
T1*T2*T3*T4	1	0	0	0.87	0.357
Error	32	4	0		
Total	47	175778			

This somewhat counterintuitive result can be explained. Since there is no synchronization between the consumer's Model Thread and the rest of the system, a relative phase, ϕ, results. This mimics the actual startup conditions of an LVC. The random assignment is then, in effect, a sampling of the state space. Capturing this behavior was intentional, as it mimics a real-world distributed simulation where multiple asynchronous simulation applications are using the same state data, each at potentially different points in time.

After eliminating the consumer's Model Thread (T6) as a factor from the experiment, a second three factor, three-level experiment, as shown in Table 4.4, was conducted to provide additional detailed data on each of the remaining factors contribution to aging including any second- and/or third-order interaction effects.

ANOVA results shown in Table 4.5 indicated that factors T1, T2 and T3 explain all of the variance in the LVC system model, and are statistically significant, each having a p-value of zero to three significant digits. Furthermore, each factor's

Table 4.4: 3-Factor, 3-Level Design

Factor	Levels
Producer Model Thread (T1)	50, 80, 100 Hz
Producer Sampling Thread (T2)	5, 10, 20 Hz
Network Latency (T3)	5, 50, 100 ms

Table 4.5: 3-Factor, 3-Level ANOVA

Source	DOF	SS	MS	F	P
T1	2	974	487	170161.47	0.000
T2	2	210038	105019	36696883.44	0.000
T3	2	325093	162547	56798898.05	0.000
T1*T2	4	0	0	0.42	0.796
T1*T3	4	0	0	1.99	0.098
T2*T3	4	0	0	2.02	0.093
T1*T2*T3	8	0	0	1.31	0.238
Error	189	1	0		
Total	215	536105			

contribution to state space aging is independent; there are no significant second- or third-order interaction effects. This validates the second LVC hypothesis; with respect to data aging characteristics, each factor's contribution to aging is independent, and there are no interaction effects.

Furthermore, because the age of data as seen from the perpective of a consumer is the sum of each factors contribution to age, and because each factor is linearly related to either the modeling or sampling thread periods or network latency, the system satifies both properties of additivity and homogeneity. This implies that in terms of the data aging, an LVC is a first-order linear system.

4.3 Analytic Model

Using the results in Section 4.2, an analytic model to estimate the mean worst-case aging and variance of a system design is developed. That is, an analytic model considers all the identified factors affecting data aging, including the initial startup dynamics described in Section 4.1, as well as the network latency characteristics.

Because LVCs are linear systems with respect to data aging, estimates can be made by adding each factor's contribution to age as follows.

A uniform distribution models the contribution to state space aging by Model Thread transition T1. The mean of this distribution is

$$\mu_{Model} = \frac{p_{Model}}{2} \tag{4.1}$$

where p_{Model} is the period of the Model Thread. State space age variance is

$$\sigma^2_{Model} = \frac{p^2_{Model}}{12}. \tag{4.2}$$

In a similar manner, the uniform distribution also characterizes the contribution to state space aging by the Sampling Thread, T2. The mean of this distribution is

$$\mu_{Sampling} = \frac{p_{Sampling}}{2} \tag{4.3}$$

where $p_{Sampling}$ is the period of the Sampling Thread. The contribution to age variance is

$$\sigma^2_{Sampling} = \frac{p^2_{Sampling}}{12}. \tag{4.4}$$

Using (4.1) and (4.3), the contribution of the producer's architecture to mean aging is

$$\mu_{Producer} = \mu_{Model} + \mu_{Sampling}. \tag{4.5}$$

Using (4.2) and (4.4), the contribution of the producer's architecture to variance is

$$\sigma^2_{Producer} = \sigma^2_{Model} + \sigma^2_{Sampling}. \tag{4.6}$$

The mean age of the replicated state space as seen by the consumer is

$$\mu_{SS} = \mu_{Producer} + \mu_{Network} \tag{4.7}$$

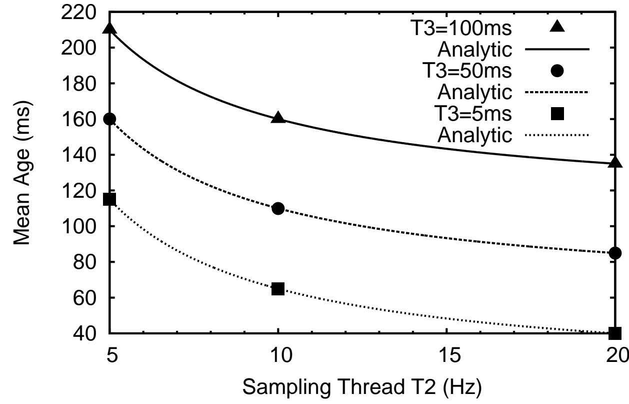


Figure 4.5: Mean Worst-Case Age (ms) (T1=50Hz)

where $\mu_{Network}$ is the mean age due to network latency.

The variance associated with replicated state data aging is

$$\sigma^2_{SS} = \sigma^2_{Producer} + \sigma^2_{Network}. \tag{4.8}$$

Mean network latency, $\mu_{Network}$, and variance, $\sigma^2_{Network}$, can be estimated using empirical data collected by tools such as ping, or the characteristics of a representative distribution. A network modeled as by a two-parameter exponential distribution for example, would have a mean and variance contribution to aging of

$$\mu_{Network} = \gamma + \frac{1}{\lambda} \tag{4.9}$$

and

$$\sigma^2_{Network} = \frac{1}{\lambda^2} \tag{4.10}$$

respectively, where γ and λ are the location and scale parameters of the distribution.

Figures 4.5 and 4.6 compare the analytic model and simulation results for the mean worst-case and standard deviation of aging for several experimental design cases.

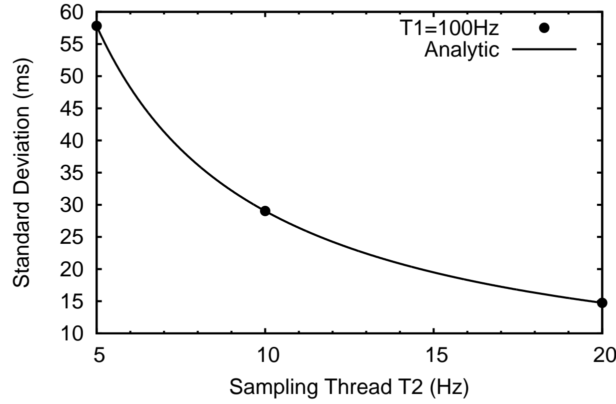


Figure 4.6: Standard Deviation (ms) (T1=100Hz, T3=5ms)

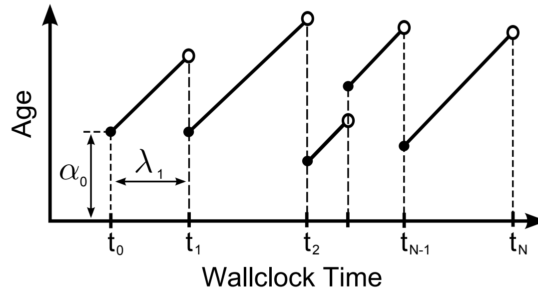


Figure 4.7: Distributed State Space Data

For each design, the analytical model's mean is within ±1ms of the simulation result with 95% confidence.

4.4 Measuring Consistency

These simulation results lead to the development of an effective and efficient algorithm to calculate the mean age and variance as seen by the consumer. Data generated by the producer and placed into the consumer's State Space can be viewed as shown in Figure 4.7.

That is, as a consumer receives data, its value is updated, but not necessarily with the most current value calculated by a producer's Model Thread (T1). The value received will, most likely, have aged due to the asynchronous sampling of the

producer's Sampling Thread (T2) and network latency. Because of this and the characteristics of stochastic non-deterministic networks, arriving data might be younger or in some cases older, than the current value. Whatever the case, after its arrival, the state space ages linearly with wallclock time until the next update. Deriving the mean age and variance of state space data from the perspective of a consumers Model Thread is of most interest.

The mean of a function is the average value of the function over its domain. The mean μ of $f(t)$ over the interval (t_0, t_1) is

$$\mu = \frac{1}{t_1 - t_0} \int_{t_0}^{t_1} f(t)dt \tag{4.11}$$

where $f(t)$ is the aging function associated with state space data and α is its age upon arrival. As shown in Figure 4.7, $f(t)$ increases linearly from α_i to $\alpha_i + \lambda_{i+1}$, where λ is the update interval. In other words, data ages at the same rate as wallclock time advances. Therefore, the aging function is

$$f(t) = t + \alpha_i \tag{4.12}$$

and the mean age can be calculated as a summation over all discrete intervals (i.e., the intervals defined by the interrarrival time between received updates) divided by total elapsed time. Thus, the mean age of the state space is

$$\mu_{SS} = \frac{1}{t_N} \left[\int_{0}^{t_1} f(t)dt + \int_{t_1}^{t_2} f(t)dt + ... + \int_{t_{N-1}}^{t_N} f(t)dt \right] \tag{4.13}$$

which after integration and simplification is

$$\mu_{SS} = \frac{1}{t_N} \sum_{i=1}^{N} \left(\frac{\lambda_i^2}{2} + \alpha_{i-1}\lambda_i \right) \tag{4.14}$$

where λ_i is the interrarrival time of the data, and t_N is total elapsed wallclock time over N intervals.

The variance of state space aging is the mean square error minus the square of the mean. The mean squared error is

$$\text{mse} = \frac{1}{t_1 - t_0} \int_{t_0}^{t_1} [f(t)]^2 \, dt. \tag{4.15}$$

Using (4.12), (4.15) can be integrated and reduced to

$$\text{mse} = \frac{1}{t_N} \sum_{i=1}^{N} \left(\frac{\lambda_i^3}{3} + \alpha_{i-1}\lambda_i^2 + \alpha_{i-1}^2 \lambda_i \right). \tag{4.16}$$

Finally the variance of the state space age is

$$\sigma_{SS}^2 = mse - \mu_{SS}^2. \tag{4.17}$$

The continuous integration of data age as it is received is a practical way to compute data aging characteristics in the Petri net simulation. It is also a useful algorithm to compute, in real-time, the temporal consistency of state data for an LVC.

4.5 Generalized System Model

Because an LVC is a linear system, sources or generators of latency can be classified into general categories such as "computing system," "network," and "middleware." For the producer model, factors T1 and T2 define the characteristics of a simulation applications computing system latency component while factor T3 defines the latency properties of the network modeled by an exponential distribution. This general concept is extended to include other potential sources of latency including middleware software such as HLA, DDS and TENA.

Figure 4.8 shows how computing system and network latency is classified with respect to the Open Systems Interconnection (OSI) Reference Model. For this model, the network infrastructure latency includes time consumed by all media, bridges,

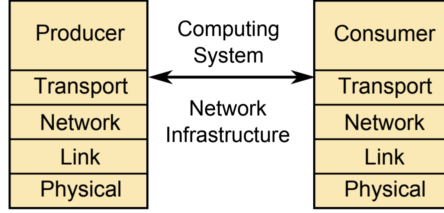


Figure 4.8: Latency Classification & OSI Model

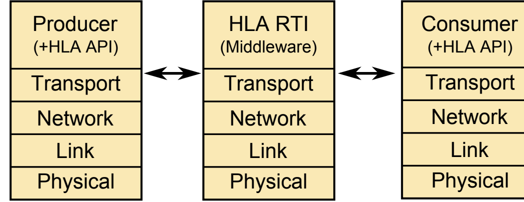


Figure 4.9: HLA-based Communication

routers, gateways, encryption/decryption devices and intervening networks. This is representative of a DIS-based simulation.

Figure 4.9 shows the software architecture of an HLA-based distributed simulation with respect to the OSI model. Because HLA is an interface specification, the various implementations do not necessarily include a separate HLA Run-Time Infrastructure (RTI) application. In this case, HLA API interfaces within the producers and consumers establish and manage all the communication between the nodes. In others, the HLA API communicates with a separate RTI which manages all shared state data within the distributed simulation.

An estimate of data aging for an LVC system design that includes all of these sources is

$$\mu_{SS} = \mu_{Computing} + \mu_{Middleware} + \mu_{Network} \tag{4.18}$$

and the age variance of a system design is

$$\sigma^2_{SS} = \sigma^2_{Computing} + \sigma^2_{Middleware} + \sigma^2_{Network}. \tag{4.19}$$

4.6 Relationship to Validity Interval

In Section 3.8, we defined a system design to be temporally consistent if each shared data object is received by the n nodes (i.e., consumers) in the distributed simulation so that $\forall i$, $1 \leq i \leq n$, $\theta_{TS} + \theta_{i,VI} \geq t$.

To tune the system to meet this requirement, the relationship between a data object's validity interval and a system designs adjustable parameters is used or

$$\theta_{VI} \geq \frac{1}{f_{Model}} + \frac{1}{f_{Sampling}} + \text{ network delay} \tag{4.20}$$

where f_{Model} and $f_{Sampling}$ are the model update and sampling frequencies respectively, and "network delay" is specified given an acceptable reliability statistic (e.g., 95% of the time).

Often, network delay is the least adjustable parameter in the system. If that is the case, an effective validity interval can be written as

$$\theta^{'}_{VI} \equiv \theta_{VI} - \text{ network delay} \geq \frac{1}{f_{Model}} + \frac{1}{f_{Sampling}} \tag{4.21}$$

In order for the system to be temporally consistent, the computing system latency must be less than or equal to the effective validity interval $\theta^{'}_{VI}$. Using Figure 4.10 which shows the influence of Model and Sampling Thread frequencies on computing latency, system parameters can be adjusted to meet this requirement. As long as computing system latency is less than $\theta^{'}_{VI}$, the LVC will be temporally consistent.

4.7 Application

Given managing LVC data consistency adds complexity to a system design, the motivation to interconnect geographically dispersed simulation applications can reasonably be questioned. To this point, we have considered a LVC system to be a set or collection of autonomous simulation applications, each designed to fulfill a specific role in the system design. While this might be ideal, more commonly,

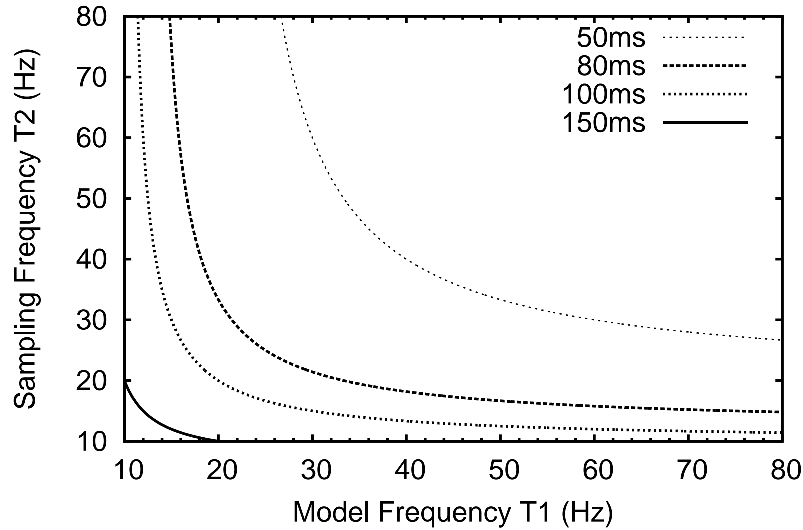


Figure 4.10: Computing System Latency

LVC systems are assembled by interconnecting a collection of *existing* independent simulation applications or by sampling real operational system hardware (live entities) which are often located at different geographic locations.

From an LVC development perspective, this is appealing, as much of the effort to create functional applications is complete. Unfortunately, this strategy limits the system architects ability to partition and tailor the set of dynamic shared states for maximum performance and scalability. An ideal partitioning scheme would involve sharing states with relatively large validity intervals.

Any proposed LVC system design should be considered a “candidate” to evaluate against defined consistency requirements based upon intended use and purpose. If the candidate system does not meet requirements, a new candidate should be derived that satisfies the consistency requirements.

4.8 Aerial Combat Example

As a practical example, consider an LVC system to train fighter pilots for close range aerial combat (i.e., dogfighting). The objective of the training is to learn tactical maneuvers which provide an advantage over an adversary. The LVC design connects two high fidelity motion-based flight simulators located at different geographic locations.

Considering the positional accuracy requirements to conduct this training and the quality of dead reckoning algorithms to estimate aircraft position, the system is said to be temporally consistent (i.e., correct) if shared data is no older than 140ms, 98% of the time. As a point of reference, the DIS standard specifies a transport-to-transport (i.e., network infrastructure) latency value of 100ms with an acceptable reliability of 98% for "tightly" coupled interactions [IEE95].

4.8.1 Candidate System Design. The specifications for the candidate LVC system are

- Two high fidelity motion-based flight simulators connected across a dedicated network with a mean latency of 50ms and a standard deviation of ±5ms.
- The dynamic shared state consists of continuous data objects that specify the position of each fighter.
- The local state space managed by each simulation application is updated at 50Hz which conforms to the producer system model.
- Simulation applications transmit shared state updates at 10Hz.

4.8.2 Evaluation. This candidate system design is evaluated by considering each source or latency generators contribution to aging as presented in Section 4.5.

As shown in Table 4.6, computing system latency is determined considering the startup dynamics (relative phasing relationship) between the modeling and sampling threads for the producer as described in Section 4.1. For this scenario, worst-case

Table 4.6: Computing System Worst-Case Analysis

Factor	Upper Limit	Maximum (ms)
Model Thread	Bounded	20
Sampling Thread	Bounded	100
Total	Bounded	120

aging occurs when the relative phase between the threads is such that the sampling and periodic transmission of updates uses data from the producers state space already aged 20ms. Because the next periodic transmission does not occur for another 100ms, a consumer might receive data at least 120ms old. After including network latency component of 50ms ±5ms, an estimated mean for data aging, μ_{SS}, is 170ms. Thus, this candidate design does not meet requirements.

This is resolved by increasing the periodic rate at which state space updates are transmitted to 20Hz which reduces the worst-case computing system latency to 70ms and results an in overall estimate for μ_{SS} of 120ms to 139ms with 98% reliablity. At this new rate, the LVC is now within bounds of the consistency requirements.

4.9 Summary

This chapter characterized LVCs as a set of asynchronous simulation applications each serving as both producers and consumers of shared state data. Owing to the asynchronous execution and the non-deterministic characteristics of the interconnecting networks, state data used by a consumer is often inconsistent with the most recent value produced. Because of this, the consistency requirements of dynamic shared state data must be described in terms of accuracy and timeliness – each mapping to a validity interval for each node in the system.

In terms of data aging, an LVC system can be viewed as a first order linear system and the rate at which the consumer uses state data is irrelevant to the aging itself. Owing to this, simple analytic models to estimate data aging based upon system architecture are derived. A useful algorithm to compute, in real-time, the temporal consistency of state data for an LVC in operation is provided. Finally, the relationship

between a data object's validity interval and an LVC's system parameters is defined so a temporally consistent system can be designed.

V. Real-Time Design Patterns

In software engineering, a *design pattern* is a general reusable solution to a commonly occurring problem in software design [GHJV95]. It is a description or template for how to solve a problem in many different situations. The *Model-View-Controller* (MVC) and *Component* design patterns are particularly interesting and are adapted herein to the domain of virtual simulation. The MVC pattern provides a high-level architectural structure of an application and classifies objects according to the roles they play. The *Component* pattern is used as a basis to implement those specific objects.

Accepted real-time software organization paradigms are incorporated into these patterns so rate monotonic quantitative methods can be used to estimate the performance of virtual simulation applications. Incorporated paradigms include the separation of software code into foreground and background tasks while the scheduling of individual jobs (i.e., software code) mimics a fixed cyclic scheduler. The patterns also incorporate hierarchical modeling concepts to define modeled systems.

For each pattern, an implementation that leverages modern object-oriented software techniques is assumed. This provides the flexibility to use the concepts of "selective abstraction" and "focused fidelity" to prune object trees, thus improving system performance.

5.1 Real-Time Concepts

This section highlights key concepts associated with software timing constraints during execution including how software systems with temporal requirements are organized. For additional information Liu [Liu00] provides a comprehensive discussion of the theoretical foundations of real-time systems while Laplante [Lap04] focuses more attention on implementation issues.

A real-time system is a system whose specification includes both *logical* and *temporal* correctness requirements.

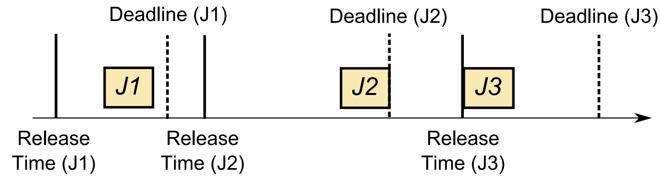


Figure 5.1: Release Time and Deadline Relationships

- Logical correctness produces correct outputs.
- Temporal correctness produces outputs at the right time.

Software systems that respond to external inputs and generate outputs that affect the *real-world* fall into this category. Real-time software systems interact with an external environment such as a person or a piece of hardware. Thus, any timing constraints due to the external environment impose requirements on the software system.

5.1.1 Jobs. Computer software is executed by scheduling code (units of work) on an operating system. These scheduled units of work are called jobs. The temporal characteristics of an individual job are defined by parameters such as release time, absolute and relative deadlines, and response time as follows:

- Release time - the instant the job becomes ready to execute. The job can be scheduled and executed any time at or after its release time if data and control dependency conditions are met.
- Absolute deadline - the instant of time when the job must complete execution.
- Relative deadline - the difference between the absolute deadline and release time.
- Response time - the difference between the time the job completed and release time.

Figure 5.1 graphically shows the execution of several jobs instances (J1, J2 and J3), each with its own release time and deadline. J1 starts after its release time and

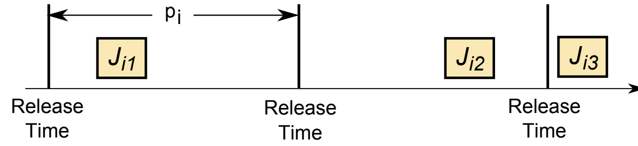


Figure 5.2: A Periodic Task with 3 Jobs

completes before its deadline. J2 completes its execution at the specified deadline, and J3 starts its execution at the point of release.

It is clear from Figure 5.1 that J2 has the worst response time as it completes just before its deadline and J3 has the best response time.

5.1.2 Periodic Task Model. Scheduled units of work are called *jobs*, while a task is defined as a set of related jobs. This association between tasks and jobs is very general as there is no rigid "structure" implied by associating other than that provided by the *periodic task model* [LL73]. In the periodic task model, a task is a sequence of jobs executed at regular intervals. This interval defines when jobs are released by a task.

The periodic task model is a well known deterministic workload model in real-time system theory. In Figure 5.2, the *period* p_i of a periodic task T_i is the time interval between release times of consecutive jobs, J_{i1}, J_{i2}, J_{i3}, in T_i. As Figure 5.2 graphically depicts, a job can execute any time after its release. The execution time, e_i, for the ith task, T_i, is the sum of the maximum execution times of each job in the ith task set.

Typically it is assumed jobs complete their execution by the next release time. In other words, the job deadline D_i is equal to the period p_i for all tasks in the system $T_1, T_2, ..., T_n$. This assumption bounds the response time for a task. If the response time needs to be shortened, a deadline that is shorter than the task period can be specified.

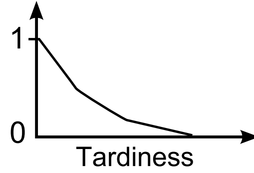


Figure 5.3: Example Usefulness Function

5.1.3 Reliability. The reliability of a system meeting its timing constraints divides real-time systems into two classes, *hard* or *soft*. Hard real-time systems must satisfy explicit (bounded) response-time constraints or risk severe consequences, including failure [Lap04]. Relaxing reliability such that some deadlines can be missed results in a soft real-time system. That is, the system has degraded performance but does not fail if it misses a response-time constraint [Lap04].

5.1.3.1 Usefulness Function. Two approaches capture the quality of a soft real-time system where timing constraints are occasionally violated. The first uses probabilistic values to define how often deadlines will be met. For example, 99% of deadlines will be met.

The second approach defines a "usefulness" function for each job as shown in Figure 5.3. This function characterizes how the system is affected or degraded as a result of completing a job after its deadline. For a hard real-time system, the value of the usefulness function is zero for jobs completing after their deadline.

5.1.4 Utilization. The ratio $u_i = e_i/p_i$ is the *utilization* of a task, where p_i is the task period and e_i is the maximum execution time associated with the task. That is, the utilization is the fraction of time a periodic task keeps a processor busy. The *total utilization* U is

$$U = \sum_{i=1}^{n} {}^{e_i}\!/_{p_i}$$

where n is the number of tasks in the system.

While utilization is a unitless quantity, a common unit of measure in practice is the number of milliseconds (ms) per second a task consumes. For example, consider a task defined by a single job executed at 50 Hz or every 20ms. Assume the maximum execution time for the job is 2 ms. The utilization can be specified as 0.1, 10% of cpu time, or simply as 100 ms/s.

5.1.5 Foreground/Background Systems. An important requirement of the periodic task model is the maximum execution time, e_i, for each periodic task. Without knowing the maximum execution time, guaranteeing the timely completion of tasks is impossible. To meet these bounds, software is partitioned into real-time and non-real-time tasks. Real-time system research literature calls this a *foreground/background* system in which the *foreground* and *background* are:

- Foreground - the set of real-time tasks or processes.
- Background - the set of tasks or processes that are not time critical.

Organization of the code into a *foreground/background* design is a very common software architecture for embedded applications. In fact, all real-time implementations are special cases of that design [Lap04].

5.1.6 Rate Monotonic Analysis. Rate Monotonic Analysis (RMA) is a collection of quantitative methods and algorithms to specify, understand, analyze, and predict the timing behavior of real-time software systems. RMA grew out of the theory of fixed-priority scheduling. A fixed-priority scheduling policy assigns a priority to each periodic task relative to other tasks. The term rate monotonic derives from assigning priorities to tasks based upon a monotonic function of their rates. A system is said to be schedulable if all tasks meet their deadlines. Thus, RMA provides a mathematical and scientific model for reasoning about the schedulability of independent tasks.

A very influential fixed-priority scheduling paradigm is the rate-monotonic (RM) algorithm [LL73]. It is an optimal static priority algorithm for a task model in which

tasks with a shorter period are given a higher priority than tasks with longer periods. The rate-monotonic theorem (described below) is the most important and useful result of this theory [Lap04]. Another important result is the identification of an upper bound on processor utilization such that all foreground tasks will meet their deadlines. In other words, the algorithm identifies both an optimal (rate monotonic) schedule and a bound that places a limit on processor utilization for given real-time tasks. Both theorems can be stated as follows:

Rate Monotonic Theorem [LL73] *Given a set of periodic tasks and a preemptive priority scheduling discipline, then assigning priorities such that the tasks with shorter periods have higher priorities yields an optimal schedule.*

Rate Monotonic Analysis Bound [LL73] *Any set of n periodic tasks is rate monotonic schedulable if the processor utilization, U, is no greater than $n(2^{1/n} - 1)$.*

5.1.7 Threads as Tasks. A thread in computer science is short for a *thread of execution*. Threads are a way for a program to split itself into two or more simultaneous (or pseudo-simultaneous) computational jobs and is the basic unit of work handled by a scheduler. Most commercial operating systems do not support periodic threads, but a periodic task at the user level can be implemented as a thread that alternately executes jobs and is then suspended until the beginning of the next period [Liu00].

To differentiate between real-time foreground threads (implemented as periodic tasks) and non-real-time background threads, background threads are assigned the lowest priority in the system; they can be preempted by any higher priority foreground thread. A system in which a higher-priority task is always able to preempt a lower-priority task is called a *preemptive-priority system*.

Mapping tasks to threads that can be assigned a priority allows the application of rate monotonic theorems as long as the assumptions used to derive the bounds are met. In particular, schedulability and the impact of specific implementations in relation to rate monotonic bounds can be determined.

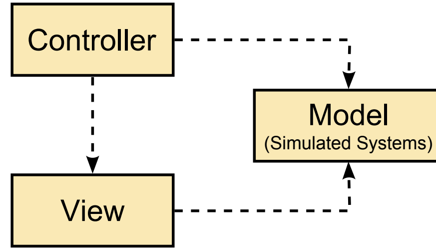


Figure 5.4: Model-View-Controller Pattern

5.2 Model-View-Controller Pattern

In the MVC pattern, there are three types of objects: model objects, view objects, and controller objects. Figure 5.4 shows the roles these objects play in the application and their lines of communication. When designing an application, choosing or creating custom classes for objects that fall into one of these three groups is a major step since it determines object boundaries and communication with other types of objects occurs across those boundaries [Inc07].

For a particular application domain, Model objects represent special knowledge and expertise; they hold an applications data and define the logic that manipulates that data. A well-designed MVC application has all its important data encapsulated in model objects and, ideally, a model object has no explicit connection to the user interface [Inc07]. For a virtual simulation application, the model object is the simulation itself. It contains all simulation state data, behaviors in the form of hierarchical system models, and management of simulation time. The model object as defined in the MVC pattern should not be confused with simulation model. For example, the model object in the MVC pattern is the domain-specific representation of the data on which the application operates (i.e., simulation state data); a simulation model is an abstract representation of a real or imagined system such as an aircraft.

A view object knows how to display or present data to an external viewer. The view is not responsible for storing the data it is displaying and comes in many different varieties. For a virtual simulation, the view includes the drawing of graphical displays such as GUI interfaces and operator displays. In the case of distributed virtual

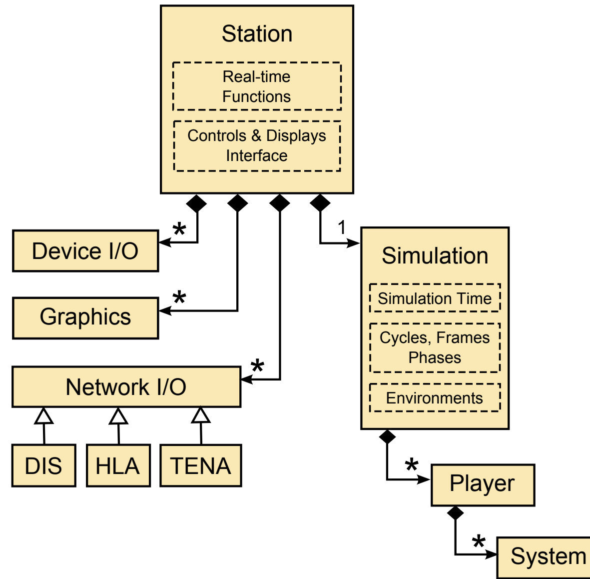


Figure 5.5: Simulation Pattern

simulations (DVS), a view is also responsible for sharing simulation state data across a network.

The controller object acts as the intermediary between the application's view objects and its model objects. Ideally, it is the sole source of user inputs and connects the simulation to its graphical displays. Practically, one often merges the roles played by an object. For example, an object that fulfills the roles of both controller and view would be called a "view-controller" [Inc07].

A view-controller is view layer centric. It "owns" the views, while still managing the interface and communications with the model. Combining roles like this is common [Inc07] and reflects the tailored MVC simulation design pattern shown in Figure 5.5.

The simulation pattern in Figure 5.5 has a top-level "Station" object containing one simulation object (i.e., the model in the MVC pattern) and multiple view-controllers. This top-level object is called a Station to reflect its close association between the management of I/O functions and visual displays with a real physical

operator station. The Station object also manages high-level functions that create threads associated with each of the view-controllers as needed.

The simulation object consists of a list of players organized as a set of hierarchical-based system models consisting of systems with sub-systems. The simulation object manages simulation time and provides attributes needed to implement a fixed cyclic scheduler as described in Section 5.4. View-controllers have handlers that read and/or write to I/O devices, interactive graphical displays and interoperability network interfaces. The network interoperability interface for sharing simulation state data implements a variety of standards such as the Distributed Interactive Simulation (DIS), the High-Level Architecture (HLA), and the Test and Training Enabling Architecture (TENA) specifications.

5.3 Multi-Threading

Ideally, the execution of a simulation application based upon the MVC simulation pattern would consist of a loop that sequentially reads inputs, executes the system models, and generates outputs (i.e., update graphics and process network activities) once per frame. This is a typical execution strategy for constructive simulations where the requirement to execute in real-time is often relaxed since everything is simulated by models. A virtual simulation that performs all of these tasks in real-time, however, is limited by processing power, so this approach becomes problematic as frame rates increase, thereby reducing the amount of time to complete all tasks.

To resolve this fundamental problem, the processing time associated with input devices, graphic display(s) and interoperability network management functions (i.e., the view-controllers) are partitioned into separate periodic tasks, each executed asynchronously with respect to each other, at particular frequencies. For example, the update rate associated with graphical displays might be much less than the rate simulation advances time. Furthermore, the division of software code into foreground and background tasks, as discussed in Section 5.1.5, reduces the workload associated with processing time-critical tasks. In other words, separating time critical code from

code that can be executed in the background decreases the maximum execution time, e_i, of a task.

The challenge is to organize software code to promote this separation of work, thereby enabling the use of RM quantitative methods to estimate performance. At a high-level, RM assumptions translate into software coding rules (or constraints) which are promoted in these design patterns. One such RM assumption is that periodic tasks (i.e., threads) execute independently of each other. Thus, the execution of one thread should never be blocked waiting for data produced by another. This assumption precludes the use of semaphore locks to control access to data available to two or more independent threads.

5.4 Component Pattern

The MVC simulation pattern, as shown in Figure 5.5, is the first step in separating a virtual simulation application into high-level objects that can be executed independently. Further improvement can be made by partitioning the real-time and non-real-time jobs defined by those independent objects (i.e., the simulation and view-controller objects) into foreground and background tasks. The *Component* design pattern facilitates this separation while simultaneously supporting hierarchical modeling concepts.

5.4.1 Hierarchical Modeling. Most systems selected for simulation-based analysis are complex [Rao03]. Because managing the complexity of models is a challenging task, large systems are seldom modeled in a monolithic fashion. In fact, they are usually divided into smaller, interacting subsystems. The subsystems themselves are further divided into smaller sub-subsystems until a manageable level of complexity is reached. In other words, the *system under study* can be viewed as a "system of systems." These divisions results in a hierarchical structure in the *system under study* itself.

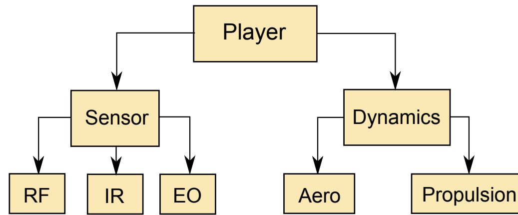


Figure 5.6: Hierarchical Player Model

An example of this hierarchy is shown in Figure 5.6, where the top level model is a "player" or "entity" within the simulation. The player is composed of both a dynamics and a sensor model. The sensor model is a composite of several sensors, namely, radio frequency (RF), infrared (IR), and electro-optical (EO) models. Dynamics is composed of an aerodynamics and propulsion model.

Hierarchical models from a software engineering point of view are software "components." Conceptually, a component is an entity, a real-world object, viewed as a "black box." Its interface, behavior, and functionality are visible but its implementation is not [RHJ+09]. These components naturally map to object-oriented implementation paradigms supported by languages such as C++.

Gamma [GHJV95] contains a catalog of commonly used design patterns in software development and provides solutions developed and evolved over time. *Structural* design patterns provide classes and objects that form larger structures. Of particular interest for hierarchical modeling is the *composite* pattern in Figure 5.7 which implement hierarchical models in object-oriented programming languages.

The composite pattern uses a tree structure where components can have *children*, i.e., subsystems and sub-subsystems. The *Component* class declares the interface for objects in the composition and implements default behaviors for all the classes. The *Leaf* class has no children while the *Composite* class defines behavior for components that have children. The *operation* method is a placeholder for the functionality of the model. Using this structure, modeled systems can be divided into sub-systems and defined as *Components* via inheritance.

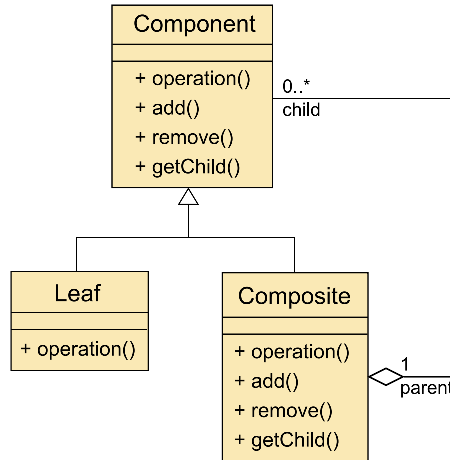


Figure 5.7: Structural Composite Pattern [GHJV95]

When implementing a composite pattern there are trade-offs related to software design safety and transparency. Gamma [GHJV95] provides an extensive discussion that considers several implementation approaches. For example, the component class declares the *add* and *remove* methods to provide a transparent interface for all components, but these do not make sense for a leaf. These trade-offs are considered as this pattern is adapted to the domain of system modeling and real-time processing.

5.4.2 Partitioning Code. The hierarchical-based approach addresses model complexity, but does not address the temporal performance of code execution, specifically, the reliable completion of jobs at or before their deadline. Partitioning code into real-time foreground and non-real-time background tasks as discussed in Section 5.1.5 is recommended.

Given hierarchy models with the structural composite patterns shown in Figure 5.7, software partitioning can be incorporated by replacing the single *operation* method by two methods, *updateTC*, and *updateData* as shown in Figure 5.8. The *updateTC* method (where TC means time critical) is a placeholder to implement a real-time task which includes calculations associated with updating model state space.

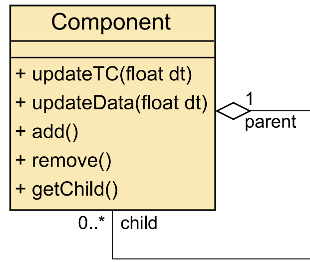


Figure 5.8: Component With Partitioning Support

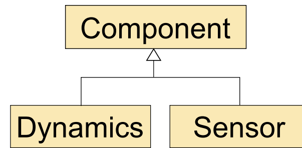


Figure 5.9: Example Component Models

Less time-critical jobs, such as saving or logging data to a hard drive is placed within the updateData method.

We add and explicitly pass the simulation step-size (sometimes referred to as *delta-time*) as a parameter. Step-size is used by mathematical calculations associated with system models. Since *updateTC* automatically calls all of its children's *updateTC* methods, executing a complete hierarchical model (implemented as a component tree) occurs with a single method call to the root component.

Our component design pattern considers all components to be composites. In other words, when modeling systems, sub-system, and sub-sub systems, there are no leaves, as each model is an abstraction at some level.

Consider, for example, the player model in Figure 5.6. To implement this system, several models are created by subclassing from the *Component* class as shown in Figure 5.9. Component models whose functionality is described by a set of differential equations might include a numerical solver in the *updateTC* method. Other background, less time critical jobs, such as saving vehicle position data at each simulation step for analysis, is in the *updateData* method. After each component model is built, the complete flight control system is assembled into a component tree that

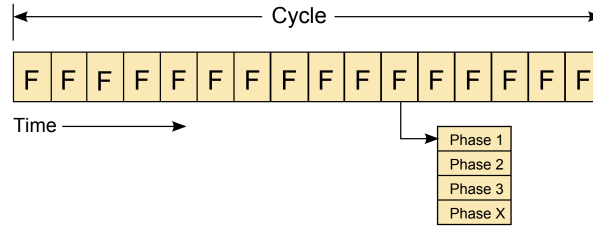


Figure 5.10: Cyclic Scheduler Structure

is the complete modeled system. Subsequent execution or simulation of the modeled system occurs by calling the updateTC method of the root component.

5.4.3 Scheduling Jobs. Designing a software system to meet temporal requirements is a scheduling problem. More formally, to meet a program's temporal requirements in real-time systems, a strategy is needed for ordering the use of system resources [Lap04]. This strategy results in a schedule for executing jobs. Of particular interest is how to schedule jobs to maintain a consistent simulation state space.

To accomplish this, a cyclic scheduler specifies when jobs are executed. The schedule is static, which may not be optimal, but is highly predictable and simple to implement. A cyclic scheduler makes decisions periodically. The time interval between scheduling decision points are called frames. Scheduling decisions are made at the beginning of every frame and there is no preemption within a frame.

A notional structure for a scheduler is shown in Figure 5.10. Frames are grouped into a "cycle," and subdivided into an arbitrary number of *phases*. Frames are divided into phases to resolve data and control dependencies among jobs and specify an execution order.

Adding features to support static scheduling in a *Component* class is as simple as adding attributes, specifically, cycle, frame and phase attributes in the form of class variables as shown in Figure 5.11. Subclasses of *Component* can be built that not only partition model code (i.e., jobs) for execution in the foreground and background, but explicitly define which frames and phases jobs should be processed. Providing

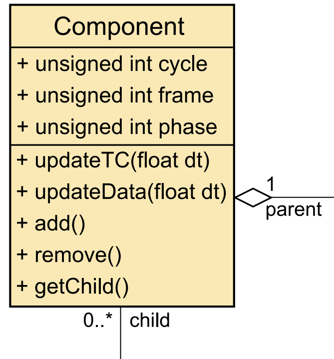


Figure 5.11: Component with Scheduling Support

direct access to scheduling attributes allows the developer to design a model or set of models that balances execution load across frames.

Consider a system model derived from the Component class with the *updateTC* method coded below:

```
updateTC(dt) {
  switch (phase) {
    case 0:
      // system model code A
      break;
    case 1:
      // system model code B
      break;
  }
}
```

The phase attribute is used to impose an execution order within each frame for modeling systems. Conditional code before the switch statement limits processing to selected frames within a cycle. A very common technique conditionally selects a single frame, all even or odd frames, or all frames within a cycle for execution. The parameter "dt" (delta time) is the simulation time advance step-size and is passed to the *updateTC* method and made available for system model calculations. For this design pattern, determining the maximum execution time, e_i, for the task defined by the periodic execution of the *updateTC* method is a matter of computing the total execution time for each individual frame in the cycle, and selecting the maximum.

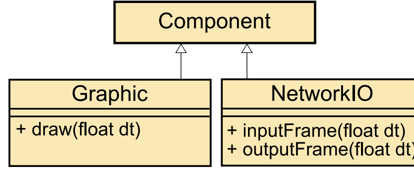


Figure 5.12: Graphic and Network Classes

5.4.4 Modeling a Player. Consider a player or entity defined by an object tree specified by the set of Component instances $\{C_1, C_2, C_3, ..., C_k\}$. The cyclic scheduler for the object tree has p phases, and f frames per cycle. The maximum execution time for the task defined by this single hierarchical system model can be determined by computing the execution time in each frame,

$$e_f = \sum_{comp}^{c} \sum_{phase}^{p} e_{f,c,p}, \tag{5.1}$$

where comp is the set of components, followed by selecting the maximum frame execution time in the cycle,

$$e_{Player} = \max_{1 \leq f \leq cycle} \{e_f\}. \tag{5.2}$$

For a virtual simulation, this is the execution time of a single instance of a player managed by the simulation object shown in Figure 5.5.

5.4.5 Graphics and Input/Output. To support unique features of view-controller objects, specialized *Components* can be created with additional methods. For example, just as the single *operation* method in *Component* was replaced with *updateTC* and *updateData* to partition jobs, additional methods can be added to support the execution of specific jobs unique to a particular view-controller. Effectively, each new method defines an independent execution path through a hierarchical system model or object tree.

As shown in Figure 5.12, specialized *Component* classes to support graphics and interoperability networks are defined. Analogous to the *updateTC* method provided

by *Component*, the *Graphic* class provides a *draw* method for specifying graphic operations. In a similar vein, the *NetworkIO* class provides two methods for receiving and transmitting state data across a network, *inputFrame* and *outputFrame*. The *NetworkIO* class also serves as an abstract interface to support a wide range of interoperability protocols providing a clear separation between models and specific interoperability implementations.

Since the *Graphic* and *NetworkIO* classes are specialized *Components*, they can use *updateTC* for real-time model execution and *updateData* for background processing. For example, *Graphic*-based components can use *updateTC*, graphic operations in draw, and non-real-time background processing in *updateData*. Strictly speaking, this violates the spirit of the MVC pattern as the model would be closely coupled with the view and controller, but is acceptable to meet temporal constraints.

5.5 System Abstraction

Implementing hierarchical, component-based models using the *Component* design pattern efficiently implements "selective abstraction." Selective abstraction [SF98] reduces the complexity of models by identifying and discarding details of the model which have minimal impact on the overall results. This allows the developer to *prune* the object tree at selected points to reduce the level of complexity to improve runtime performance.

Another approach starts with highly abstract system representations and adds fidelity as needed. The term "focused fidelity" is introduced to capture this concept. "Focused fidelity" provides the appropriate level of detail (resolution) to the *system under study* to provide the required accuracy while eliminating undesirable system inputs. This is important because complex models that are not directly under study often affect independent variables which are inputs into the *system under study*, and can therefore confound the study results. Additionally, it is inefficient and often counter-productive to develop more complex models than needed for the simulation.

For example, in Section 3.3 a pilot was inserted into Fujimoto's [Fuj00] simulation of an aircraft flying from New York to Los Angeles to highlight the challenges virtual simulations must address. While the pilot adds fidelity, inserting him into the simulation environment might, depending on the focus of the study, introduce a source of extraneous inputs which can confound results. By focusing on the *system under study*, analyzing both the required fidelity of the system models and the required control over the system's inputs determines whether or not the pilot is relevant and needs to be inserted.

The *Component* class provides the means to implement *selective abstraction* and *focused fidelity* concepts. Applying them reduces simulation development time and cost, while simultaneously improving runtime performance and validity of simulation results.

5.6 Estimating Performance

The performance of a simulation design based upon the MVC simulation, as shown in Figure 5.5, and the Component pattern, as shown in Figure 5.11, can now be estimated. Object trees created with *Component*, *Graphic* and *NetworkIO* classes, the root methods *updateTC*, *draw*, *inputFrame*, *outputFrame*, and *updateData* can be viewed as independent execution paths that can be sequentially processed or associated with individual threads.

As mentioned in Section 5.2, virtual simulations are limited by computational processing power. This forces a developer to associate execution paths to independent threads and assign relative priorities. Typically, the thread associated with the simulation object that sequentially processes the players in the player list has the highest priority to ensure state space updates occur in sync with the wallclock. Assigning the highest priority also avoids the possibility of preemption. Assigning a thread to draw graphics is a function of requirements. The same is true of transmitting network data through *outputFrame*. Receiving network data occurs in *inputFrame*, and is usually assigned to a thread that is blocked until data arrives. The thread assigned to pro-

cess the *updateData* method is always assigned the lowest priority as it remains in the background.

To determine the maximum simulation execution time, e_{sim}, the time to sequentially process all of the players in the player list for each frame in a cycle is computed, followed by the selection of the maximum. At first glance it might appear that this maximum can be computed for each player independently, without regard to other players in the player list. Computing the maximum in this fashion, however, does not account for the execution time associated with player interactions. For example, consider the player shown in Figure 5.6. While the execution time of aircraft dynamics is independent of other players in the simulation player list, this is not the case for the RF sensor model. For this reason, the maximum simulation execution time should also be computed considering all the players in the player list and any execution time associated with player interactions.

Determining the maximum execution time for *draw*, e_{draw}, and *outputFrame*, e_{net}, is simpler because each frame is treated the same, or in other words, not grouped into a cycle. Each thread has an associated period, p_{sim}, p_{draw}, and p_{net}, respectively. By definition, the background thread does not have a period, it simply executes as often as possible. Assuming the three foreground threads just mentioned, the utilization for the system is

$$U = \frac{e_{sim}}{p_{sim}} + \frac{e_{draw}}{p_{draw}} + \frac{e_{net}}{p_{net}}. \tag{5.3}$$

If this computed utilization is not greater than the RMA bound, $n(2^{1/n} - 1)$, or 780 ms/s for $n = 3$, then the system is schedulable. If the bound is not satisfied, abstracting system models (cf. Section 5.5) to reduce complexity and improve runtime performance should be considered.

As another example, consider a virtual simulation in which simulation models are updated by a high priority thread, and graphics are updated by a lower priority thread. Assume task utilizations of 200 and 300 ms/s, respectively. Consider all disk

I/O and any other non-real-time tasks to be assigned the lowest priority thread (i.e., background). Calculating an RMA bound for 2 real-time tasks yields a utilization of 828 ms/s and since the total utilization for the system is 500 ms/s, the system is schedulable. Thus, each job will meet its deadline.

It is reasonable to assume the simulation object which contains the system models are updated at a rate equal to, or higher than, graphical views derived from the data itself. In other words, graphical views typically display information either directly or indirectly derived from state space variables. Updating graphical views at a rate faster than the rate in which data changes is inefficient. This same argument can be applied to transmitting state data across a network.

Often in distributed virtual simulations, player or entity state data can be estimated for vehicle position using dead reckoning. In this case, a usefulness function (cf. Section 5.1.3.1) indicates the value of the data if it is tardy. Thus, the quality of the simulation system degrades slowly and background processing of network activity might be sufficient.

5.7 Consistency and Utilization

The association between consistency and utilization is made clear by writing (5.3) as

$$U = f_{sim}e_{sim} + f_{draw}e_{draw} + f_{net}e_{net} \tag{5.4}$$

where f_{sim}, f_{draw}, and f_{net} are the frequencies of the threads being executed by the simulation application. In this form, f_{sim} corresponds to the model frequency f_{Model}, and f_{net} corresponds to the sampling frequency $f_{Sampling}$ of (4.20).

The frequency of the simulation and the drawing of displays is determined considering modeling objectives and local interaction requirements. The frequency of the network thread should be set to distribute shared state data so that LVC and DVS consistency requirements are met. Because of this additional consideration for con-

sistency throughout the distributed simulation, the frequency of the network thread plays an important role in determining overall application utilization.

5.8 Summary

A flight simulator is useless if it does not reflect the performance of a real aircraft, helicopter, or spacecraft with sufficient accuracy. To meet the challenging task of developing simulations that reliably execute in real-time, two real-time design patterns were developed; a tailored version of the model-view-controller architecture pattern along with a companion Component pattern. Together they facilitate the development of hierarchical simulation models, graphical displays, and network I/O that incorporate real-time system paradigms.

The patterns presented have not been developed in isolation. In fact, they have been carefully crafted and used for many years by simulation engineers. They are also heavily used in the open-source OpenEaagles [RHJ+09] simulation framework as discussed in Appendix B. These design patterns promote software designs that consider real-time requirements and allow performance estimates to be calculated using rate monotonic analysis techniques. Furthermore, the association between thread frequencies and its affect on LVC/DVS consistency and application utilization was discussed.

VI. Conclusion

Since the first simulation networks of the early 1980s to the current state-of-the-art in high-performance distributed computing and gaming, a common vision held by researchers, technologists, and practitioners has been to seamlessly network live, virtual and constructive entities into a common environment. Excluding multi-player gaming, the Department of Defense (DoD) is the largest developer of these systems [SZ99] and invests a significant amount of money into them. LVC and DVS systems are used for training, test and evaluation, experimentation and strategy evaluation. An example is the Air Force-Integrated Collaborative Environment (AF-ICE) [BM06].

Unfortunately, these environments are notoriously difficult to design, implement, and test due to their concurrency, real-time and networking characteristics [YZD00]. System designs are complicated by the conflicting requirement to simultaneously connect geographically distributed simulation applications, while each executes and responds to operator and/or hardware inputs in real-time. This conflict can only be resolved by relaxing the consistency of shared data. Requirements associated with these system have, in the past, been principally driven by operator interaction requirements at the expense of LVC/DVS consistency.

Because of this, it is important to understand the relationship between consistency and interaction quality of these environments which are at odds with each other due to the underlying design and architecture of these distributed simulation systems. It is especially important to understand this relationship from the standpoint of verification, validation and the potential accreditation of these simulations for their intended use.

LVCs are characterized as a set of asynchronous simulation applications each serving as both producers and consumers of shared state data. Because of the asynchronous execution and the non-deterministic characteristics of the interconnecting networks, state data used by a consumer is often inconsistent with the most recent value produced, therefore, consistency requirements of dynamic shared state data must be described in terms of accuracy and timeliness – each mapping to a validity

interval for each node in the system. Temporal consistency theory from the domain of soft real-time database theory is adopted as a basis and framework to describe requirements.

In terms of data aging, an LVC system is a first-order linear system and the rate a consumer uses state data is irrelevant to the aging itself. Because of this, simple analytic models to estimate data aging based upon system architecture can be derived. An algorithm to compute, in real-time, the temporal consistency of state data for an LVC in operation is developed and the relationship between validity intervals and an LVC's systems parameters is defined.

Furthermore, to meet the challenging task of developing simulations that reliably execute in real-time and provide the facility to model hierarchical systems, two real-time design patterns are developed; a tailored version of the model-view-controller architecture pattern along with a companion Component pattern. Together they provide a basis for hierarchical simulation models, graphical displays, and network I/O in a real-time environment. Appendix B provides additional information that shows how these patterns have been leveraged by real simulation applications.

Finally, the relationship between consistency and interactivity was established in Chapter V by mapping threads created by a simulation application to factors that control both interactivity and shared state consistency throughout the distributed environment. These factors are the frequencies or rates at which various system components operate. The utilization of simulation applications can then be computed considering both requirements and compared with rate monotonic principles to determine if a system design is feasible.

In summary, this research defines a fundamental framework to describe LVC/DVS simulation data consistency requirements and provides the means to evaluate system designs that consider both interaction and consistency requirements.

6.1 Future Research

The Technical Cooperation Program [BCE+06] flowchart shown in Figure 7.1 outlines the activities to conduct a valid simulation experiment. The arrows on the side of the chart indicate the added activities of "determining validity intervals" and "data consistency monitoring" in the Experimental Development and Experiment Execution phases respectively. Both of these activities are important for LVC and DVS experiments and are topics for future research.

6.1.1 Determination of Validity Intervals. Accuracy requirements and thus validity intervals flow from modeling and fidelity requirements defined during the Experiment Development phase. For continuous data with a bounded rate of change, error thresholds are the basis for defining such an interval. For discrete data, the timeliness of the data and its impact on models serves the same purpose.

Consider the interaction of a missile launched at an aircraft. Furthermore, consider an LVC designed such that the dynamics of missile position are updated by a simulation different than the one managing the position of the aircraft. Because of the inconsistency in shared data between the simulations, the relative positions between the missile and aircraft in both simulations are in error. This error needs to be quantified so that appropriate validity intervals for both missile and aircraft state data can represent the interaction correctly. If determining whether the detonation of the missile destroys the aircraft is the goal, the validity interval for the transmitted discrete state data representing the detonation must also be carefully specified.

A rigorous methodology for defining validity intervals will no doubt be heavily dependent on specific experiment objectives, but establishing methods to translate or compute the interval from modeling, fidelity and interaction requirements is essential.

6.1.2 Data Consistency Monitoring. The historical approach for detecting problems in a distributed network environment is to monitor the network hardware and computer applications over time and note the occurrence of abnormal events. Ge-

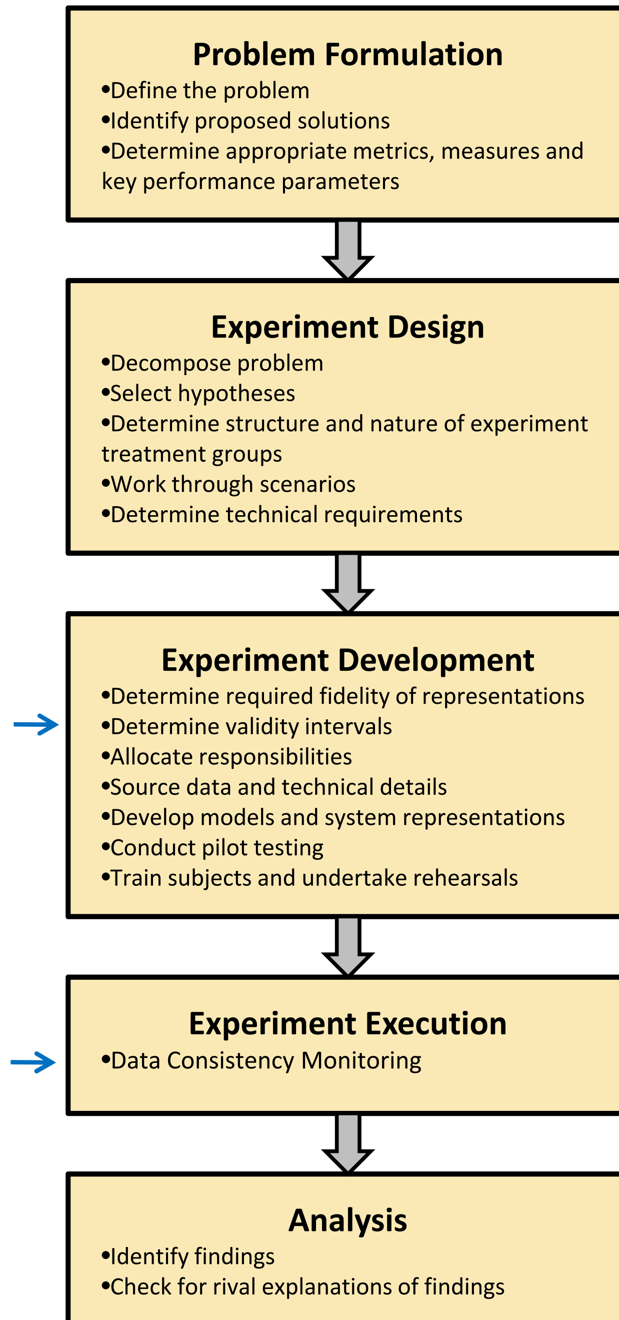


Figure 6.1: Experiment Planning Flowchart (adapted from [BCE+06])

ographically dispersed groups of network engineers and application developers assess whether the network or application is "healthy" enough to support the experiment. Health assessments are usually determined by observing whether the current behavior of the network or applications is within its "normal" pattern of behavior.

Measuring real-time infrastructure performance and assessing its health without adversely impacting the primary mission (connecting and interchanging data between players) of the infrastructure is challenging. The brute force approach of simultaneously measuring all network paths' performance, from each end-host to all other end-hosts in a large distributed network system can add an untenable load on computing nodes and the network infrastructure.

When executing an LVC/DVS, message transmission delays are non-deterministic and unpredictable because of the network protocols and infrastructure used. Therefore, some means of monitoring the quality of the data while an LVC/DVS is in operation is required to ensure the experiment being conducted is valid. Ideally, data monitoring would take place in real-time and non-intrusively so experiments executing outside consistency bounds can be corrected as quickly as possible.

The development of techniques and tools to monitor data validity of an LVC/DVS simulation without introducing additional traffic is important to avoid additional network congestion, thus potentially altering the delay of the state data itself.

For the simulation and performance analysis of DVS and LVC systems, a modeling and simulation tool was developed. The development of the tool leveraged the open-source interactive stochastic timed Petri nets modeling and simulation tool called *STPNPlay* developed by Čapek [Cap01]. To support his research, the *STPNPlay* Petri net simulator was developed to investigate the throughput of non-deterministic Media Access Control (MAC) layers of computer networks.

STPNPlay is suitable for discrete-event token player analysis of stochastic timed Petri net models. It is written in C++, well designed, and compiles on the Windows platform. Because of this, it was deemed a useful starting point for the design of a flexible analysis framework tailored to support this research. As this research progressed, *STPNPlay* was significantly rewritten and reorganized to support the modeling of LVC and DVS systems by extending and incorporating the concept of colored tokens, places with token queues, transitions with embedded algorithms to facilitate color type transformations, and batch processing modes to support scalability estimates. Token color (datatype) transition transformations capture the temporal characteristics of tokens.

As an example of the new features, a DIS-based DVS system model can be modeled with the graphics-based Petri net editor where "colored" tokens represent "data messages" or PDUs, representative traffic generation is specified by specialized transitions associated with a stochastic distribution, and specialized places are used for data collection and statistical calculations. These new features allow for the direct examination of data aging as PDUs are propagated throughout a system model.

A.1 General Features

To facilitate a variety of system models, simulations, and flexible batch execution, the original *STPNPlay* codebase was transformed into a general purpose discrete-event Petri net software framework. The framework was then used as a basis to build custom tools including, for example, exploratory data analysis applications

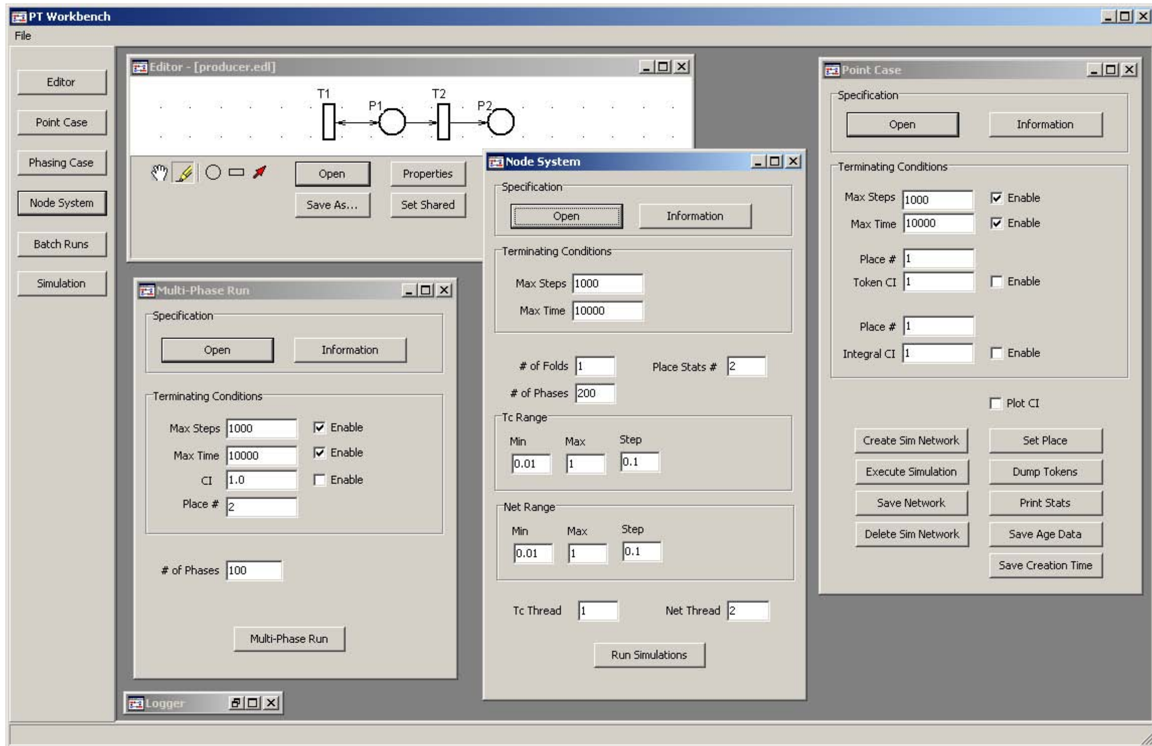


Figure A.1: PT Workbench

useful for investigating LVC and DVS properties. Several of these tools have been grouped together into a general purpose application called *PT Workbench* as shown in Figure A.1.

To create a system model, the Petri net editor as shown in Figure A.2 is used to create and connect places to transitions with arcs. The design can then be saved into a text-based format which can be edited to set specialized token, place and transition object attributes. This is where stochastic temporal transition characteristics are set. The model can then be read into the simulation for execution.

A.2 Software Organization

The software is organized into a set of libraries as shown in Figure A.3. The "basic" library provides a C++ system object and a parser that can read a simple context free language. This language is used to describe the Petri nets. The "gfx" library provides a few classes that draw places, transition and arcs within the graphical

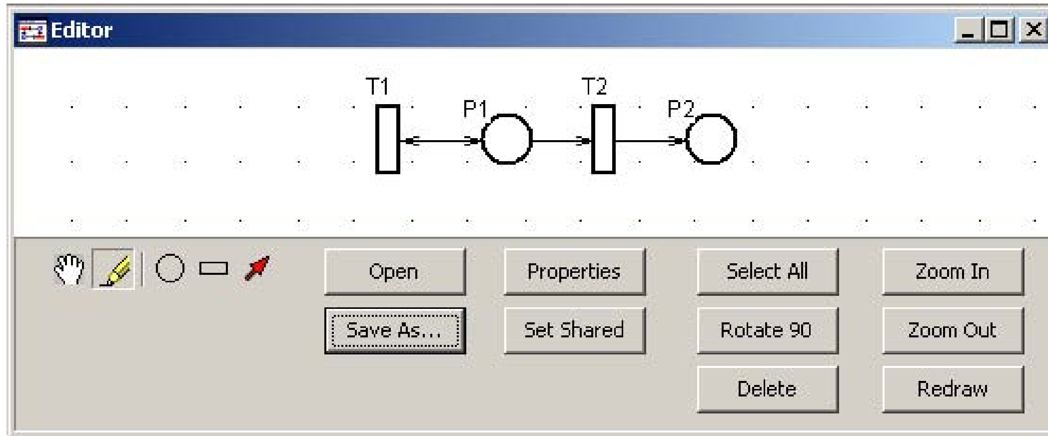


Figure A.2: Petri Net Editor

editor. The "simulation" library defines the discrete-event engine and the Petri net primitives for places, transitions and tokens. These primitives do not implement any "color" features or temporal dynamics; they are often referred to as "black" in the literature. The "model" library contains specific stochastic distributions to model time such as fixed, uniform, lognormal, exponential, and Pareto. It also contains specific transitions to model a variety of components, such as, threads, samplers, sources, logical processes (i.e., LVC and DVS simulation nodes), and networks.

A.3 Execution and Analysis

While *PT Workbench* is useful to for exploratory data activities, simple custom simulation applications were written to support a single or batch set of runs and specific data collection. Data was then imported into the Minitab® statistical package for analysis.

```
<ptworkbench>
  + basic               : object system and parser
  + gfx                 : graphics classes
  + simulation          : discrete-event engine
     - Place
     - Transition
     - Token
     - Rng
  + models              : specific models
     + distribution     : stochastics
       - Fixed
       - Uniform
       - Exponential
       - Parero
       - Lognormal
     + place            : data collection and statistics
       - DataPlace
     + transition       : temporal dynamics
       - Thread
       - Sampler
     + token            : tokens with data ''color''
       - DataToken
```

Figure A.3: Software Organization

Appendix B. Application of Design Patterns

The Simulation and Analysis Facility (SIMAF) located at Wright Patterson AFB (WPAFB), Ohio participates in a number of distributed simulation activities each year that include live, virtual (human-in-the-loop) and constructive entities. The majority of the simulation applications have been developed using the Extensible Architecture for the Analysis and Generation of Linked Simulations (EAAGLES) software package which extends and is based upon the open-source OPENEAAGLES [Ope09a] framework.

The framework supports the development of robust, scalable, virtual, constructive, stand-alone, and distributed simulation applications and implements the design patterns presented in Chapter V. It leverages modern object-oriented software design principles while incorporating fundamental real-time system design techniques to meet human interaction requirements.

By providing abstract representations of system components (that the object-oriented design philosophy promotes), multiple levels of fidelity can be easily intermixed and selected to tune runtime performance. Abstract representations of systems allow a developer to design an application that runs efficiently so that human-in-the-loop interaction latency deadlines can be met. On the flip side, constructive-only simulation applications that do not need to meet time-critical deadlines can use models with even higher levels of fidelity.

The framework embraces the Model-View-Controller (MVC) software design pattern by partitioning functional components into the packages shown in Figure B.1. This concept is taken a step further by providing an abstract network interface so custom protocols can be implemented without affecting system models. Examples include the Distributed Interactive Simulation (DIS) protocol and the High Level Architecture (HLA) interfaces.

Specific applications using the framework to support simulation activities include representative fighter cockpits, an Unmanned Aerial Vehicle (UAV) ground

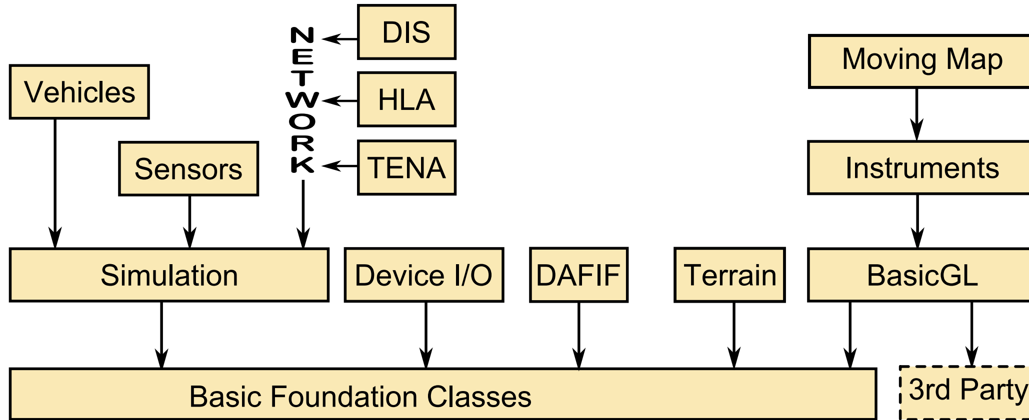


Figure B.1: OpenEaagles Packages

control station (Predator MQ-9), Integrated Air Defense Systems (IADS) and a futuristic battle manager.

B.1 Frameworks, Toolkits and Applications

A framework is a set of cooperating classes that make up a reusable design for a specific class of software [Deu89, JF88]. A framework is customized to a particular application by creating application-specific subclasses of abstract classes from the framework [GHJV95]. A toolkit is a set of related and reusable classes that provides useful, general-purpose functionality. They are the object-oriented equivalent of subroutine libraries [GHJV95].

The OpenEaagles framework itself is not an application. Applications are stand-alone executable software programs like Microsoft Word designed to satisfy a particular need. OpenEaagles is an object-oriented modeling and simulation framework coded in C++. It is partitioned into packages that serve as functional toolkits for a software developer. An example is the graphics toolkit, which facilitates the development of operator/vehicle interfaces and displays.

The framework enables the creation of a diverse set of simulation applications. Derived simulation applications using the framework can be run stand-alone or dis-

tributed. Distributed applications interoperate with other systems and simulations through DIS and/or HLA interfaces. The application might include software agents that represent human participation (constructive) or interact with a real human participant (virtual).

Software execution is partitioned into a foreground/background system, but instead of managing a jump-list (or a list of functions to process), scheduling is interwoven into the object hierarchy as presented in Chapter V. It is specifically designed to take advantage of low-cost multi-core PCs which support the creation of a time-critical foreground thread. Because multiple processors are available, reliable execution of a time-critical thread is assured with general purpose operating systems such as Windows and Linux.

The framework implements a cycle or frame-based systems and is not a discrete-event simulator. This approach satisfies the requirements for which it is designed; namely, support for models of varying levels of fidelity including higher level "physics-based" models, digital signal processing models and the ability to meet real-time performance requirements. Model state can be captured with state machines and state transitions can use the message passing mechanisms provided by the framework.

B.2 An Object-Oriented Real-Time Framework

OpenEaagles is an object-oriented simulation framework implemented in C++. C++ was chosen since:

- Most real-time systems are developed in C for performance reasons [Lap04]. Object-oriented languages tend to be viewed with skepticism as overall system performance often outweighs flexibility. But for the modeling and simulation domain, the advantages afforded by an object-oriented language outweighs this performance penalty.

- C++ is portable and compilers exist on virtually every platform. This allows developers to build applications on any of the major popular operating systems (Windows, Linux, IRIX, Solaris, etc).
- C++ is flexible.
- It is desirable to define memory management so it does not interfere with the overall performance of the application. Therefore, the use of the new/delete operators is preferable to garbage collection.

It is beyond the scope of this chapter to cover each and every class defined in the framework, but a few key classes deserve attention thereby gaining insight into the structure of the framework.

B.2.1 Object. The *Object* class is the C++ system object for the framework. Unlike other object oriented languages (for example Java or Ruby), the C++ language does not provide a system object. C++ also does not provide native garbage collection. The absence of these two features could be viewed as a negative when comparing the native features of various languages, it is a positive when the domain consists of applications that need to meet real-time requirements.

C++ provides the flexibility to define how these mechanisms work for different application domains. For example, if the developer is writing an application in which "control" over potentially time-consuming memory management operations is of little concern, the framework provides smart pointers to automatically manage the creation and deletion of objects. If, on the other hand, the application has time constraints to meet (i.e., a real-time system), the "uncontrolled" creation and destruction of objects will lead to performance problems. One of *Object*s capabilities is to provide a simple reference counting system for the memory management of all framework objects. Thus, a developer can manually control and tune performance-oriented applications when they arise for example in the real-time processing of modeled radio frequency (RF) emission packets or infrared radiation (IR) geometry information.

The other subtle but important aspect of providing a system object is type-checking. The presence of a system object, and the derivation of all classes from it, enables the dynamic casting of objects. It also avoids the pitfalls associated with un-typed functions and classes. The OpenEaagles coding standard explicitly prohibits the use of void pointers for this very reason.

B.2.2 Component. In object-oriented programming, a container class is a class of objects that contain other objects. The OpenEaagles component class implements the *Component* pattern discussed in Chapter V and much more. Not only does *Component* serve as a container for other components, it also includes a basic messaging system used throughout the framework.

From the outset, the framework is designed to facilitate the creation of simulation applications that execute in real-time and/or interact with a human participant. Applications with time constraints and latency/response deadlines typically separate time-critical tasks and non-time-critical tasks; for example, the execution of an aerodynamic model at a specific frequency as opposed to writing data to a hard disk, or printing a document.

This separation is facilitated by two methods in the component class as discussed in Chapter V. When designing a model in the framework, code that needs to execute in a time-critical manner (usually mathematical calculations) is placed in an overridden virtual *updateTC* (update time-critical) method. Code that can be run in a non-time-critical manner is placed in the overridden virtual *updateData* method.

This organization of code has a number of advantages:

- Since time-critical code is clearly separated from background code, applications can be designed to meet performance requirements.
- All the code (time-critical and background) associated with a model is contained within the same class (i.e., usually within the same physical text file).

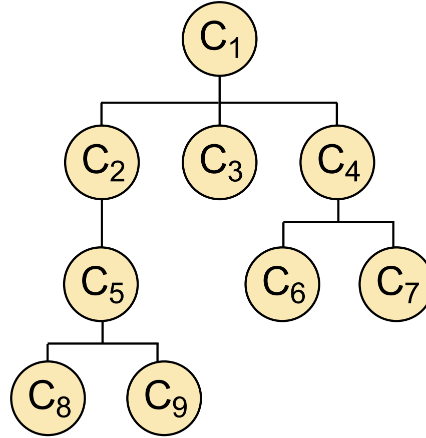


Figure B.2: Component Tree

One can view an instance of a simulation application as a tree of *Component*s as shown in Figure B.2. A call to the top (or root) of the tree's *updateTC* method, automatically executes every subcomponent's *updateTC*. In other words, every component will execute the code of its children. This process continues until the entire tree has been processed. The same process takes place for the background code.

The OpenEaagles coding standard spells out basic rules to follow when writing code in *updateTC* (e.g., no blocking I/O calls). These rules parallel many of the rules used when designing real-time systems.

B.3 Simulation Architecture

A developer using the framework as a basis for a simulation typically builds an application by either using existing classes (or models) or extends them to add new functionality. Then the developer writes the mainline (*main()*) for the application.

The mainline usually has the following structure:

- Read an input file that describes the class/object hierarchy and associated attributes. OpenEaagles provides a parser (written with Flex and Bison) that can read a simple context-free scheme-like input language.

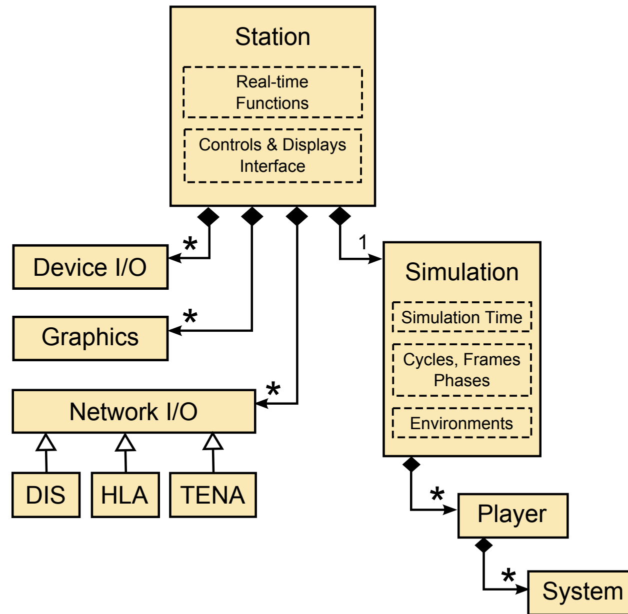


Figure B.3: Simulation Pattern

- Setup the threads as desired. For applications without real-time requirements (e.g., a constructive-only application that processes a series of batch runs) a single thread is all that is needed. For a virtual simulation with time-critical code, a time critical (or high priority) thread should be created.

- Start the simulation by calling *updateTC* and *updateData* as required. If it is a virtual simulation or a simulation where real-time performance is important, the time-critical thread will call the *updateTC* method of the root node.

Full control of the mainline is in the hands of the developer for maximum flexibility. The framework does not even provide a *main()* function! Furthermore, application mainlines tend to be short. Most of the work is in the design and extension of new classes.

Simulation applications are organized like the structure as shown in Figure B.3 which is identical to the simulation pattern presented in Chapter V. Thinking in terms of a tree of components, the class *Station* resides at the top, or the root node. Every other component is a subcomponent of *Station*.

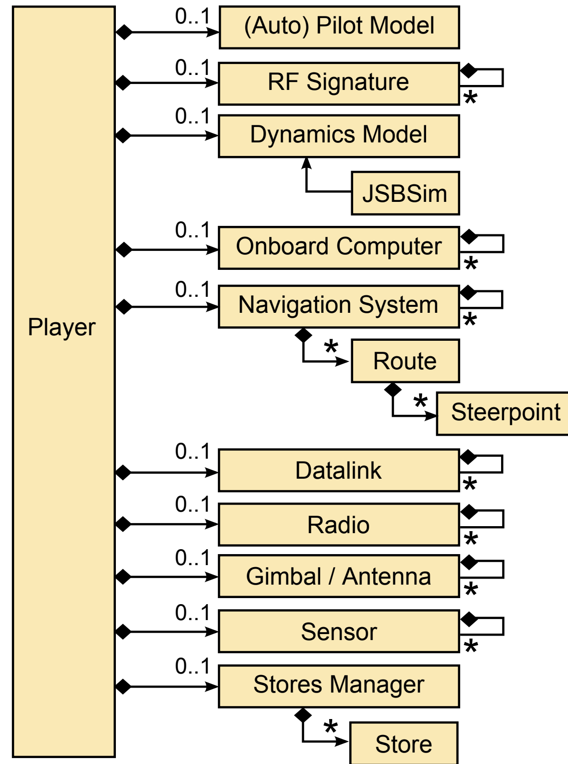


Figure B.4: Player Pattern

The role of *Station* is to connect models to views (or graphical displays) and controls. It owns an instance of the *Simulation* object which manages a list of players (entities), keeps track of simulation time, which includes the cycle, frame and phase that is currently being processed.

Being a frame-based system (not a discrete-event simulator), delta time is passed as an argument to *updateTC* so proper calculations involving time can be performed. Having models rely on delta time for calculation means the frequency of the entire system can change without having to change each and every model (so long as Nyquist rates are met). Additional time related information is recorded in terms of cycles (typically 16 frames) and phases. Phases sequence the flow of data throughout a model. Four phases are currently defined:

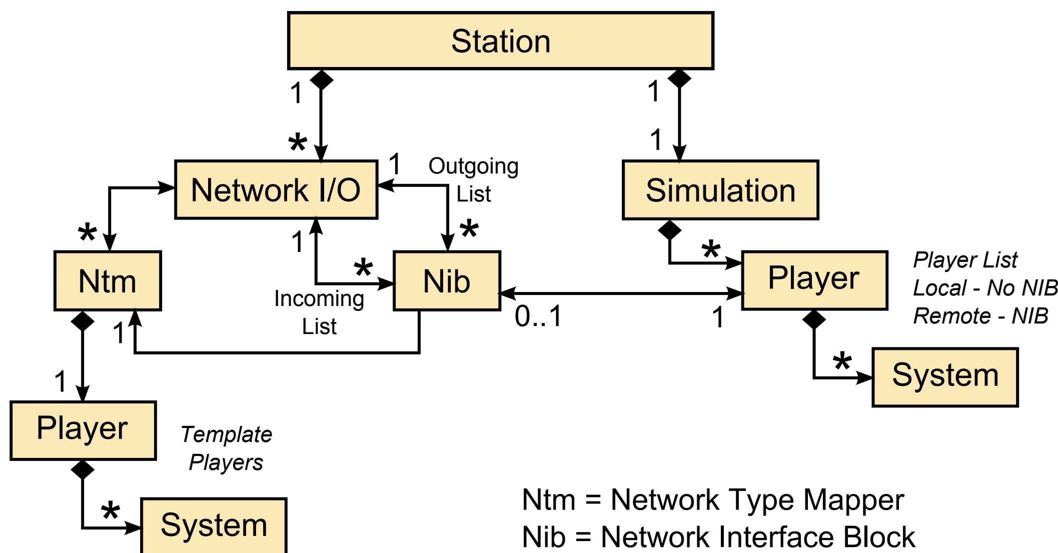


Figure B.5: Interoperability Pattern

- Dynamics – update player or system dynamics including aerodynamic, propulsion, and sensor positions (e.g., antennas, IR seekers).
- Transmit – propagate emission packets, which may contain datalink messages, are sent during this phase. The parameters set in the emission packet include transmitter power, antenna gains and losses.
- Receive – incoming emissions are processed and filtered, and the detection reports or datalink messages are queued for processing.
- Process – used to process datalink messages, sensor detection reports and tracks, and to update state machines, on-board computers, shoot lists, guidance computers, autopilots or any other player or system decision logic.

A *Player* is a subclass of component that adds dynamics and other unique behaviors. Some components that can be "attached" include signatures, antennas, sensors and stores as shown in Figure B.4. Derived air and ground players are included within the framework.

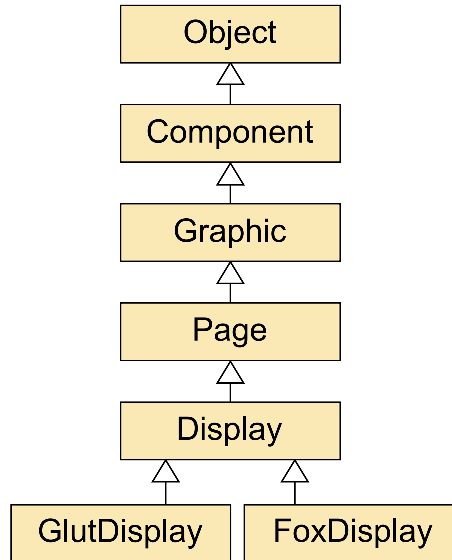


Figure B.6: Graphics Class Hierarchy

An abstract interoperability network interface, as shown in Figure B.5 is defined so specific protocols can be incorporated, such as DIS, for interacting with other distributed simulation applications. This network interface automatically creates new players in the player list. As far as the simulation is concerned, these players are like any other.

B.4 Graphics Architecture

The framework defines several graphic toolkits for the development of operator/vehicle interface displays. The toolkits are based on OpenGL [Ope09b] for all primitive drawing, thus making the framework compatible with virtually any computer platform.

The foundation for graphics drawing is contained in the *basicGL* package. It contains classes for drawing graphic objects such as bitmaps, input/output fields, fonts, polygons, readouts, textures, and others.

The graphics architecture has key fundamental relationships between the *Graphic*, *Page* and *Display* classes (see Figure B.6). The *Graphic* class encapsulates attributes

associated with a graphic such as color, line width, flash rate (for a graphic that flashes), coordinate transformations, vertices and texture coordinates, select names and scissor box information. Since *Graphic* is a component, it can contain other graphics. *Page* is a "page" of graphics that facilitates the creation of Multi-Function Displays (MFD) where specific page transition events need to be defined. The *Display* class defines all the resources available for drawing such as fonts, the color table and both the physical and logical dimensions of the display viewport. Finally, open source GUI toolkits (such as Glut [GLU09], Fox [FOX09], FLTK [FLT09], wxWidgets [wxW09] and Qt [Qt09]) are leveraged by the framework through their respective display classes.

OpenEaagles graphic classes ease the development of operator/vehicle displays and leverage open-source GUI toolkits, but they do not replace visual scenegraph displays (such as heads up displays). The overarching philosophy of the framework is to avoid reinventing the graphics "wheel."

Higher level toolkits that use this structure include the instrument library which includes dials, buttons, gauges, meters, pointers, and countless other fully functional instruments, along with simple maps. The moving map library is another such library.

All of the graphical toolkits are independent of the simulation modeling environment. Models don't have any knowledge of graphics and graphics have no knowledge of models. The code that connects the two resides within the application and is typically associated with the *Station* class.

Through an *ownship* pointer in the *Station* class, the controls and displays of any player can be switched at anytime. Switching from player to player is useful for observing simulation interactions from different perspectives.

All of the graphics classes are derived from *Graphic* which is derived from *Component*. Being a component, all time-critical code can be written into the *updateTC* method and background processing can be written into the *updateData* method. Sometimes, in real-time system development, it is desirable to set graphic drawing to

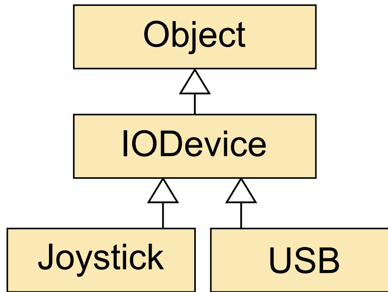


Figure B.7: Device Class Hierarchy

an even lower priority than other background processing. Therefore, another method within the *Graphic* class is defined that serves as a placeholder to do actual OpenGL graphics drawing.

A sample application included in an OpenEaagles distribution illustrates basic graphics by drawing a “worm” that moves around the screen and “bounces” off the walls. Code for this example is organized as follows. All mathematical calculations for the position, speed and direction of the worm are performed in *updateTC*. All the work to setup what to draw is done in *updateData*. The actual drawing of the graphic is performed by *Graphic*’s draw function.

Organizing code this way enables the application developer to determine how to execute the code and to define threads to meet requirements. For this example, a thread is set up to execute time-critical mathematical calculations associated with the worm in “real-time”, and in a non-time-critical manner the operating system (or Glut in this case) draws the worm during idle times.

B.5 Device I/O Architecture

The framework abstracts I/O devices so each hardware interface appears to the application developer as nothing more than a device with a number of analog (axis) and digital (button) values as shown in Figure B.7. This deviceIO package has interface code for several platforms that support joysticks, USB devices, BG System serial boxes and Keithley PCI digital acquisition cards.

Figure B.8: Generic Heads Down Display

Once the device is initialized, a call to the virtual receive method, defined in the *IODevice* class, obtains the latest values from the device. Information about button transitions can also be determined as well as the definition of deadbands for analog inputs.

The *Station* class defines how axes and buttons are connected or "mapped" to the models and views of the simulation application.

B.6 Fighter Cockpit

One of the first EAAGLES-based applications developed at the SIMAF facility was a generic fighter cockpit with a heads down display. The heads-down display was developed using the graphics toolkit as a foundation (see Figure B.8). Window

Figure B.9: MQ-9 Ground Control Station

management is controlled by Glut which is a *Display* that contains other *Graphic* objects and *Display*s as highlighted in the figure. The *Display*s have multiple pages of graphics. This work effectively jump started the creation of the instrument library which continues to mature and expand in scope as well as across application domains.

To the casual observer, the fighter application might appear to be nothing more than a pretty cockpit, but it is actually much more. The application driving the cockpit is an entire simulation ready to be connected into a distributed virtual simulation via DIS or HLA. The cockpit itself is set up through the *Station* class where the heads-down display and controls are associated with one of the players in the simulation player list via *ownship* pointer. In other words, the fighter cockpit is really a simulation entity that is being flown by a human operator. Since the controls and displays are logically separate from the player model, switching and controlling different players during a run can be as simple as moving the *ownship* pointer.

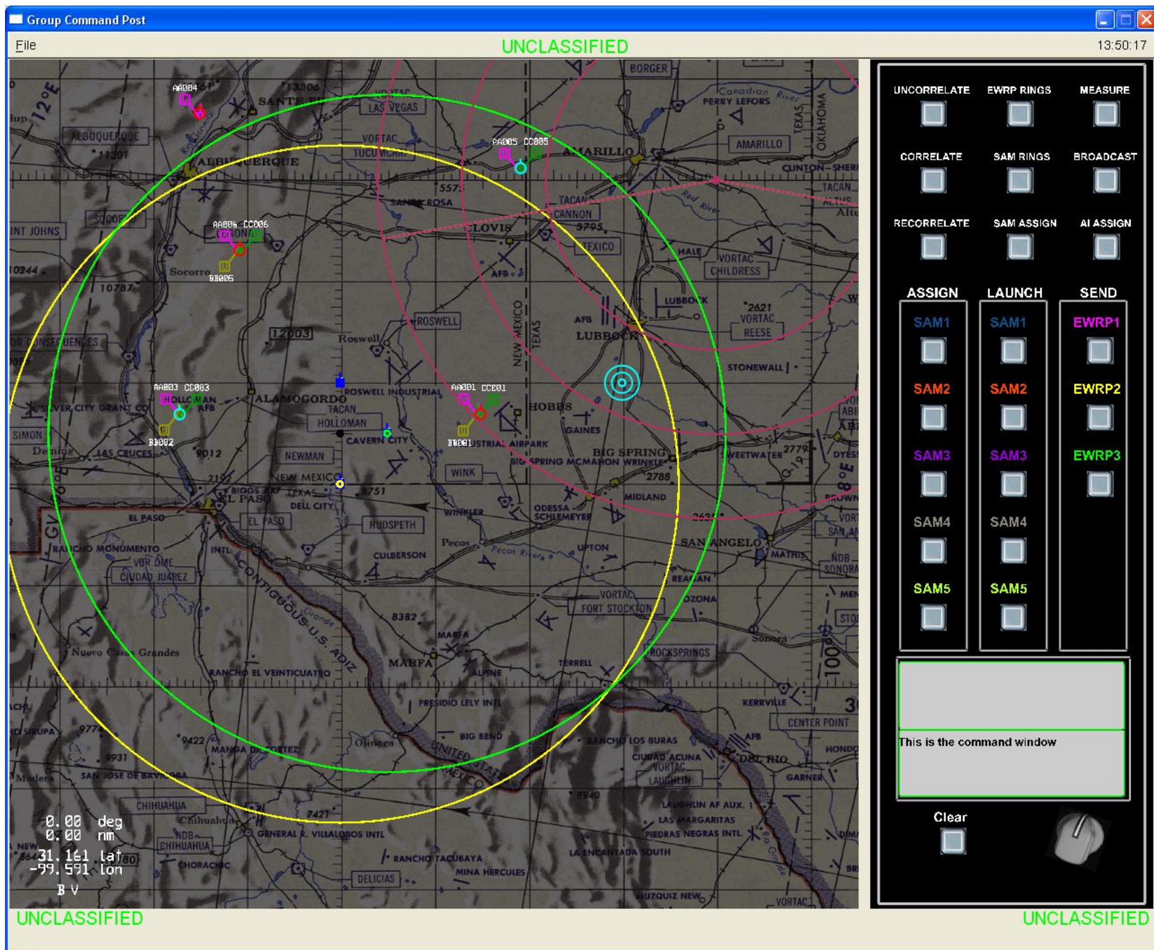


Figure B.10: Group Command Post

This application is used in almost every distributed simulation activity SIMAF participates in or sponsors. It is also used by a number of facilities throughout the different military services.

B.7 MQ-9 Ground Control Station

Compared to the fighter cockpit, the Predator MQ-9 Ground Control Station (GCS) in Figure B.9 appears as a completely different simulation application although it is also built upon the EAAGLES framework. It is a good example of leveraging different frameworks and toolkits to their fullest potential to build an application.

For example, the real GCS controls a Predator with two sets of control sticks. One set controls or flies the Predator directly, and the other controls the sensor ball attached to the UAV. Four displays are presented to the operators: a tracker display in which the operator defines and uploads routes for the Predator to follow; a visual of what the sensor ball is looking at; and two lower displays with multiple pages of textual status information.

The ground control station is simulated with a few EAAGLES-based applications and the Fox GUI toolkit which is a windows based application with menus and dialog boxes used to build the tracker application. OPENEAAGLES-based OpenGL graphics draws the tracker map for planning routes.

SubrScene, an open-source Image Generation System (IGS), generates a visual scene of what the sensor ball is viewing and is controlled by another EAAGLES-based application. All control sticks and inputs use the *DeviceIO* library. This application is routinely used by SIMAF in the Air Forces Virtual Flag event conducted several times each year.

B.8 Group Command Post

The Group Command Post (GCP) is a key component of an overall Integrated Air Defense System (IADS). The GCP receives tracks formed from early warning radar posts and filter centers under its control and develops a sector air picture. It determines which tracks are hostile and assigns the appropriate weapons system to counter the threat directly by assigning the threat to a surface-to-air missile, anti-aircraft artillery, airborne interceptor or indirectly assigning the threat to a weapons post responsible for assigning the appropriate weapon system (see Figure B.10).

This application, along with two other EAAGLES-based applications (Early Warning Radar Post and SAM site), forms the core of the IADS infrastructure. This infrastructure is used in a number of distributed simulation events including Airborne Electronic Attack (AEA) which examines the impacts of various electronic warfare

techniques upon both an enemy's integrated air defense system and blue force capabilities.

B.9 Summary

The EAAGLES software package and the open-source OPENEAAGLES framework upon which it is based provide a mature infrastructure to build simulation applications designed to work in LVC simulations. At the lowest level, the framework implements the design patterns presented in Chapter V. There are other "patterns" used throughout the framework for RF and IR modeling, but the essential partitioning of software code (i.e., jobs) is accomplished with the *Component* class.

The framework is routinely compiled with Microsoft Visual Studio for the Windows environment and GCC for Linux. Applications perform best when executed on multi-core CPU systems because of the priority based threading in these systems. Windows and Linux are both designed for general purpose processing, not real-time processing, thus, one CPU can be dedicated to the operating system kernel which reduces the possibility of interfering with a time-critical task.

EAAGLES is government-owned and not proprietary. It is managed by the SIMAF facility located at WPAFB, OH.

Bibliography

BCE+06. Dean Bowley, Paul Comeau, Roland Edwards, Paul J. Hiniker, Geoff Howes, Richard A. Kass, Paul Labbé, Chris Morris, Rick Nunes-Vaz, Jon Vaughan, Sophie Villeneuve, Mike Wahl, Kendall Wheaton, and Mike Wilmer. *Guide for Understanding and Implementing Defense Experimentation (GUIDEx) - Version 1.1.* The Technical Cooperation Program (TTCP), 2006.

BG92. Dimitri P. Bertsekas and Robert Gallager. *Data Networks - Second Edition.* Prentice Hall, New Jersey, 1992.

BM06. Col Eileen Bjorkman and Timothy Menke. Air Force-Integrated Collaborative Environment (AF-ICE) development philosophy. *International Test and Evaluation Association (ITEA)*, March/April, 2006.

Cap01. Josef Capek. *Petri Net Simulation of Non-deterministic MAC Layers of Computer Communication Networks.* PhD thesis, Czech Technical University, Department of Control Engineering, 2001.

CK06. François E. Cellier and Ernesto Kofman. *Continuous System Simulation.* Springer, New York, NY, 2006.

Com06. Douglas E. Comer. *Internetworking with TCP/IP - Principles, Protocols, and Architecture – Vol 1, Fifth Edition.* Prentice Hall, 2006.

Deu89. L. Peter Deutsch. Design Reuse and Frameworks in the Smalltalk-80 System. In *Software Reusability Volume II: Applications and Experience*, pages 57–71. Addison Wesley, 1989.

DG99. Christophe Diot and Laurent Gautier. A distributed architecture for multiplayer interactive applications on the Internet. *IEEE Network*, July/August 1999.

DoD95. DoD. *Department of Defense Modeling and Simulation Master Plan, 5000.59-P.* Defense Modeling and Simulaton Office (DMSO), October 1995.

DoD97. DoD. *Department of Defense Modeling and Simulation Glossary, 5000.59-M.* Defense Modeling and Simulaton Office (DMSO), December 1997.

DoD02. DoD. *Foundation Initiative 2010: The Test and Training Enabling Architecture – Architecture Reference Document*, 2002. `https://www.tena-sda.org`.

FLT09. FLTK. A cross-platform C++ graphical user interface toolkit, July 2009. `http://www.fltk.org`.

FOX09. FOX. A cross-platform C++ graphical user interface toolkit, July 2009. `http://www.fox-toolkit.org`.

Fuj00. Richard M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley-Interscience, New York, NY, 2000.

GHJV95. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Design*. Addison-Wesley, Upper Saddle River, NJ, 1995.

GHS95. Richard Gerber, Seongsoo Hong, and Manas Saksena. Guaranteeing real-time requirements with resource-based calibration of periodic processes. *IEEE Transactions on Software Engineering*, 21(7):579–592, July 1995.

GL00. Sumit Ghosh and Tony S. Lee. *Modeling and Asychronous Distributed Simulation – Analyzing Complex Systems*. IEEE Press, 2000.

GLU09. GLUT. The OpenGL utility toolkit, a window system independent toolkit for writing OpenGL programs, July 2009. `http://www.opengl.org`.

Hl04. Felix George Hamza-lup. *Dynamic Shared State Maintenance in Distributed Virtual Environments*. PhD thesis, University of Central Florida, 2004.

IEE95. IEEE. *Standard for Distributed Interactive Simulation – Communication Services and Profiles, Standard 1278.2*, 1995.

IEE98. IEEE. *Standard for Distributed Interactive Simulation, Standard 1278*, 1998.

IEE00. IEEE. *Standard for Modeling and Simulation High Level Architecture, Standard 1516*, 2000.

Inc07. Apple Incorporated. Cocoa Fundamentals Guide. *The Model-View-Controller Design Pattern*, 2007.

Jai91. Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley and Sons, Inc, 1991.

Jen97a. Kurt Jensen. A brief introduction to coloured petri nets. In *TACAS '97: Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 203–208, London, UK, 1997. Springer-Verlag.

Jen97b. Kurt Jensen. *Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*. Springer-Verlag, 1997.

Jen97c. Kurt Jensen. *Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods*. Springer-Verlag, 1997.

Jen97d. Kurt Jensen. *Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use. Volume 3, Practical Use*. Springer-Verlag, 1997.

JF88. Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

KLA+03. Ben Kao, Kam-Yiu Lam, Brad Adelberg, Reynold Cheng, and Tony Lee. Maintaining temporal consistency of discrete objects in soft real-time database systems. *IEEE Transactions on Computers*, 52(3):373–389, 2003.

Kol03. Boris Koldehofe. *Collaborative Environments: Aspects in Communication and Educational Visualisation*. PhD thesis, Chalmers University of Technology and Göteborg University, Sweden, March 2003.

KS97. C.M. Krishna and K.G. Shin. *Real-Time Databases*. McGraw-Hill, 1997.

KSG99. J. Kato, A. Shimizu, and S. Goto. Active measurement and analysis of delay time in the Internet. *IEEE Proceedings of the 1999 International Workshops on Parallel Processing*, pages 254–259, 1999.

Lap04. Phillip A. Laplante. *Real-Time Systems Design - Third Edition*. IEEE Press/Wiley Interscience, Piscataway, NJ, 2004.

Liu00. Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, Upper Saddle River, NJ, 2000.

LK00. Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis, Third Edition*. McGraw Hill, New York, NY, 2000.

LL73. C. L. Lui and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973.

MKBK98. Rajesh Mascarenhas, Dinkar Karumuri, Ugo Buy, and Robert Kenyon. Modeling and analysis of a virtual reality system with time petri nets. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 33–42, Washington, DC, USA, 1998. IEEE Computer Society.

Mos93. David Mosberger. Memory consistency models - tech report that is an update of paper published in ACM SIGOPS. *Operating Systems Review*, 27, 1993.

MT95. Duncan C. Miller and Jack A. Thorpe. SIMNET : The advent of simulator networking. *Proceedings of the IEEE*, 83(8), August 1995.

Mur96. Tadao Murata. Temporal uncertainty and fuzzy-timing high-level petri nets. In *Proceedings of the 17th International Conference on Application and Theory of Petri Nets*, pages 11–28, London, UK, 1996. Springer-Verlag.

Ney97. David L. Neyland. *Virtual Combat - A Guide to Distributed Interactive Simulation*. Stackpole Books, 1997.

NSP+97. S. Narayanan, Nicole L. Schneider, Chetan Patel, Todd M. Carrico, John DiPasquale, and Nagesh Reddy. An object-based architecture for developing interactive simulations using Java. *Simulation*, 69(3):153–171, 1997.

OMG09. OMG. The Data Distribution Service (DDS) for real-time systems is a specification of a publish/subscribe middleware for distributed systems, June 2009. `http://www.omg.org`.

OMN09. OMNet++. An extensible, modular, component-based C++ simulation library and framework for building network simulations, June 2009. `http://www.omnetpp.org`.

Ope09a. OpenEaagles. The Open Extensible Architecture for the Analysis and Generation of LinkEd Simulations framework, June 2009. `http://www.openeaagles.org`.

Ope09b. OpenGL. The Open Graphics Library is a standard specification defining a cross-language, cross-platform api for writing applications that produce 2d and 3d computer graphics, July 2009. `http://www.opengl.org`.

Pet62. Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, Fed. Rep of Germany, 1962.

Pet77. James L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, 1977.

Qin02. Xiao Qin. Delayed consistency model for distributed interaction systems with real-time continuous media. *Journal of Software*, 13(6):1029–1039, 2002.

Qt09. Qt. A cross-platform C++ graphical user interface toolkit, July 2009. `http://www.qtsoftware.org`.

Ram93. Krithi Ramamritham. Real-time databases. *ACM International Journal of Distributed and Parallel Database*, 1(2):199–226, 1993.

Ram07. Krithi Ramamritham. Taming the dynamics of distributed data. In *ACM Proceedings of the Eighteenth Conference on Australasian Database*, 2007.

Rao03. Dhananjai M. Rao. *A Study of Dynamic Component Substitutions*. Ph.D. dissertation, University of Cincinnati, 2003.

RHJ+09. Dhananjai M. Rao, Douglas D. Hodson, Martin Stieger Jr, Carissa B. Johnson, Phani Kidambi, and Sundaram Narayanan. *Design & Implementation of Virtual and Constructive Simulations Using OpenEaagles*. Linus Publications, 2009.

RJL+97. Maria Roussos, Andrew E. Johnson, Jason Leigh, Craig R. Barnes, Christina A. Vasilakis, and Thomas G. Moher. The NICE project: Narrative, immersive, constructionist/collaborative environments for learning in virtual reality. In *ED-MEDIA/ED-TELECOM*, 1997.

SF98. Alex F Sisti and Steven D. Farr. Model abstraction techniques: An intuitive overview. *Aerospace and Electronics Conference, NAECON, IEEE National*, 1998.

SL92. Xiaohui (Carol) Song and Jane W.E. Liu. How well can data temporal consistency be maintained? *IEEE Symposium on Computer-Aided Control System Design (CACSD)*, pages 275–284, 1992.

SL95. Xiaohui (Carol) Song and Jane W. S. Liu. Maintaining temporal consistency: Pessimistic vs. optimistic concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):786–796, 1995.

SS93. Lui Sha and Shirish S. Sathaye. Distributed real-time system design: Theoretical concepts and applications. Technical report, Software Engineering Institute - Carnegie Mellon University, 1993.

SS95. Lui Sha and Shirish S. Sathaye. Distributed system design using generalized rate monotonic theory. Technical report, Carneigie Mellon University, 1995.

SZ99. S. Singhal and M. Zyda. *Networked Virtual Environments – Design and Implementation*. Addison Wesley, 1999.

Tan95. Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

Uni07. University of Aarhus, Denmark. *Coloured Petri Nets 1 and 2 - Modeling and Validation of Concurrent Systems Course Notes, University of Aarhus, Denmark*, January 2007.

VDGG04. Juan J. Vargas, Ronald F. DeMara, Avelino J. Gonzalez, and Michael Georgiopoulos. Bandwidth analysis of a simulated computer network running OTB. *Swedish American Workshop on M&S Conference Proceedings*, 2004.

WCPW05. John W. Woodring, John B. Comiskey, Orlin M. Petrov, and Brian L. Woodring. Creating executable architectures using visual simulation objects (VSO). *Proceeding of the Society for Optical Engineering (SPIE)*, 5805:165–176, 2005.

Wel02. Lisa Wells. *Performance Analysis Using Coloured Petri Nets*. PhD thesis, University of Aarhus, 2002.

Wes01. Douglas B. West. *Introduction to Graph Theory – Second Edition*. Prentice Hall, 2001.

Wik07. Wikipedia. Picture copyright 1997, Center for Innovative Computer Applications – referenced by Wikipedia, February 2007. http://inkido.indiana.edu/a100/handouts/Image116.gif.

Wik09. Wikipedia. CAVE Automatic Virtual Environment, June 2009. http://en.wikipedia.org/wiki/Cave_Automatic_Virtual_Environment.

wxW09. wxWidgets. A cross-platform C++ graphical user interface toolkit, July 2009. http://www.wxwidgets.org.

XLLG06. Ming Xiong, Biyu Liang, Kam-Yiu Lam, and Yang Guo. Quality of service guarantee for temporal consistency of real-time transactions. *IEEE Transactions on Knowledge and Data Engineering*, 18(8):1097–1110, 2006.

Yan05. Yue Yang. *Formalizing Shared Memory Consistency Models for Program Analysis*. PhD thesis, The University of Utah, 2005.

YZD00. Tadao Murata Yi Zhou and Thomas A. DeFanti. Modeling and performance analysis using extended fuzzy-timing petri nets for networked virtual environments. *IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics*, 30(5):737–756, 2000.

ZCLT01. Suiping Zhou, Wentong Cai, Francis B.S. Lee, and Stephen J. Turner. Consistency in distributed interactive simulation 01E-SIW-003. *European Simulation Interoperability Workshop*, 2001.

ZCLT04. Suiping Zhou, Wentong Cai, Bu-Sung Lee, and Stephen J. Turner. Time-space consistency in large scale distributed virtual environments. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 14(1):31–47, 2004.

ZMD99. Y. Zhou, T. Murata, and T. DeFanti. Modeling and analysis of collaborative virtual environments by extended fuzzy-timing petri nets. *The Institute of Electronics, Information and Communication Engineers (IEICE) Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 1999.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 074-0188*

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to an penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From – To)*
12-2009	Doctoral Dissertation	June 2009-October 2009

4. TITLE AND SUBTITLE

Performance Analysis of Live-Virtual-Constructive and Distributed Virtual Simulations: Defining Requirements in Terms of Temporal Consistency

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Hodson, Douglas D., YF-02, ASC/XRA - SIMAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way, WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/DCE/ENG/09-25

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Intentionally left blank

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved For Public Release; Distributed is Unlimited

13. SUPPLEMENTARY NOTES

This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States

14. ABSTRACT

This research extends the knowledge of live-virtual-constructive (LVC) and distributed virtual simulations (DVS) through a detailed analysis and characterization of their underlying computing architecture. LVCs are characterized as a set of asynchronous simulation applications each serving as both producers and consumers of shared state data. In terms of data aging characteristics, LVCs are found to be first-order linear systems. System performance is quantified via two opposing factors; the consistency of the distributed state space, and the response time or interaction quality of the autonomous simulation applications. A framework is developed that defines temporal data consistency requirements such that the objectives of the simulation are satisfied. Additionally, to develop simulations that reliably execute in real-time and accurately model hierarchical systems, two real-time design patterns are developed: a tailored version of the model-view-controller architecture pattern along with a companion Component pattern. Together they provide a basis for hierarchical simulation models, graphical displays, and network I/O in a real-time environment. For both LVCs and DVSs the relationship between consistency and interactivity is established by mapping threads created by a simulation application to factors that control both interactivity and shared state consistency throughout a distributed environment.

15. SUBJECT TERMS

simulation, distributed, real-time, temporal, consistency

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
REPORT	ABSTRACT	c. THIS PAGE			Rusty O. Baldwin, (ENG)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	134	937-255-6565 x4445 (email: rusty.baldwin@afit.edu)

Standard Form 298 (Rev: 8-98)
Prescribed by ANSI Std. Z39-18