

Distributed Virtual Environment for Radar Testing

A Major Qualifying Project

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Matthew Lyon

James Montgomery

Lucas Scotta

Date: October 14, 2010

MIT Lincoln Laboratory Supervisor: D. Seth Hunter

Approved:

Professor George T. Heineman, Major Advisor

Professor Hugh C. Lauer, Co-Advisor

Professor Edward A. Clancy, Co-Advisor

This work was sponsored by the Department of the Air Force under Contract No. FA8721-05-C-0002. Opinions, interpretations, recommendations and conclusions are those of the authors and are not necessarily endorsed by the United States Government.

Abstract

This paper describes the design and prototype implementation of a distributed radar simulator for MIT Lincoln Laboratory. This simulator is designed to test radar control software developed at the Laboratory by mimicking radar hardware and simulating radar returns from targets in a virtual environment. The team worked closely with Lincoln Laboratory staff to ensure that the simulator design would be extensible to support different types of radar systems and scalable to thousands of targets. Finally, a distributed simulator was implemented in order to validate the project design.

Executive Summary

Radar is one of the core technologies developed at MIT Lincoln Laboratory in support of national security. Central to a radar sensor system is the processing software used to interpret radar returns and configure the physical hardware. Radar simulation is a common technique used by the Laboratory to test this software before the actual hardware is deployed. A simulator exists within the Laboratory and has been used over the past 10 years for testing radar software.

While the existing radar simulator is capable of modeling complex hardware and environmental effects, it is limited in the number of targets (such as planes or missiles) it can model due to the fact that it can only run on one machine. Additionally, there is a desire for the current simulator to model additional scenarios and be extensible to simulate new types of radar hardware. This project addresses these issues by creating a new simulator, designed to use a distributed architecture and scale to support an arbitrarily large number of targets. The first section of the project was the design of the simulator, while the second major section was implementing a prototype simulator to validate this design.

Throughout the project, members of the group worked in close collaboration with staff from Lincoln Laboratory to leverage expertise in technical areas such as radar and general simulation. The design was developed through weekly iterations in which the project team met with advisors and Lincoln staff to review the current design and ensure the proper direction of the project. The group also made use of existing software created at the Laboratory such as a simulator architecture called Open Architecture Simulation Interface Specification (OASIS) and a program for inter-process communication called the ROSA Thin Communications Layer (RTCL). These resources expedited the progress of the project through the use of proven software used in other projects at the Laboratory.

Over the course of the project, the team developed a high-level design where the radar hardware was simulated separately from the targets and environment. The design intended for one

hardware simulator to be used in conjunction with several target simulators that have a set of targets distributed across them. Since the majority of the computation in simulating radar comes from determining the radar returns produced by targets, this would divide the computational load across multiple processes, allowing it to be distributed to multiple machines.

In order to make the new simulator architecture as flexible as possible, the project team used a design similar to the OASIS architecture developed at the Laboratory. This architecture suggests a separation between the domain-specific parts of a simulation (such as all radar-related calculations) and the communication that occurs between different parts of the simulator. For this reason, the new simulator was divided into a collection of *models*, which represent domain-specific objects (such as radar transmitters or receivers) and *engines*, which coordinate event-driven communication between the different models being simulated. Using this design, the team was able to create a generic engine for event-driven simulation in addition to a set of accurate radar domain models made through iterative design with the Laboratory staff.

The team also developed a prototype simulator implementation in parallel with this design. The implementation served to validate the design and prove the capability of the new simulator to be distributed across several processes. It also ensured the correct operation of the model objects and the accuracy of all calculations used to determine radar returns. The implementation required the integration of the model and engine classes, as well as the use of time synchronization between separate hardware and target simulators.

As a result of the project, the team presented several key deliverables. The first of these was a rich simulator design, which passed a formal design review held with staff at Lincoln Laboratory. The second main deliverable was the prototype simulator implementation, which demonstrated the design's ability to be distributed across multiple processors. Finally, an extensive testing suite and set of

documentation were created to ease the adoption of the new simulator framework by the Laboratory. Lincoln Laboratory has expressed substantial interest in continuing the development of a new distributed radar simulator using the design and research conducted through this project.

Acknowledgements

The completion of this project would not have been possible without the contributions of numerous individuals who dedicated their time and talent in order to make this project a success. We would foremost like to thank MIT Lincoln Laboratory for presenting us with the opportunity to perform the project at the Laboratory and make use of the tremendous resources available there. We would like to thank our supervisor at the Laboratory, Seth Hunter, who was responsible for organizing this project and regularly oversaw the progress made by the team, as well as our group leaders.

We would especially like to thank the members of the Ranges and Test Beds Group who aided us in the project by providing design feedback and familiarizing us with domain-specific knowledge. Specifically, Marcia Powell, Gregory Gimler, Matt Leahy, Andrew Clough, and David Carpman were all valuable resources throughout the course of the project. We would also like to thank all Lincoln Laboratory staff who attended our design reviews and provided valuable feedback used in driving the simulator development.

Finally, we would like to thank our advisors George Heineman, Hugh Lauer, and Edward Clancy who consistently kept us motivated and professional throughout the project. We appreciate the amount of dedication they put forth in attending weekly meetings, providing design feedback, and helping to organize the final written report.

Table of Contents

Abstract.....	2
Executive Summary.....	3
Acknowledgements.....	6
Table of Contents.....	7
Table of Figures.....	10
1 Introduction.....	11
1.1 MIT Lincoln Laboratory.....	11
1.2 Radar Open Systems Architecture (ROSA).....	12
1.3 The Simulator.....	13
1.4 Project Description.....	15
2 Background Research.....	17
2.1 Radar.....	17
2.2 ROSA II.....	22
2.3 RTCL.....	24
2.4 The Simulator.....	25
2.5 Simulation.....	27
2.5.1 Analytic Simulations and Virtual Environments.....	27
2.5.2 Sequential Discrete-Event Simulation.....	28
2.5.3 Open Architecture Simulation Interface Specification (OASIS).....	29
2.6 Distributed Systems.....	31
2.7 Real-time Computing.....	32
3 Methodology.....	34
3.1 Work Environment and Tools.....	34
3.1.1 Language Choice.....	34
3.1.2 Integrated Development Environment (IDE).....	35
3.1.3 RTCL.....	37
3.1.4 Boost.....	37
3.1.5 Version Control and Collaboration.....	38
3.1.6 Test Cases.....	39
3.1.7 Code Coverage.....	40

3.1.8	Documentation	41
3.2	Software Engineering Practices	43
3.2.1	Iterative Design and Development	43
3.2.2	Sponsor Collaboration.....	44
3.3	Procedural Timeline	44
3.4	Division of Labor	48
4	Design and Implementation.....	50
4.1	Functional Requirements.....	50
4.2	Design Overview	50
4.3	Layered Architecture.....	51
4.4	Model Layer	52
4.4.1	Modeling the Radar Range Equation	53
4.4.2	Models and Events.....	54
4.4.3	The Model Class Hierarchy.....	56
4.5	Simulation Engine Layer.....	57
4.5.1	Configuring and using the simulation engine	58
4.5.2	Event scheduling and subscription	59
4.5.3	Global event broadcasting and time synchronization	60
4.6	Component Layer.....	62
4.7	Middleware Layer	64
4.8	Scalability of the Design	64
5	Results and Analysis.....	68
5.1	Design Review	68
5.2	Implementation Results.....	70
5.3	Testing Results	74
5.4	Documentation	76
5.5	Summary of Results	78
6	Conclusion.....	80
6.1	Outstanding issues.....	80
6.1.1	ROSA Interface	80
6.1.2	Multi-threading.....	82
6.1.3	Timing and Synchronization.....	83

6.1.4	DDS Configuration	84
6.2	Future work.....	84
6.2.1	Load balancing	84
6.2.2	Configuration Objects	86
6.2.3	Status Messages.....	86
6.2.4	Simulator components as ROSA II components	87
6.2.5	Graphical User Interface	87
6.3	Concluding Thoughts	89
	Appendix A. Current ROSA Simulator Feature Tree.....	90
	Appendix B. Minimal Implementation Requirements As Specified By Sponsor	92
	Appendix C. Design Review.....	93
	Appendix D. Simulation Engine Class Diagram	94
	Appendix E. Target Model Class Diagram	95
	Appendix F. Hardware Model Class Diagram.....	96
	Appendix G. Target Simulator Sequence Diagram.....	97
	Appendix H. Hardware Simulator Sequence Diagram	98
	Appendix I. Target Simulator Parallelism Sequence Diagram.....	99
	Appendix J. Simulation Engine Global Event Broadcast Sequence Diagram	100
	Appendix K. System Deployment Diagram	101
	Appendix L. Glossary	102
	References	103

Table of Figures

Figure 1 – An example ROSA radar deployed by Lincoln Laboratory.	13
Figure 2 – Various types of radar and their respective functions.....	17
Figure 3 – The Electromagnetic spectrum	18
Figure 4 – High-level overview of a Radar Hardware system	19
Figure 5 – Azimuth and Elevation of a radar antenna	20
Figure 6 – The radar range equation [O'Donnell, 2002].	21
Figure 7 – Block Diagram of a ROSA II System	23
Figure 8 – RTCL Publish/Subscribe	25
Figure 9 - Radar simulation	26
Figure 10 - Simulation Engine and Application	29
Figure 11 – The OASIS Layers [MIT Lincoln Laboratory, 2009].	30
Figure 12 – Amdahl's Law	32
Figure 13 – The Eclipse Integrated Development Environment	36
Figure 14 – The BullseyeCoverage Browser	41
Figure 15 – A sample page of doxygen, which documents information for a specific class.....	42
Figure 16 – Initial Macro-level System Design	51
Figure 17 – The adapted OASIS layers used in the design of the new simulator	52
Figure 18 - Each step in calculating the radar range equation [O'Donnell, 2002].....	53
Figure 19 - The radar range equation organized into models and events.....	54
Figure 20 - The inheritance tree for the model classes	56
Figure 21 - The Simulation Engine	58
Figure 22 - Event Broadcast Sequence Diagram	60
Figure 23 - Component State Machine	62
Figure 24 - Component and its Subclasses	63
Figure 25 - Scalability of the DVERT simulator.....	65
Figure 26 - Parallel event processing in the simulation engine	66
Figure 27- The common acceptance test used in testing the implementation	71
Figure 28 - The energy returns from two targets.	72
Figure 29 – The energy returns from the acceptance test after several seconds of elapsed time.	73
Figure 30 - Project code coverage numbers provided by Bullseye Coverage.....	74
Figure 31 - A sample doxygen class diagram	77
Figure 32- A sample page from the User's guide, walking through the installation of the simulator.....	78
Figure 33 - Distributing target computation by sector (A), target range (B), or volume (C)	85
Figure 34 – A Basic simulator Control GUI	88

1 Introduction

Over the past century, radar has evolved from pure theory to an important technology with many applications including national defense, air traffic control, and weather sensing. At the turn of the 20th century, scientist Nikola Tesla theorized that radio waves could be used to detect objects and their trajectories by transmitting short pulses and listening for energy reflections [Secor, 1917]. Thirty years after his prediction, Great Britain was deploying Radio Detection and Ranging (RADAR) systems to detect incoming aircraft during World War II. Following the war, radar was adapted for civilian usage. Radar revolutionized air traffic control, allowing civilian airports to coordinate the safe approach and departure of aircraft. Weather sensing radars, perhaps the most well-known application of radar, were first developed in the mid-1950s. These radars allow meteorologists to detect impending storms before they arrive, providing advance notice of inclement weather. While civilian radar development continued, the onset of the Cold War spurred a renewed interest by the defense community. The detection of intercontinental ballistic missiles and long-range bombers became a paramount priority as the nuclear threat grew. Radar remains a critical technology in support of national security, and research continues today to improve radar detection capabilities and apply radar technology to new areas.

1.1 MIT Lincoln Laboratory

MIT Lincoln Laboratory, the sponsor for this project, has an important place in the history of radar development [Ward, 2000]. The Laboratory was founded in 1951 with a primary focus on air defense, with radar being a large part of these defense efforts. Since then, the Laboratory has made large strides in developing radar technology and has fielded radars in sites all over the world. Now one of the leading radar authorities in the world, the Laboratory has also branched out to research areas such as optics, communication, and weather sensing.

MIT Lincoln Laboratory is one of nine Federally Funded Research and Development Centers (FFRDCs). The Laboratory is located at Hanscom Air Force Base in Lexington, MA and managed by the Massachusetts Institute of Technology. The Laboratory's mission statement is "Technology in Support of National Security." [MIT LL, 2010a] Research at the laboratory focuses on the rapid prototyping of new technologies and the transfer of knowledge to industry. The Laboratory is made up of seven technical divisions, each with a specific mission area. Within each division is a series of groups that focus on specific areas of research. This project takes place within the Ranges and Test Beds Group of the Air and Missile Defense Technology Division. This group develops modern sensor systems to support ballistic missile defense and is investigating a new architecture for next-generation radar sensor systems.

1.2 Real-Time Open Systems Architecture (ROSA)

"Radar systems are traditionally developed from the ground up, using proprietary hardware and software architectures. This traditional development model is expensive and requires long design times. Further, because each radar system employs unique architectures and technology, it is difficult and expensive to maintain and upgrade the vast assortment of fielded systems" [Rejto, 2000]. The Laboratory has led a recent initiative to move the design and development of radar systems away from proprietary hardware and software and towards an openly defined radar standard.

MIT Lincoln Laboratory envisioned an open architecture for radar systems to reduce the cost and complexity of new radars. ROSA (Real-Time Open Systems Architecture) project was an initiative to design radar hardware around a standard framework, focusing on the use of commercially available hardware to replace proprietary pieces of radar hardware. ROSA was followed by a second initiative to standardize the software used in radar systems, called ROSA II. ROSA II established a standard software framework for describing the functional modules present in modern radar systems. The creation of the ROSA standard was a turning point in radar development for both the Laboratory and the defense

industry [Rejto, 2000]. Radars were able to be reliably deployed within weeks, whereas previous deployments usually took months or even years.



Figure 1 – An example ROSA radar deployed by Lincoln Laboratory.

A number of ROSA radars have already been developed by the Laboratory. ROSA radars are fielded at both the Reagan Test Site, located on the Kwajalein Atoll in the Marshall Islands, as well as at the Haystack Observatory, located in Westford, MA.

1.3 The Legacy Simulator

The complex radar systems developed at MIT Lincoln Laboratory need to be tested before they can be deployed in the field. Assembling and calibrating a complete radar requires a significant investment of time and resources. Additionally, the Laboratory has helped set up installations across the globe, including the Pacific Missile Range Facility in Hawaii and the Reagan Test Site at the Kwajalein Atoll in the Marshall Islands. Because of the vast distances between the Laboratory and the radars it maintains, it's simply impractical to perform large scale tests on-site. As such, Lincoln Laboratory

engineers developed a software simulation package known as the ROSA simulator to emulate radar hardware systems as well as a mock environment containing imaginary targets.

The legacy simulator provides a testing environment for radar control and signal processing software. Radar control software sends control messages which directs the radar hardware. The simulator accepts the same control messages and emulates the real radar hardware and returns the same signal data that would be captured by a real receiver. The simulator models targets such as missiles and planes that the radar signal processing software should be able to detect. Using the simulator saves the Laboratory and its sponsors time and money by allowing radars to be tested and problems to be fixed before the hardware is deployed. From both an engineering and a financial standpoint, simulation is an excellent way to ensure the integrity of a new radar system.

Performance limitations due to the simulator's design and architecture limit the number of targets it can model to approximately 80 on a powerful machine. Lincoln developers have expressed a desire to "model thousands of targets" in a single simulation. A higher-fidelity model of a single plane might use hundreds of targets moving together to represent the different parts that make up the plane. Calculating the energy returned from each target when it is hit by a pulse is a computationally expensive task that the simulator needs to finish before the time the radar processing software expects a return. The simulator must operate in real-time, meaning it must produce results within a deterministic and consistent period of time. The simulator was designed to execute on a single machine, and its architecture is not capable of scaling to support hundreds of thousands of targets.

1.4 Project Description

This project addressed the limitations of the legacy simulator and provides MIT Lincoln Laboratory with a foundation from which they can modernize their radar simulator infrastructure. Breaking away from the monolithic, single-machine nature of the legacy simulator, a Distributed Virtual Environment for Radar Testing (DVERT) was developed. The primary goal of the project was the design of this architecture, in addition to the implementation of a simulator prototype to validate this design.

The deliverables presented with this project are defined below:

- A simulator architecture capable of supporting thousands of targets and varied radar hardware models
- The results of a formal design review with Laboratory staff to ensure the architecture fully satisfies the needs of future projects within the Laboratory
- A concrete implementation of the architecture in order to validate its extensibility and scalability
- Integration of the prototype into an existing radar processing chain in place of the simulator for performance benchmarking
- A comprehensive test suite that provides at least 80% code coverage
- Extensive documentation detailing the use and further development of the simulation

It should be stressed that the implementation was not intended to immediately replace the simulator. As this project lasted for less than 2% of the development time of the simulator, it would have been unrealistic to re-implement all of the advanced features of the legacy simulator. Instead, the implementation aimed to validate the design and to demonstrate that a distributed radar simulation would be possible.

This paper provides a synopsis of the project as a whole. It starts with a description of the various background technologies involved in the project. From there it moves on to the various tools and techniques employed during the project. Following the methodology, a technical overview of the design and implementation provides a description of the functional components and how they interact with the DVERT architecture. Finally, the paper ends with a summary of the project's accomplishments and a detailed list of future work that could be completed.

2 Background Research

This section covers the domain-specific knowledge that was required in order to gain an appropriate background for undertaking the project. It includes sections on radar hardware and software, as well as details on simulation and distributed systems, which were central concepts relating to the project. Other sections include background information on the previous radar simulator used within the Laboratory, as well as details on the OASIS simulator architecture and the ROSA Thin Communications Layer, which were both developed at the Laboratory.

2.1 Radar

Radar operates on the principal that when an electromagnetic wave hits an object, some of the energy in the wave is reflected across the environment to a receiving antenna. This energy can be detected and measured, allowing a radar system to mathematically infer the position of a target at a given time. When a radar system transmits a pulse of electromagnetic energy across the environment, some amount of time passes before the pulse hits a target. Distant targets will have a longer response time because the wave must travel farther, while closer targets produce energy returns much more quickly. This difference in travel time allows radar systems to determine the distance to a target.

Type of Radar	Radar Explanation
Dish Radar	A radar that uses the same antenna for transmitting and receiving. Can rotate on a pedestal to scan the environment.
Multi-static Radar	Uses multiple transmitters and/or receivers at different locations. Useful for obtaining information about targets from multiple angles.
Phased Array Radar	Uses a large number of transmitters that broadcast their waves out-of-phase with each other. Modifies the phase of the transmitters in order to steer the direction of the beam.

Figure 2 – Various types of radar and their respective functions.

A variety of different types of radar exist for different purposes (see Figure 2). There are detection radars, while there are also imaging radars that use high-frequency radar energy to map the surface of an object. Radars also operate across a wide spectrum of frequencies, as different frequencies lend themselves to different applications. For example, lower frequencies are typically used for long-range applications as it's easier to increase the power of the signal, which in turn leads to an increase in the radar's range. Higher frequencies are used for imaging, as the increased resolution leads to a more detailed return from a target. This frequency, or number of cycles per second in the radar energy wave, characterizes how well it penetrates objects such as clouds, and how it is affected by atmospheric effects. Figure 3 characterizes electromagnetic waves according to their frequencies. The lowest radar frequencies fall near the radio category of the spectrum at 10^6 Hz; however, some radars utilize frequencies in excess of 10^{11} Hz.

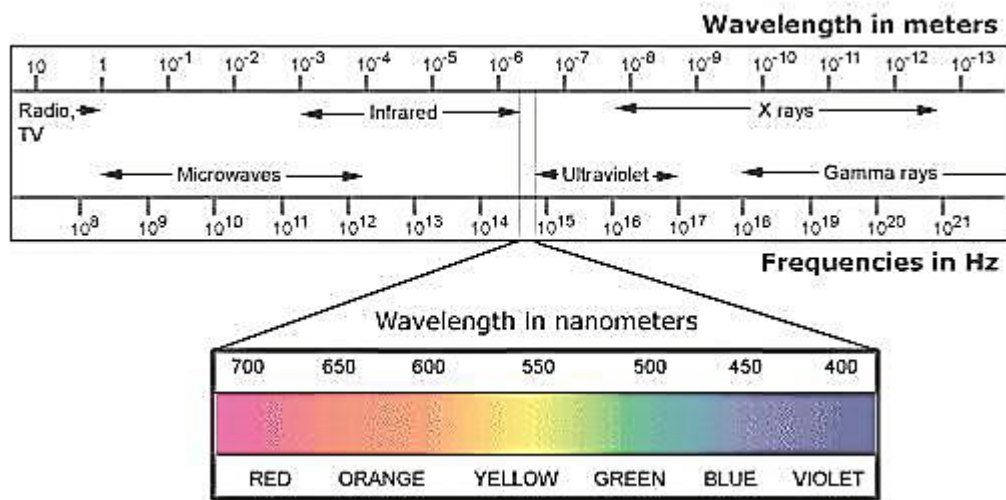


Figure 3 – The Electromagnetic spectrum, by frequency and wavelength [Thomas Publishing Company, 2010].

Radars can also transmit energy at differing power levels; some high-power signals are able to project pulses across very long distances (e.g., thousands of miles) or even into space [Sangiolo, 2001].

Most radar systems are composed of several key hardware components. Figure 4 outlines a basic overview of the high-level hardware components used in most radar systems:

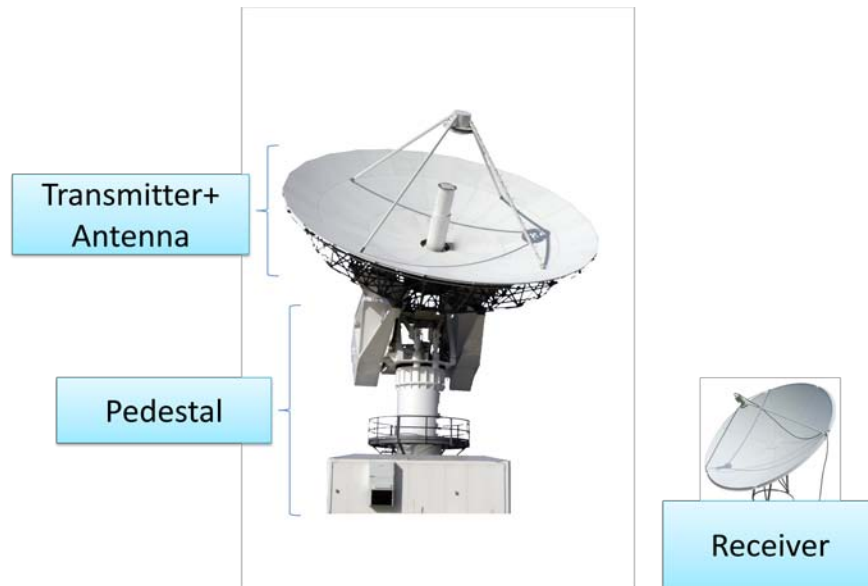


Figure 4 – High-level overview of a Radar Hardware system

A radar is composed of a transmitter and a receiver. The transmitter projects waves of electromagnetic energy across the environment and is composed of the power system, which controls the frequency and energy of the waves, as well as the pedestal, which determines the direction of transmission. Most transmitters are capable of adjusting the power and frequency of the energy pulses they transmit based on the expected type of target. A radar can be composed of one or more transmitters; multiple transmitters can be used to provide different or higher-resolution information about the environment.

While some radar transmitters are stationary, others are mounted to some kind of platform that provides more refined control over antenna movement. Pedestals are equipped with motors that can rotate an antenna and allow it to see the entire surrounding environment. The rotational position of the antenna about the vertical axis is referred to as *azimuth*. Most pedestals are also equipped to rotate the

antenna about the horizontal axis, which adjusts the *elevation* of the antenna. Figure 5 shows the azimuth and elevation of a simple radar dish.

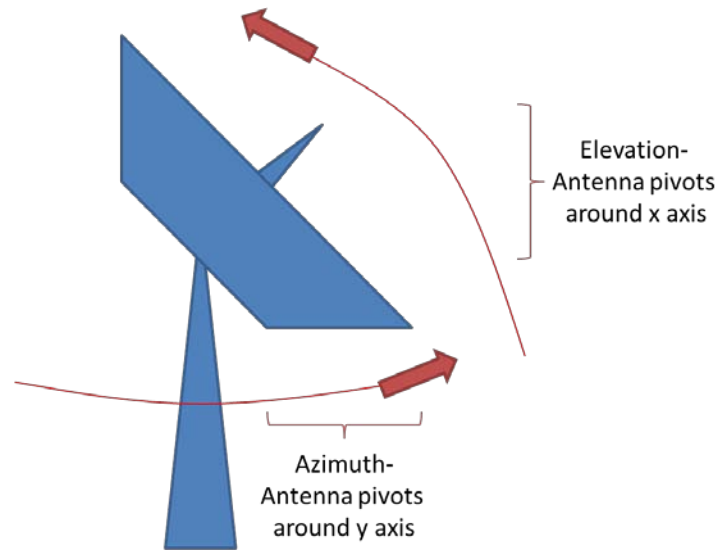


Figure 5 – Azimuth and Elevation of a radar antenna

The receiver component parallels the transmitter in form but has an inverse function; it is simply an antenna designed to receive the waves of electromagnetic energy that bounce off objects in the environment. If the receiver has knowledge of the time that the waves were transmitted, this knowledge, along with the azimuth and elevation of the receiving antenna, can be used to infer the position of a target. Some radars use the same antenna for receiving and transmitting; however, others have separate receivers or multiple receivers, enabling them to listen for pulse returns in different locations.

Radar systems require complex control and processing software. Software must control the movement of the radar pedestal. Software also controls the power and frequency of the transmitted pulses; in some radar systems, these variables can be dynamically adjusted through software to search for different types of targets or account for changing conditions.

The radar hardware is controlled by a real-time software program referred to as the Radar Control Program (RCP). The RCP directly configures the radar hardware by passing along Universal Control Messages (UCM), which set various parameters such as transmission power and frequency. Controlling and dynamically configuring the radar hardware is one of the main roles of a radar software system and must be done constantly to keep up with changing aspects of the environment.

Besides controlling the hardware, another role of the radar software is to format and process the data returned by the receiver. These *Pulse* data describe the amount of energy detected by the receiver at a given time. The *Pulse* data are merged with a secondary set of data, called *Auxiliary (Aux)* data, that describe the state of the hardware when the original pulse was transmitted. Given both *Pulse* and *Aux* data, a radar system can calculate the position of the target that reflected the pulse energy. One of the key tools for analyzing radar returns is the radar equation, shown in Figure 6 below:

$$\begin{array}{cccccccc}
 & \text{Transmit} & \text{Transmit} & \text{Spread} & \text{Losses} & \text{Target} & \text{Spread} & \text{Receive} & \text{Dwell} \\
 & \text{Power} & \text{Gain} & \text{Factor} & & \text{RCS} & \text{Factor} & \text{Aperture} & \text{Time} \\
 \text{Received Signal} & & & & & & & & \\
 \text{Energy} & = & [P_T] & \left[\frac{4\pi A}{\lambda^2} \right] & \left[\frac{1}{4\pi R^2} \right] & \left[\frac{1}{L} \right] & [\sigma] & \left[\frac{1}{4\pi R^2} \right] & [A] & [\tau]
 \end{array}$$

Figure 6 – The radar equation [O'Donnell, 2002].

In this model, the attributes of the original pulse sent out are considered, as well as all the losses caused by the environment as it travels towards the target. The equation starts with the initial power of the transmitted pulse (P_T). It then accounts for the transmit gain due to the area of the antenna (A), also known as the antenna's *aperture*, which is inversely proportional to the wavelength (λ). A larger antenna area results in a greater gain on the signal transmitted. Next, the spread factor of the wave (

$\frac{1}{4\pi R^2}$) decreases the strength of the wave based on the distance (R) between the transmitter and the target, as it diverges in an arc across the environment. Losses (L) such as hardware inefficiency and atmospheric energy loss are factored in next, again reducing the strength of the signal energy. The target's RCS (radar cross section, σ) is the next factor, as a larger target will cause a greater amount of energy to reflect off of the target, back across the environment. Finally, the spread factor of the returned wave across the environment is noted, along with the aperture or area of the receiving antenna. The pulse duration (τ) is the amount of time for which the radar is transmitting, as more energy is radiated into the environment when transmitting for a greater amount of time.

Since radar software is aware of the state of the hardware, all of the factors in the range equation can be either computed directly or measured. While the model is not perfect, it provides a reliable approach for radar software to perform analysis of radar returns.

2.2 ROSA II

The majority of radars in existence have been developed using proprietary hardware and software. There has been very little standardization of the hardware and software components that are used in radar systems, and many of these components have been custom-designed for specific radars. When a proprietary hardware component breaks in a particular radar system, an expert trained to work with that specific radar must be brought in to resolve the problem. Proprietary hardware and software makes radar systems difficult to understand and maintain in the long term. As more and more time passes, specialists must be continually educated to understand these legacy systems and to maintain the functionality of their hardware and software.

The Radar Open Systems Architecture (ROSA) defines an open standard to modularize the hardware components of radar systems and encourages the use of commercial, off-the-shelf (COTS) products.

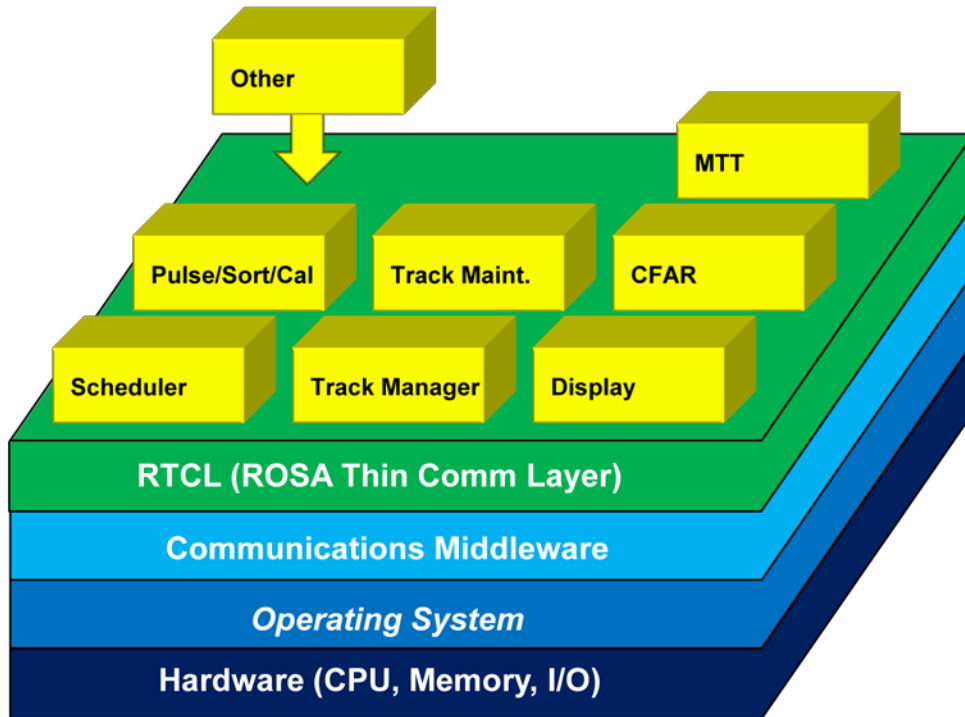


Figure 7 – Block Diagram of a ROSA II System

A subsequent project, ROSA II, defines a standard software architecture for radar systems. ROSA II is a framework for developing radar systems and other sensor systems. The ROSA model decomposes a radar-processing and -control architecture into individual, loosely-coupled *components*. Each component performs specific radar functions and can run completely autonomously. When combined, these building-block components form the entire processing and control architecture for a complete radar. ROSA II components communicate using well-defined interfaces, allowing developers to make changes within individual components without affecting the rest of the system. The loose coupling of ROSA components makes it easy to modify or upgrade a radar system. It is easy to replace one component with another component if they use the same interface to communicate with the rest of the

system. For example, a component that describes the pedestal steering system can be replaced with a new steering component to change the steering system without needing to alter any of the other components. ROSA II's middleware communication layer is the key to the loose coupling between components that makes ROSA systems flexible and maintainable.

2.3 RTCL

The ROSA Thin Communications Layer (RTCL) is a publish/subscribe message passing layer that provides a common interface to several different inter-process communication middleware implementations. RTCL was developed by MIT Lincoln Laboratory for the Radar Open Systems Architecture (ROSA) to isolate ROSA components from any specific communications middleware. ROSA systems are composed of multiple processes running on many different machines. These processes must pass messages both within the same machine and across the network. The best communication middleware to use depends on the hardware layout; shared memory might offer the best performance on a single machine, but a network based middleware such as the Data Distribution Service might be appropriate to pass messages over an Ethernet network. RTCL is configured at run-time using a configuration file to specify which communications middleware should be used for sending messages. RTCL provides flexibility to ROSA systems, allowing them to run on a variety of different hardware layouts without requiring changes to code. RTCL currently supports RTI Data Distribution Service (DDS), shared memory, Mercury Computer Bridge, and Java-Script Object Notation (JSON).

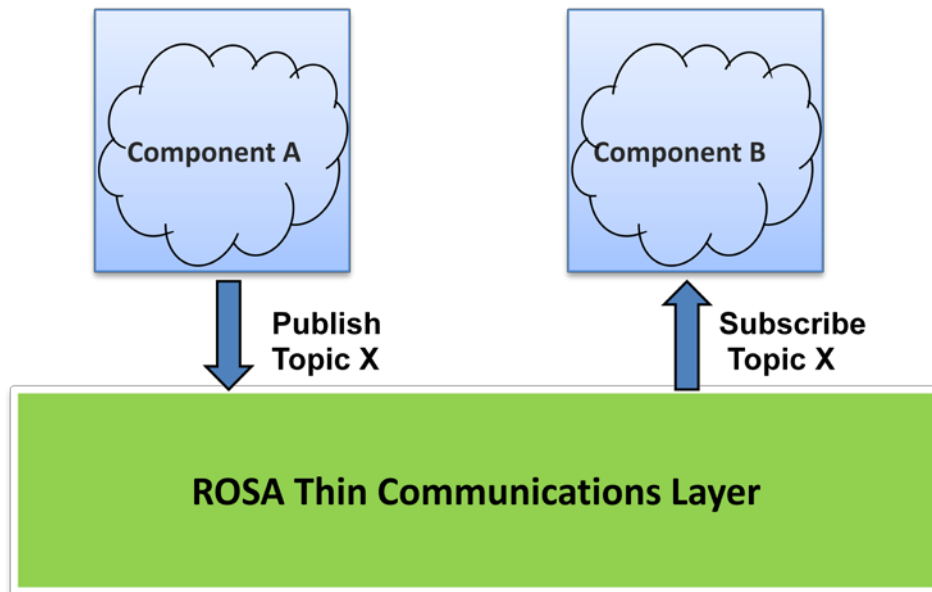


Figure 8 – RTCL Publish/Subscribe

RTCL uses a publish/subscribe messaging paradigm. Publishers write messages to a specific topic name, and subscribers receive messages associated with a specific topic name. The main advantage is that publishers and subscribers are decoupled and do not need to know about each other. RTCL messages are defined as Interface Definition Language (IDL) data structures. RTCL generates C/C++, Java, and Python data structures from the IDL definitions for use by user programs.

2.4 The Legacy Simulator

The legacy radar simulator was developed at MIT Lincoln Laboratory to provide a testing environment for ROSA radar control software. The modularization of ROSA makes it possible to replace radar hardware components and subsystems with a software simulation. As mentioned in Section 1.3, the simulator mimics radar hardware and simulates radar returns from targets in a virtual environment. Radar developers at the Laboratory use the simulator to test their software before deploying it with real hardware, saving time and money.

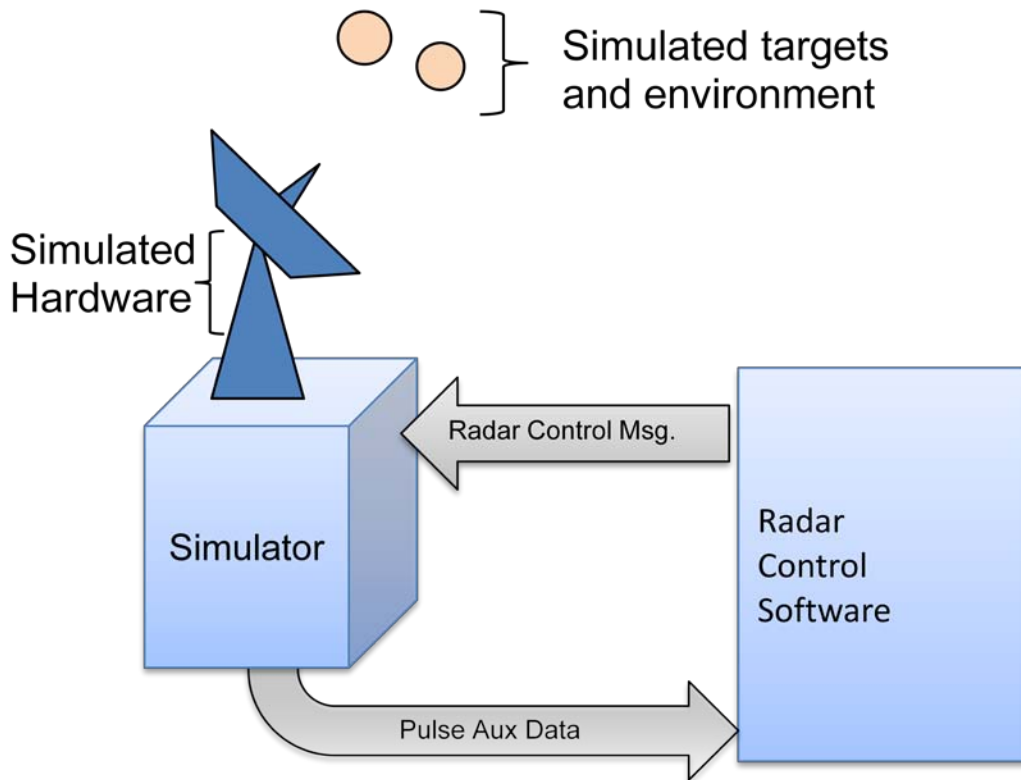


Figure 9 - Radar simulation

The simulator's design can be broken down into three logical parts: the *hardware*, the *environment*, and the *radar control program (RCP)*. The RCP is the external radar control program that the simulator is intended to test. The hardware accepts control and configuration messages from the RCP and sends pulses to the environment. The environment responds with returns, representing energy that bounces off targets. The hardware listens for returns, similar to a real radar receiver, and sends them back the RCP for processing. The RCP does not know that the returns are coming from a simulation; it treats them like real radar returns and attempts to decode them and identify targets.

In order to accurately simulate radar hardware, the simulator meets several important requirements. The RCP uses timing information about the returns it receives in order to determine the range to targets. The simulator must reproduce these times accurately in order to successfully mimic returns from real-life hardware. Radar pulses travel at the speed of light, and the simulator must finish

its complex calculations before the pulses would bounce off targets and arrive at a real-life radar receiver. The simulator supports advanced features such as configurable scenarios, complex waveforms, and environmental effects. Appendix A contains a detailed list of the simulator's features.

2.5 Simulation

A computer simulation is “a computation that models the behavior of some real or imagined system over time” [Fujimoto, 2000]. Some simulations go beyond simply modeling an approximation of a system and attempt to fully mimic its behavior, timing, and other characteristics. Simulation is a practical way to study large, complex systems such as weather patterns and air traffic. Simulations can be used to perform statistical analysis, to test and evaluate hardware, and to train personnel. Simulations are often more practical than physical tests that may be costly, infeasible, or even dangerous to carry out.

2.5.1 Analytic Simulations and Virtual Environments

Modern computer simulations fall into two main categories: analytic simulations and virtual environments. Analytic simulations are typically used for mathematical analyses of complex systems. They usually execute as fast as possible and have no external input or interaction. Analytic simulations predictably reproduce before & after relationships. Virtual environments are simulations that create a realistic or entertaining representation of an environment. They execute in real time and often accept external or user input. A human might control the behavior of entities within the simulated environment. Physical components can be integrated with virtual environments for testing purposes; for example, a missile defense system can be tested using a virtual environment that simulates missile trajectories. Virtual environments only need to preserve before & after relationships if humans or physical components that interact with the simulator can perceive them.

There are two common methods that computer simulations use to model systems over time. The first approach is time-based simulation, which advances time by a fixed interval at each step and then re-computes the state of the system. The second is event-driven simulation, which models systems as a chronological sequence of events that indicate changes in state of the system. Because it does not rely on a fixed time interval, event-driven simulation is often more efficient than time-based simulation and is the primary focus in our simulator research [Fujimoto, 2000].

2.5.2 Sequential Discrete-Event Simulation

Sequential discrete-event simulations have three major components: state variables, an event queue, and a simulation clock [Fujimoto, 2000]. State variables model the state of the simulation. The event queue contains events that occur at specific times in the simulation. An event contains a timestamp, a type, and some parameters. Creating a new event is called *scheduling* an event. A simulation clock variable denotes the current simulation time, and it is only possible to schedule an event that takes place in the future. If the simulation clock is at time T , it means that all events before time T have been simulated. The event with the smallest timestamp after T is processed next. Simulations must also keep track of physical time (time in the physical system) and wall clock time (time during the execution of the simulation program). To avoid advancing faster than wall clock time, the simulator can wait before advancing to the time stamp of the next simulated event until it matches the wall clock time.

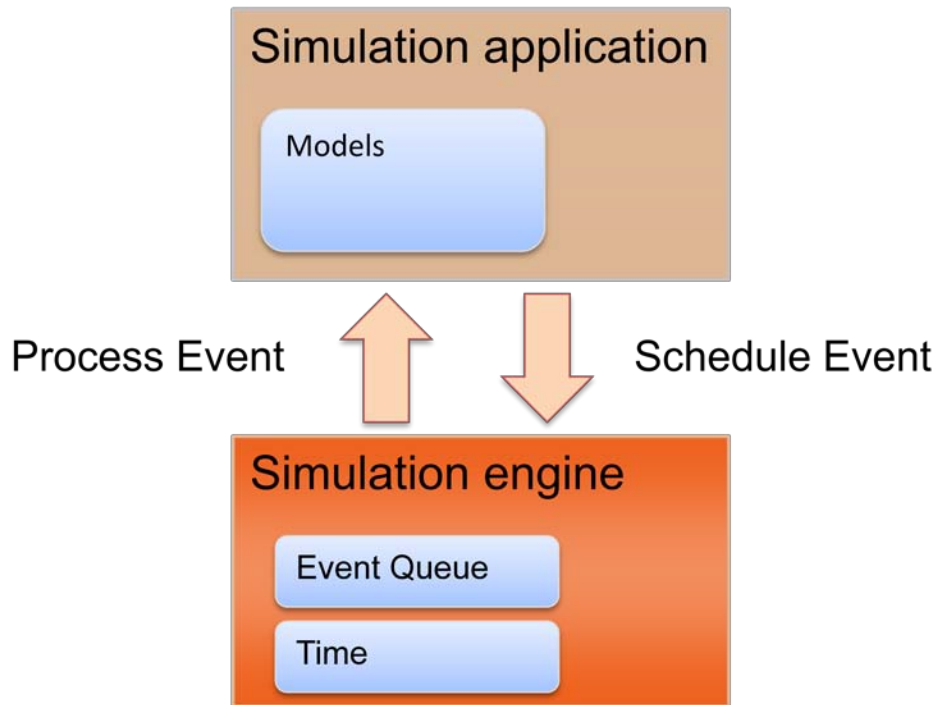


Figure 10 - Simulation Engine and Application

Event-driven simulations can be divided into two layers: the simulation application and the simulation executive, shown in Figure 10 [Fujimoto, 2000]. The simulation application contains the simulation's state variables and the code modeling the system behavior. The simulation executive maintains the event list and manages advances in simulation time. The application directs the executive to schedule events, and the executive delegates the events to the application for processing.

2.5.3 Open Architecture Simulation Interface Specification (OASIS)

To standardize and expedite the development of new simulations, MIT Lincoln Laboratory has developed a generic simulation framework called the Open Architecture Simulation Interface Specification (OASIS). The OASIS framework is structured in a set of independent layers that encompass all of the functionality necessary to create a simulation. Each layer isolates a logical portion of the simulation and encapsulates its own specific design and implementation details. Figure 11 shows the

layers of the OASIS framework. The placement of the layers adjacent to each other in the diagram indicates that they interface directly with each other during a simulation.

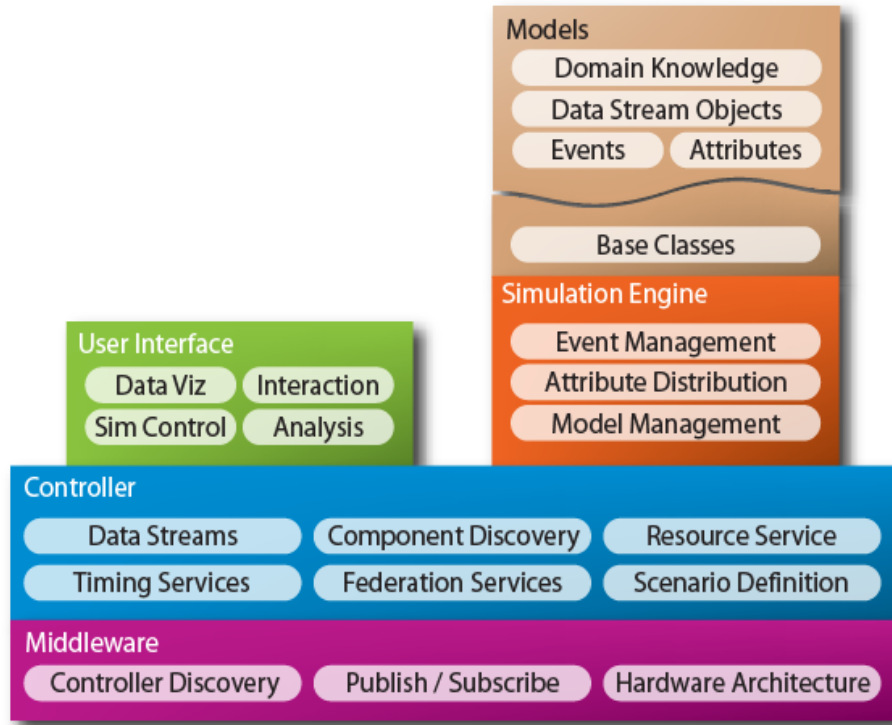


Figure 11 – The OASIS Layers.

At the heart of the framework is the simulator controller. The simulation controller is responsible for setting up and running the simulation and for managing the flow of information to and from the various components of the simulation. The controller coordinates the master simulation clock and manages the simulation set-up and configuration by loading scenario scripts. The controller also manages the logging of simulation events and operator actions and provides an interface to external analysis tools.

The simulation engine layer interfaces with the simulation controller and maintains a queue of simulation events with time stamps. The engine creates and maintains model objects from the model layer. The engine executes events in time-increasing order on the simulation models. The simulation

engine coordinates the simulation time with the master clock in the controller layer. A complete simulation that models multiple domains might have a separate simulation engine for each domain. The simulation engines can exchange events that are relevant to each other by passing them through the simulation controller.

The simulation engine interfaces directly with the model layer. The model layer defines the domain-specific models that will interact with one-another during the execution of the simulation. The models are representations of objects in the simulation, such as targets, sensors, and the environment. Models interact with each other through events, and they can schedule new events with the simulation engine.

The controller interfaces with a middleware layer, which encapsulates communication with external components and translates between different messaging formats and protocols. The middleware provides a common interface to inter-process communication both on the local machine and over the network. The middleware can be used to send simulator events to other processes, enabling multiprocessor execution of the simulator. The middleware allows for remote management of the simulation as well as providing remote database access and task synchronization.

The controller layer also interfaces with a custom component layer, also referred to as the user interface layer. The custom component layer provides an interface for external software components to interact with the simulation. Custom components can serve as external graphical displays of the simulation and can provide interactive input and control during the execution of a simulation. Custom components even include legacy simulators that interact with the OASIS simulation.

2.6 Distributed Systems

A *distributed system* is any software system in which multiple computers are used in parallel to perform a subset of the total work that needs to be done to solve a problem. Machines in a distributed

system are generally networked together so that they can cooperate on problems and divide up computations that need to be performed. When each machine finishes its specific part of the calculations to be performed, its results are sent back and synchronized to produce a final result.

One of the most appealing features of distributed systems is that they can easily be designed for scalability in processing. Put more simply, a distributed system makes it possible to gain additional computational power for processing by adding more machines. This is mathematically explained within Amdahl's Law [Amdahl, 1967], shown in Figure 12, which quantifies the highest possible speedup factor of a system with N processors that spends s fraction of time on serial tasks and p fraction of time on parallel tasks. ($S+P=1$).

$$Speedup = \frac{s + p}{s + \frac{p}{N}} = \frac{1}{s + \frac{p}{N}}$$

Figure 12 – Amdahl's Law

An important issue to note is that as the number of nodes increases to infinity, the $\frac{p}{N}$ term converges to zero. Thus the maximum attainable speedup is limited not by the number of nodes, but by the amount of processing that must occur in serial. This limitation means that in order to achieve better performance, reducing the amount of serial processing is more important than simply introducing additional computer nodes. As a result, one of our primary goals is to design with parallelization in mind in order to keep the amount of serial processing that must occur to a minimum.

2.7 Real-time Computing

One important requirement for radar simulators used for testing is that they must perform in real-time so that they can interface with the real-time radar control programs used by the Laboratory. A real-time radar simulator must run deterministically to ensure it can accurately model an entire radar and environmental system with the same timing as the physical events that would occur in real life. For

example, if a pulse is sent out at a certain time, the simulator must consistently produce the appropriate return at least within the time it would arrive at a real-life radar receiver. The radar control program relies on this timing information to identify and track targets.

The most important aspect of a real-time system is being able to support a high level of determinism in the amount of time it takes to perform a computation. This does not mean that a real-time system has greater throughput than a non-real time one, but rather that it completes its tasks within a consistent, well-defined period. A non-real-time program may execute a task within 5 microseconds on average, but occasionally have spikes of 30 or 40 microseconds to complete the calculation if the processor is busy. A real-time program may run at 7 microseconds on average, but does not deviate significantly from this number.

The most important metric for measuring the real-time capability of a system is the *worst case* system latency. This is the absolute longest amount of time that a task takes to run when executed a large number of times. In the example from the previous section, the non-real-time program has a worst case latency of 40 microseconds, even though it has a better average latency. However, the real-time program has a worst case latency of only 7 microseconds, which allows engineers to base their designs around this expected amount of time. Worst case latency must be confirmed over a large number of trials in order to truly be useful; generally, hundreds of thousands of trials must be performed to accurately characterize a real-time program.

3 Methodology

This section details all of the resources, practices, and tools that were used by the group in order to complete the project. It includes descriptions of the development tools used for programming, the software engineering practices used by the group, as well as the software used for testing and documentation. It also details the timeline for the project, as well as how work was distributed among the group members.

3.1 Work Environment and Tools

The majority of the work for this project was performed at the Lincoln Laboratory campus using facilities and project resources provided by the project sponsor. Lincoln Laboratory provided the team with three laptops and two high-performance servers to use as test-beds for research and project development. One of the team's greatest resources was the radar experts at the Lab and within the group. In order to learn about radar, the project team had access to the extensive Laboratory library and a ten-hour video lecture series [O'Donnell, 2002] produced at the Laboratory.

3.1.1 Language Choice

When selecting the language to use to implement DVERT, the team considered several factors. The sponsor requested that the team use a major object-oriented language such as C++ or Java, two languages that are commonly used for development at the Laboratory. The team had more experience with Java than C++. A major factor in the language choice was the strict real-time requirement of the simulator. The team performed a brief benchmark comparing the real-time performance of C++ and a real-time variant of Java on a Linux system running a real-time kernel.

The team's benchmark of C++ found determinism down to the microsecond level, which is well within the requirements of the real-time systems intended to be modeled and simulated. Applications were consistent in their runtime and usually only deviated by a few microseconds of latency over

thousands of trials. Many existing real-time applications are written in C++, demonstrating that C++ programs are capable of real-time performance. Even Cyclictest, one of the most prominent real-time operating system benchmarks [Gleixner, 2010], is written in C++. This confirms our confidence in the real-time capabilities of the C++.

Next, the team explored Real-time (RT) Java, a variant of the Java Virtual Machine created by Sun specifically for real-time applications. This variation is compatible with the Red Hat MRG Real-time Linux kernel [Red Hat Inc, 2010], and is coded and compiled with almost the exact same syntax as normal Java. The team was hoping to make use of RT Java due to our prior knowledge of the Java language, its ease of memory management, and its cross-platform support. Additionally, java has a number of excellent development tools such as JavaDoc [Oracle Inc, 2004] for documentation, JUnit for unit tests, and EclEmma [Hoffman, 2010] for code coverage. However, the benchmark of RT Java found that the worst case latency occasionally peaked to the hundreds of microseconds, making it unable to meet the real-time requirement of our project. Additionally, RT Java requires a special Java virtual machine and would have represented an additional dependency for the simulator. Technical papers on real-time Java confirmed the results of the benchmark [Kalibera, 2009], leading the team to decide to use C++ as the implementation language.

3.1.2 Integrated Development Environment (IDE)

After selecting a language, the team then selected an accompanying integrated development environment, or IDE. Ideally, an IDE should be a comprehensive set of tools that facilitate the software development process. Typical IDEs assist with actions including coding, compiling, building, debugging, and testing.

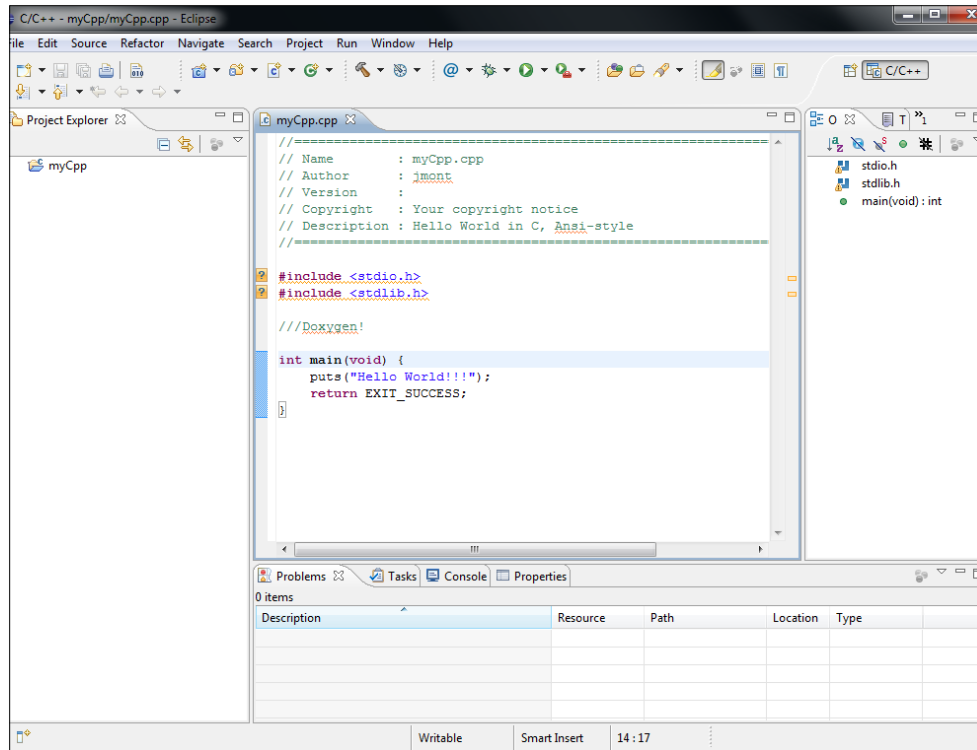


Figure 13 – The Eclipse Integrated Development Environment

We selected Eclipse for C/C++ as our IDE [The Eclipse Foundation, 2010a]. Eclipse is a popular IDE for C++ and Java development because it supports many powerful features and is free, open source software [The Eclipse Foundation, 2010b]. It has an easy-to-use graphical interface that organizes logical groups of code into *Projects* and makes coding and switching between different files very simple. It makes large projects easier to manage with features like code completion, automated refactoring, and automatic code indexing and searching. Eclipse for C/C++ gives users full control over building their applications by allowing them to specify and save multiple build and compiler configurations. By creating a managed makefile project, users do not have to maintain their own Makefiles in order to build their applications; eclipse generates Makefiles automatically, saving development time. Finally, Eclipse supports a number of plug-ins developed by the community for integrating even more useful features such as built-in documentation and version control [The Eclipse Foundation, 2010c]. All of these

features, combined with the fact that all project team members were familiar with Eclipse from previous software engineering projects, made Eclipse ideal for the project.

3.1.3 RTCL

One of the key requirements of the simulator this project seeks to design and implement is that it must be distributed. The processes in any distributed application must communicate with each other. Inter-process communication in a real-time setting is a common problem that has already been solved at Lincoln Laboratory. The project team decided to use the ROSA Thin Communications Layer (RTCL), described in section 2.3, to manage communication between processes in the simulator. RTCL was developed at the Laboratory for use in distributed ROSA II systems. It is written in C++ and has an Application Program Interface (API) for C++ and Java applications. One of the project team members worked on a Laboratory-sponsored project over the summer to create a Python API for RTCL, and, as a result, had prior experience using RTCL. RTCL meets the real-time requirements of the radar simulator and is already used in the radar control software developed at the Laboratory. The Laboratory plans to support communication between radar subsystems and the radar control software directly through the RTCL middleware layer in the future. RTCL will allow direct middleware communication between the project team's radar simulator and the radar control software when this work is complete.

3.1.4 Boost

The project team used of a popular set of C++ libraries collectively referred to as Boost [Dawes, 2007]. Boost is a group of open-source C++ libraries that extend the functionality of C++ and work well with the C++ Standard Template Library (STL). Boost provides several useful, cross-platform libraries that were extremely helpful in the development of the simulator. These libraries are Boost Thread, Boost Any, Boost Date Time, and Boost Smart Ptr.

Boost Thread provides a simple interface to threading and synchronization. Boost Thread is portable, unlike the commonly used platform-specific posix thread library. The Boost Thread API is also simpler to use than the pthread library. The Boost Any library enabled polymorphic lists and greatly simplified some of the event broadcasting functionality of the simulator.

The Boost Date Time library provided a portable set of classes and conversions for dealing with time. It provides developers with functions that can easily calculate the difference between two dates or convert from one unit of time to another. This functionality is neatly contained within a class called a boost ptme, or posix time. The team used boost ptme objects to represent time in the simulator.

The Boost Smart Ptr library defines the shared_ptr class, which the project team used extensively in the simulator. Shared pointers simplify memory management for objects with shared ownership in an application. Shared pointers maintain a reference count to the objects they point to. When all references to the data have been deleted or are out of scope, the shared pointer automatically frees the data to prevent memory leaks. With the huge amount of data and messages being passed back and forth within the simulator, not having to worry about memory management issues is a tremendous development advantage.

3.1.5 Version Control and Collaboration

A version control system is essential for managing any extensive software engineering project, especially one in which there is collaboration between multiple people. Version control is a system of keeping a central repository of project code and managing the alterations and history of a project as changes are made. This includes updating files when changes to code have been made, merging files when conflicts due to overlapping changes arise, and documenting the differences made at each stage, or revision, of a project. The general goal is to keep one master repository of code, allowing developers

to make changes to this repository while simultaneously keeping the rest of the developers updated with these changes.

The team decided to use the Apache Subversion (SVN) version control system [Apache Software Foundation, 2010]. Having chosen the Eclipse IDE, Subversion was a logical choice due to the fact that it can be very smoothly integrated into Eclipse using the Subclipse plug-in [Collabnet, Inc, 2009]. This is a free, open source Eclipse add-on that allows developers to import a project from an SVN repository and manages their personal changes while keeping the project up to date with changes from other developers. It is integrated very cleanly into the Eclipse GUI with a pop-up menu that allows users to commit their changes, update files edited by other members of the project, and merge files when changes overlap or conflict. Like Eclipse, the project team was already familiar with Subclipse, as all of the members had used it on previous projects. Therefore, no time was wasted in learning how to use the plug-in.

The team hosted the project's Subversion repository on the Lincoln Laboratory Teamforge server. This is a server internal to the Laboratory, which houses code repositories and documentation for various Lincoln Laboratory projects. Committing and downloading any code was as simple as specifying the location of the repository on the server.

3.1.6 Test Cases

Test cases are an effective way for programmers to ensure that their code operates as it should. In general, a test case creates a certain scenario within a program and checks to make sure that certain expected conditions throughout the program are met. There are a variety of utilities used to facilitate writing and organizing tests. The team was hoping to find a testing solution that could integrate with the Eclipse IDE. The team performed a brief trade study on two C++ unit testing plug-ins for Eclipse: CUTE

[Sommerlad, 2006] and ECUT [Hartsberger, 2008]. Unfortunately, these unit testing frameworks proved inadequate for the project. They lacked essential functionality and caused instability within Eclipse.

In the end, the team chose CxxTest, a minimalistic C++ testing framework [Fitch, 2009]. CxxTest allows users to write any number of test cases, which are then attached to a single *runner* or executable file that runs each test in sequence. It reports errors if any tests fail, as well as any specific messages or traces specified within tests. Unlike most other testing utilities, CxxTest has no external dependencies. The key reason for selecting CxxTest was its ease of integration with our code base; no separate installation of CxxTest is required to run the tests. All of the necessary CxxTest files are included as part of the project. Unfortunately, an Eclipse plug-in did not exist for CxxTest.

3.1.7 Code Coverage

As software engineering projects increase in size, it becomes difficult to determine whether unit tests are testing all of the code that has been written. A code coverage system helps solve this problem by identifying which lines of code are actually being called when the program executes. When combined with test cases, a code coverage system reveals untested lines of code and identifies the percentage of the software that is being tested. This is a useful metric for determining the integrity of a piece of software; a high percentage of code coverage ensures that code is well-tested and bug-free.

Several code coverage tools were considered for this project. These included the GNU coverage tool, gcov [Free Software Foundation, 2010], a third party GUI tool called Test Cocoon [Fricker, 2009], and an enterprise-level code coverage tool called BullseyeCoverage [Bullseye Testing Technology, 2010]. The team selected BullseyeCoverage because it was easy to install and use and it had GUI tools that made it simple to locate untested code. Like CxxTest, BullseyeCoverage does not integrate with Eclipse.

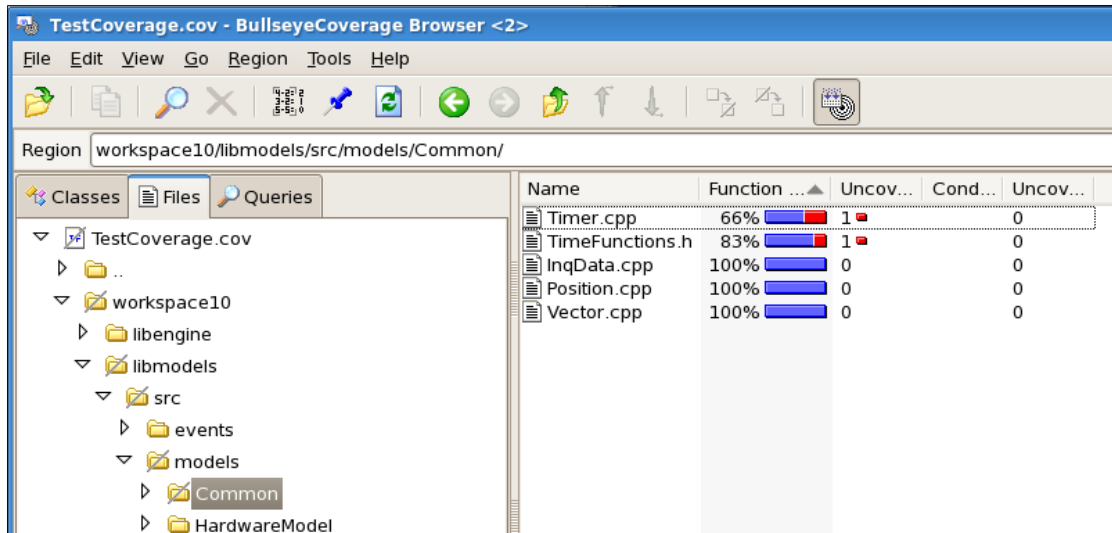


Figure 14 – The BullseyeCoverage Browser

BullseyeCoverage functions very uniquely compared to most code coverage tools. Once installed, it can be enabled to directly intercept the g++ compiler when a project is built. It uses this information to profile a program and create a database of the various libraries included within the project. It then determines the coverage of the entire project when this program is executed, producing a *.coverage file. This coverage file is then opened with the *CoverageBrowser* GUI, a tool that displays the coverage results on a per-file basis. It profiles the percent coverage of each file, as well as which lines were executed during runtime. It even provides a coverage metric for conditional logic, indicating the percentage of possible logic branches exercised by the unit tests, which gives a more complete view of a program’s test coverage.

3.1.8 Documentation

Besides inline documentation within code files, many developers seek a more comprehensive solution for documenting the structure of their programs. For this project, the team used a tool called Doxygen [Van Hesch, 2010], which generates documentation for a project automatically by extracting the comments contained in the source code. For example, it can take a C++ class and automatically

generate an HTML document that displays the class's inheritance diagram, states its methods and properties, and displays comments from the source code, as shown in Figure 15.

Main Page	Classes	Files
Class List	Class Index	Class Hierarchy
Class Members		

PhysicalModel Class Reference

Superclass for physical model objects with 3D position, orientation, and motion. [More...](#)

```
#include <PhysicalModel.h>
```

Inheritance diagram for PhysicalModel:

```

graph BT
    Receiver --> HardwareModel
    Transmitter --> HardwareModel
    K650Transmitter --> Transmitter
    PointSourceTarget --> Target
    HardwareModel --> PhysicalModel
    Target --> PhysicalModel
  
```

[List of all members.](#)

Public Member Functions

	PhysicalModel (shared_ptr< Trajectory > inTrajectory)
shared_ptr< Trajectory >	getTrajectory ()
void	setTrajectory (shared_ptr< Trajectory > inTrajectory)

Figure 15 – A sample page of doxygen, which documents information for a specific class

Doxygen allows developers to write code and documentation simultaneously. Maintaining separate documentation is a time-consuming process, and generating documentation from source code ensures that the documentation is never out-of-date. Doxygen makes it easy to maintain documentation for large code bases.

Doxygen uses what is referred to as a doxyfile, in order to configure itself before generating documentation. Every separate project requires its own doxyfile, which is used to specify the directory of the source code, where to put the output documentation, as well as a number of configuration parameters. To make the use of Doxygen even simpler, the team used a GUI extension called Doxywizard. By simply specifying the directory of the project, the language being used, and the directory

of the output, a user can generate documentation with minimal work and no need to manually edit a doxyfile.

Doxygen also can interface with the GraphViz [AT&T Research, 2010] visualization tool in order to generate more sophisticated graphs and diagrams. Moving beyond the basic inheritance diagram for a single class, GraphViz can create class diagrams for the entire codebase, clearly showing any and all dependencies that exist between classes. This greatly simplifies the laborious process of manually creating graphs to explain the overall structure of the simulator.

3.2 Software Engineering Practices

In order to create a simulator that met the requirements of the sponsor, the project group employed a variety of standard software engineering practices. These included iterative design and development, as well as close collaboration with Lincoln Laboratory experts throughout the duration of the project.

3.2.1 Iterative Design and Development

One of the most important practices used by the group was iterative design. The design of the simulator evolved over the course of the project. The team added new features each week and modified the design to support them. The team continuously acquired new knowledge about radar systems and applied it to the design. The team also gathered feedback and new requirements for the design at weekly meetings with the project advisors and Laboratory staff. The team used this feedback to revise the design and submit the changes for review at the next meeting. Using this process, the group steadily moved towards a stable final design and was able to ensure the proper direction of the project each week.

A similar approach was taken for the development of the implementation, which was performed in parallel with the design of the project. The models and engine code were regularly iterated on to

reflect changes made to the design. Code within the project testing suite was also regularly revised to represent changes made within the implementation. Coding the implementation in parallel with developing the design provided valuable validation for each iteration of the design and reinforced the integrity of the design as a whole.

3.2.2 Sponsor Collaboration

In order to gain proficiency in the domain-specific backgrounds related to the project, the group consistently leveraged expertise from members of the Laboratory. In addition to weekly meetings, separate meetings were scheduled to discuss how radar systems function and the mathematical modeling behind a radar simulation. By pulling from these resources, the project group was able to apply existing knowledge at the Laboratory in order to create a more robust and professional simulator design.

3.3 Procedural Timeline

The timeline for completing the project was divided up and organized by week at the start of the project using a Gantt chart. The initial part of the project consisted of a proposal phase, in which the project group organized background concepts and specifically defined the scope and intent of the project. For this phase, the group prepared both a written proposal and an oral presentation in order to receive approval from the sponsor and advisors to begin the project. The next phase of the project was the commencement of the MQP, in which the group carried out all the design, implementation, and documentation necessary for the project. The weekly timeline for the entire project is outlined below:

Week 1:

- Defined requirements for new simulator and characterize features of legacy simulator
- Performed background research on radar, real-time systems, and distributed systems

- Performed trade studies to determine project language choice
- Investigated and became familiar with existing simulator
- Met with OASIS simulator experts from the Optical Systems Technology Group
- Began writing project proposal
- Began organizing proposal presentation

Week 2:

- Completed initial macro-level overview of simulator design
- Began micro-level design of radar hardware simulator
- Reviewed initial design with Laboratory staff
- Delivered written proposal and proposal presentation
- Selected development tools and set up workspace to prepare for implementation
- Produced a “blank” simulator with the appropriate interfaces required by our ROSA II system

Week 3:

- Finished design of hardware models
- Implemented the initial set of simulator hardware models
- Began designing and implementing simulation engine
- Carried out initial design review with advisors and LL staff; made revisions as necessary
- Began creating doxygen documentation
- Wrote first draft of Introduction and Background report sections
- Began writing unit tests

- Began measuring and recording models and engine code coverage

Week 4:

- Completed message passing/middleware integration into the engine
- Finished initial engine class-level design
- Finished initial Target model class-level design
- Began implementing models for the Target simulator
- Carried out major mid-project design review with LL staff; revised design as necessary
- Revised Background and Introduction report sections with advisor comments
- Wrote first draft of Methodology report section

Week 5:

- Finished implementing simulation engine
- Began integrating models with engine in the implementation
- Set up a working ROSA II system to interface with the new simulator
- Created the Component base class used by all simulator processes
- Revised Methodology with advisor comments

Week 6:

- Performed a major re-structuring of model classes to make them more organized
- Continued debugging integration of models and engine in the implementation
- Implemented generic models extending transmitter, target, and receiver in order to test implementation output
- Wrote first draft of Design and Implementation report section
- Outlined Results and Analysis section

Week 7:

- Completed all model calculations and message passing through the simulator; verified that the proper returns were being received.
- Began adding time synchronization between simulation engines
- Added logging within the implementation
- Revised Design and implementation report section with advisor comments

Week 8:

- Imposed “feature freeze:” ceased adding new features to the simulator
- Carried out final design review with Lincoln staff to gain approval from the Laboratory
- Completed timing synchronization within simulator
- Performed acceptance test to verify simulator output
- Wrote “User’s guide” documentation
- Created final presentation
- Reviewed final presentation with advisors

Week 9:

- Rehearsed for presentation dry runs
- Delivered presentation dry runs
- Delivered final presentation
- Finalized minimal working implementation of new simulator, using multiple targets and multiple Target simulators
- Imposed “Code freeze:” ceased writing new code for the project.
- Finished writing test cases to attain 80% code coverage for models and engine

- Wrote “Developer’s Guide” documentation detailing how the simulator could be extended to accomplish specific use cases.
- Wrote Results and Analysis and Conclusion sections of the report
- Assembled final report and finished making all revisions
- Submitted final report and deliverables

3.4 Division of Labor

The two main components of the project were the simulator and design and implementation, which were developed in parallel throughout the project duration. The initial macro-level design was a team effort, with all members contributing equally. Once the overall foundation for the simulator was decided, the team was able to branch off, with some members focusing on designing the next parts of the simulator while others worked on implementing the design.

Throughout the project, Lucas Scotta was the lead developer of the implementation. His initial work was in developing the “blank” simulator that mimicked the interfaces of the existing simulator. Later, he focused his efforts largely on implementing the generic simulation engine and message-passing components of the system. Lucas was well-suited for these parts of the project because he had pre-existing experience with using the ROSA Thin Communications Layer from his summer work at the Laboratory.

James Montgomery and Matthew Lyon worked in tandem to develop the hardware and target models with which the simulation engine would eventually interface. This involved continuous review with members of the Laboratory staff to ensure that the radar-specific attributes within the models were accurate and relevant. Upon completing the design of the models, they began implementing the model classes and preparing them for integration with the engine classes being developed by Lucas.

Upon separate completion of the simulation engine and models, the team regrouped and began working on integrating the model and the engine code together to produce a complete minimal implementation. Lucas led the effort to integrate the models and the engine code, while James worked on debugging errors occurring as a result of this integration. Matt began research into the time synchronization of the simulator, and he collaborated with Lucas to add these timing changes within the implementation. Unit tests were written by James and Matt, while Lucas wrote several larger integration tests to verify the simulator at the system level. DOxygen documentation was performed by all members of the group whenever new code was added and James wrote the User and Developer guides to supplement the project documentation.

All team members were expected to attend regular weekly meetings with advisors and members of the Lincoln Laboratory staff. For every meeting, the team was expected to send out an agenda one day in advance to the project advisors and planned meeting attendees. One member of the group was designated to chair the meeting, while another member was expected to assume the role of meeting secretary and record all relevant notes from the exchange. These roles alternated between the members every week, so that the roles of chair and secretary were evenly distributed throughout the duration of project. In order to keep the advisors and project sponsors aware of the group's progress, each group member was also responsible for sending out a weekly report detailing their individual progress for the week.

Report writing was evenly distributed over the course of the entire project. Matt and James did more report writing earlier in the project as Lucas focused on making progress on the implementation. Upon completion of the implementation, Lucas contributed a very large amount of content to the report and the entire team performed revisions and added the necessary sections to complete the report.

4 Design and Implementation

This section presents the design of a distributed radar simulator known as the Distributed Virtual Environment for Radar Testing (DVERT). The section begins with the requirements for the design, as determined by the project sponsor. The design itself is composed of several layers, inspired by the OASIS simulation architecture.

4.1 Functional Requirements

DVERT was designed to meet three primary requirements as specified by the sponsor. First, DVERT had to be able to interface with radar control software developed at Lincoln Laboratory. Second, DVERT's design had to scale to thousands of targets. Third, the simulator's design had to be easy to extend to support additional types of radar systems and targets. These requirements are outlined in Minimal Implementation Requirements As Specified By Sponsor (section 1.4 and Appendix B.)

4.2 Design Overview

Figure 16 shows a high-level overview of our architecture. The separation of the radar hardware and the targets and environment parallels the functionality of radar systems.

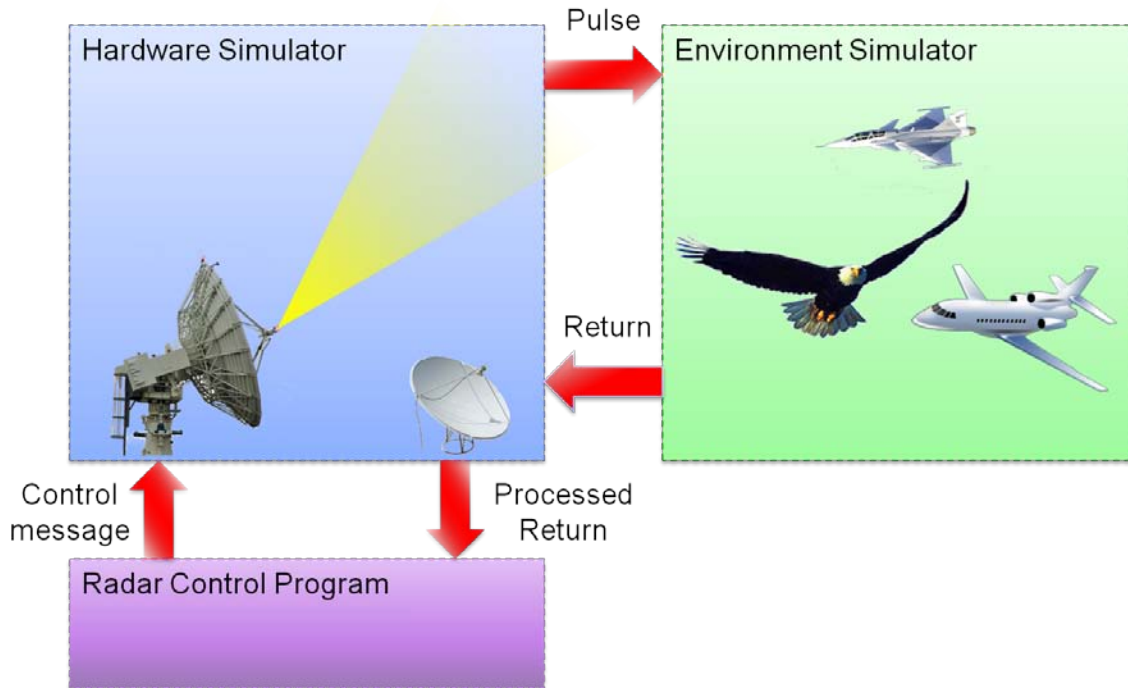


Figure 16 – Initial Macro-level System Design

The design is split up into a hardware simulator and one or more environment simulators. These simulators work together to simulate an entire radar system. The hardware simulator broadcasts pulse events to each environment simulator. Each environment simulator maintains a subset of the targets and calculates the radar returns for each of them when they are hit by a pulse. These simulators can run on separate machines, allowing the simulation to be distributed across multiple processors so it can scale to a large number of targets. The hardware simulator interfaces with the radar control software that the simulator is intended to test.

4.3 Layered Architecture

Our design uses a layered architecture similar to the one in the OASIS simulator framework, detailed in Section 2.5.3. We used the OASIS specification as a reference to guide the structure of our design. We modeled parts of our design on the OASIS framework because it addressed the need for

flexibility and future extensibility. OASIS's layered architecture provided the infrastructure necessary to support the domain-specific radar models we wanted to build.

The first step to adapting the OASIS architecture for our design was to break down each of the layers present in the OASIS framework and decide the purpose of each layer within the context of our simulator. This allowed us to create a revised diagram of the OASIS layers to fit the specific requirements of our simulator, shown in Figure 17, below.

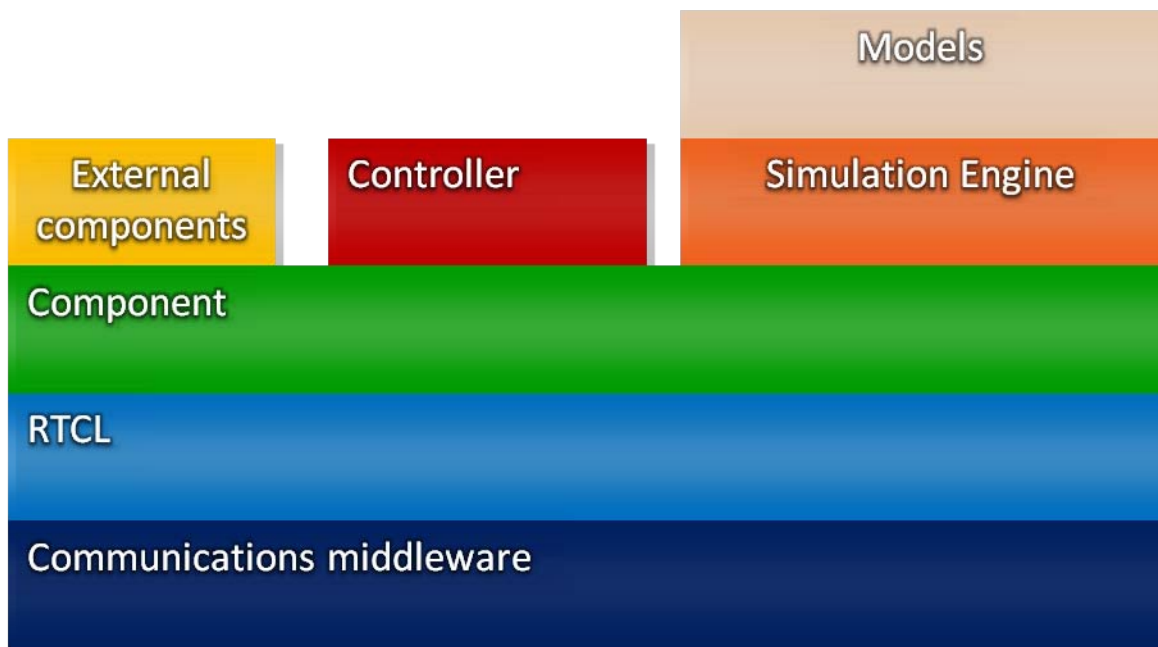


Figure 17 – The adapted OASIS layers used in the design of the new simulator

The specific details of each layer and their internal functions are described in more detail in the following sections.

4.4 Model Layer

The model layer defines the domain-specific radar models and events that interact during a radar simulation. A large part of the project involved designing the domain-specific models that would

be included with the minimal simulator package. This included creating model classes for transmitters, receivers, targets, and all related factors necessary to calculate a radar return from a transmitted pulse. The group worked closely with the staff at Lincoln Laboratory to ensure that these models appropriately described the attributes of a basic radar scenario while still being extensible enough to support more complex scenarios in the future.

4.4.1 Modeling the Radar Equation

All of the models used within the simulation were designed around the mathematics present in the radar equation (see Figure 6). This equation quantifies the amount of energy returned to a receiver from a pulse hitting a target in the environment. The equation can be broken up into the steps shown in the diagram below:

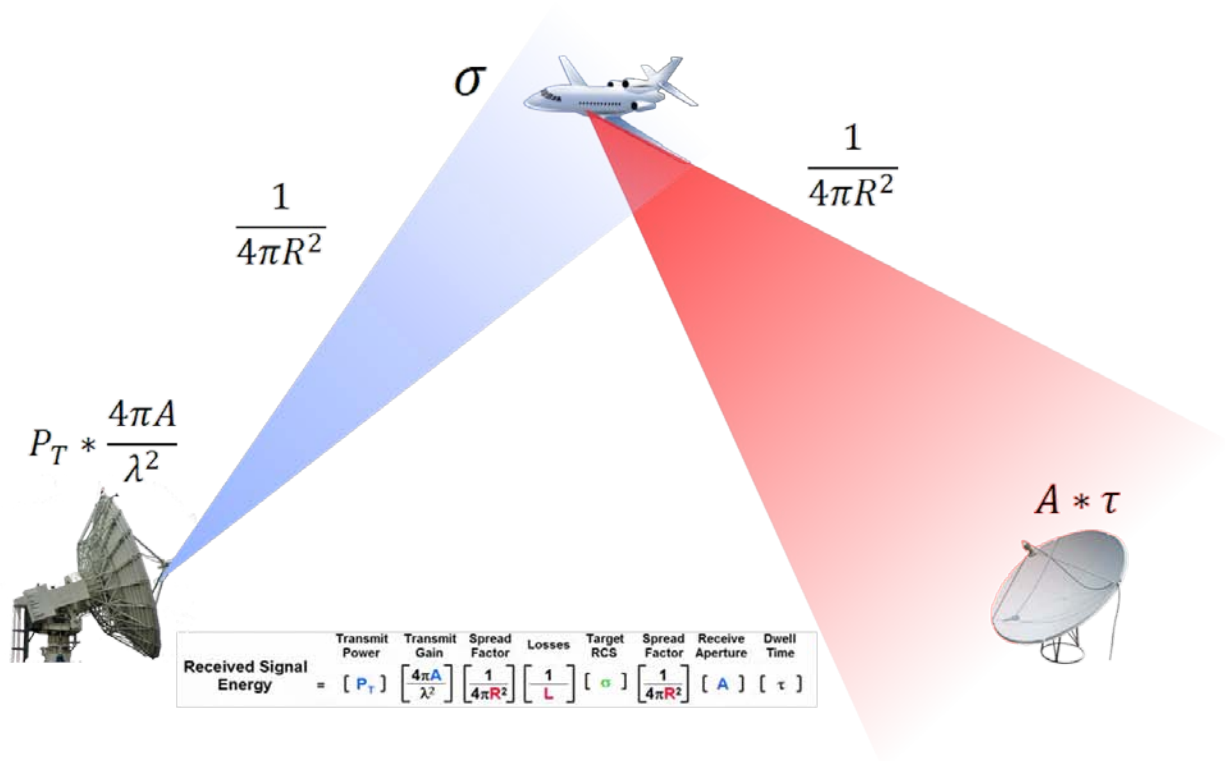


Figure 18 - Each step in calculating the radar range equation [O'Donnell, 2002]

The radar transmitter and antenna is responsible for determining the transmit power and gain, which determines the characteristics of the pulse that is transmitted. As the pulse travels from the transmitter to the target, it diverges with a spread factor that depends on the range to the target. Once the target is hit by the pulse, the specific geometry of the target as well as the angles between the target, transmitter, and receiver are used to determine the specific radar cross section of the target for a pulse. The energy that hits the target diverges again as it travels back to the receiver, and the amplitude of the final return is affected by the area of the receiver antenna and the amount of time that the receiver is listening for.

4.4.2 Models and Events

The physics presented in this equation can be described through the use of *models* within our simulator architecture. A *model* is an object that is capable of responding to and scheduling messages called *events*. Events contain data that are passed between models as a means of communicating throughout the simulation. The diagram below organizes the radar range equation into a series of models that encapsulate the mathematics of the simulation:

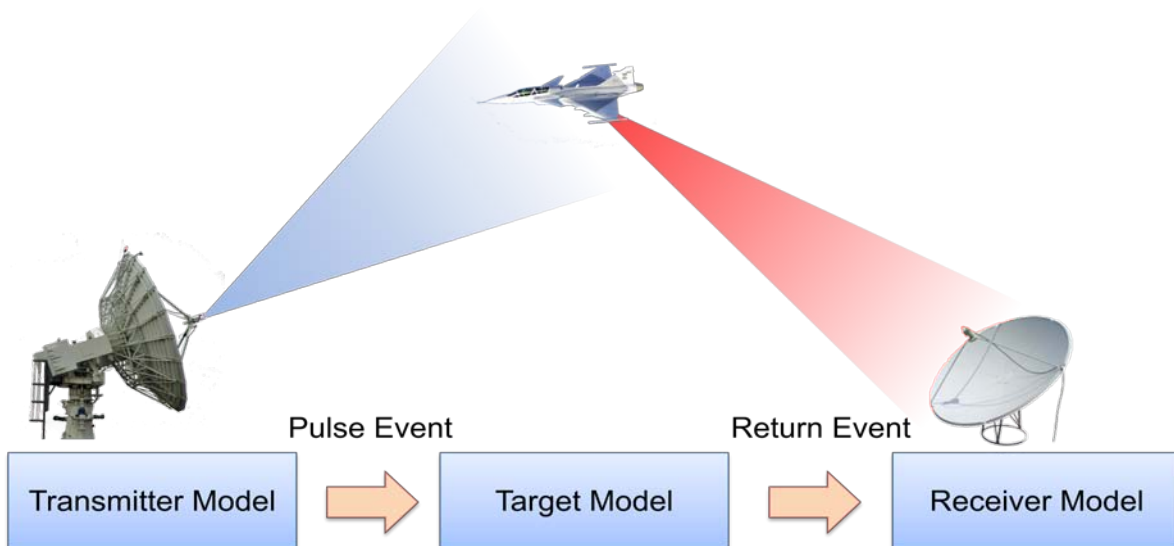


Figure 19 - The radar range equation organized into models and events.

The simulated radar range equation begins with the transmitter model, which possesses attributes such as the current transmit power, as well as the transmission frequency and antenna aperture. The transmitter model contains the physical state of the hardware as well as the logic to calculate pulse attributes such as transmit power and gain. The transmitter model communicates with the rest of the simulator by creating pulse events, which describe the data contained in a single transmitted pulse as it travels across the environment. Pulse events also contain metadata describing the location and orientation of the transmitter from which the pulse originated, which is used later when determining the range to the target.

Pulse events are received and processed by Target models, which represent objects of interest such as missiles or planes being simulated within the environment. Target models contain information regarding their position and trajectory, as well as information on their specific geometry and orientation that affects their radar cross section. Targets use the metadata contained in pulse events in order to determine the properties of the energy return that will be sent to a radar receiver. Hence, targets create *Return* events, which contain the data describing the energy return from the target. Much like pulse events, these return events also contain metadata describing the position of the target so it can be used in later calculations.

At the end of the processing chain, the receiver model contains all of the attributes that describe the physical receiver hardware, such as the antenna aperture and the frequency the receiver is listening for. The receiver model processes *Return* events and calculates the returned energy using the radar range equation. These energy returns are organized by the receiver model into *range gates* based on the time they were received; returns that take a longer amount of time to arrive at the receiver are estimated to be farther away from the transmitter. Hence, a return being received at a certain time implies that it is located a certain distance away from the radar.

4.4.3 The Model Class Hierarchy

In order to contain the previously described models, based on the radar range equation, a model class hierarchy was created to support all models used with the simulator. All models descend from a common *ModelObject* class, which defines the basic functionality that all models must fulfill. Based on the OASIS specification (see Section 2.5.3), a model is an object that is capable of receiving and processing events. This abstract base class provides the foundation for the rest of the models that derive from it, including transmitters, receivers, and targets. The inheritance tree for the model classes is displayed below:

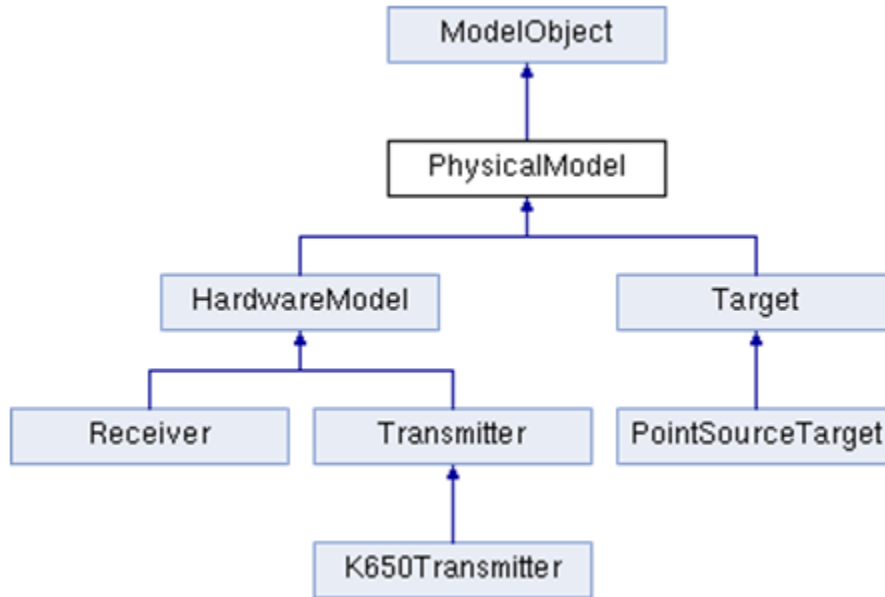


Figure 20 - The inheritance tree for the model classes. *PointSourceTarget* and *K650 Transmitter* are used as examples of classes which could be extended from other general model types.

Directly descending from the *ModelObject* class is the *PhysicalModel* class. This class describes the motion of models throughout simulation through the use of trajectories that define the position of a model at a given time. Radar *HardwareModels* and *Targets* descend from this, since both targets and even the physical radar hardware itself may be moving during the simulation (for example, if the transmitter is on a boat). Transmitters and Receivers both descend from a *HardwareModel* base class,

which contains common information used to describe the orientation of the antennas common to actual transmitters and receivers.

It is important to note that the Receiver, Transmitter, and Target are abstract classes. These classes only define the events that they can process, leaving all other attributes to be defined by subclasses. For example, the Target class only explicitly states that it processes pulse events. The internal behavior of targets outside of the processing of this event is contained within the subclasses of Target.

4.5 Simulation Engine Layer

The simulation engine is the key to the extensibility of DVERT's design. As described in Section 2.5.2, a simulation engine contains the event queue and maintains the current simulation time. The domain-specific models such as Targets, Transmitters, and Receivers interact with the engine by processing and scheduling events. The simulation engine is generic; it contains the fundamental infrastructure required to support an event-based simulation, and has no knowledge of the domain-specific models that use it. The engine can be reused for many different kinds of event-based simulations.

A detailed class diagram of DVERT's simulation engine is located in Appendix D, and more detailed information is located in the DVERT Doxygen documentation. The simulation engine contains a list of all the models in the simulation. The simulation engine allows models to schedule events and subscribe to specific types of events. Models can schedule events on themselves or on other models by specifying their unique id. Models can also schedule events globally, and the engine will broadcast them across the middleware layer to other components and simulation engines. These features of the engine are each encapsulated in their own class, as shown in Figure 21, below.

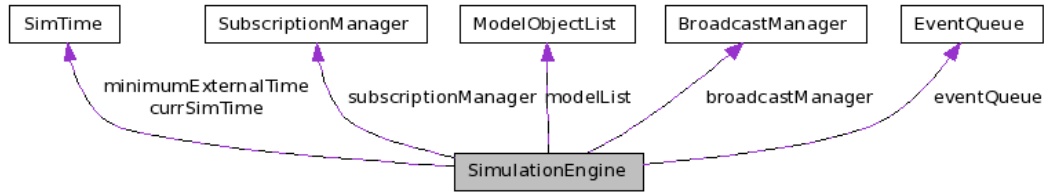


Figure 21: The Simulation Engine

Models do not interact with the simulation engine directly. The interface to the simulation engine is encapsulated in the Model Manager class. Each model object is assigned a common Model Manager, which it uses to interact with the simulation engine. The Model Manager is an example of the façade design pattern. It exposes a specific set of the simulation engine’s functionality to the models.

DVERT’s architecture supports multiple simulation engines within a single distributed simulation. The simulation engines exchange relevant events between each other and synchronize the current simulation time. The middleware layer is the key to the architecture’s flexibility, allowing simulation engines to execute on the same machine or on multiple machines across the network without requiring changes to the code or configuration. The ability to run separate simulation engines on multiple machines makes it possible to scale a simulation to a large number of models by reducing the amount of computation that must be performed on a single machine.

4.5.1 Configuring and using the simulation engine

The first step in running a complete simulation is to configure the simulation scenario, which includes the initial time, a set of models, and any initial events. A scenario is represented by a Configuration Object. The engine must load a Configuration Object before a simulation can begin. When the engine loads a Configuration Object, it adds the models to its model list, adds any initial events to the event queue, and sets the current simulation time to the scenario’s initial time.

The simulation engine’s *run* method begins the simulation and enters the main simulation loop. While in the run state, the simulation engine takes the first event off the event queue, updates the

current simulation time, and determines what models in the model list need to process the event. The engine passes the event to each of these models and waits for them to finish processing the event. The models might schedule new events to place on the event queue. The engine repeats this process until the event queue is empty or until a model requests the termination of the scenario, at which point the simulation is over.

As an example, consider an airport simulation that models airplane arrivals and departures. Airport A has 40 airplanes, with departures every 10 minutes. Airport B has 10 airplanes, with departures every 30 minutes. Airplanes fly between airport A and B and the trip takes 5 minutes each way. Eventually, all of the airplanes will end up at airport B, and the simulation will end.

A simulation for this example would require an airport model, an arrival event, and a departure event. An airport model would have an initial number of planes, a departure rate in minutes, a unique id, and a destination id. Each airport would process an initial departure event. After processing each departure event, the airport would schedule an arrival event at the destination airport 5 minutes in the future. The airport would also schedule a new departure event in the future according to the departure rate. If the airport receives a departure event and has no planes left, the simulation is over.

4.5.2 Event scheduling and subscription

Events represent the interactions between models. To ensure that the simulation can support a wide variety of models and events, the simulation engine uses a robust mechanism to schedule events and distribute them to models for processing. This mechanism is a modified version of the one used in the OASIS simulation specification, developed at MIT Lincoln Laboratory.

The simulation engine supports two separate mechanisms to determine which models need to process any given event. The mechanism to use is determined by the model that schedules the event. Models can schedule an *Event For One* on themselves or on another model by specifying its unique ID.

The engine also allows models to subscribe to specific types of events. If the model does not know the ID of the model(s) that should process the event, it can schedule an *Event For None*, and any models that subscribe to the event's event type will receive it for processing. The *EventForOne* and *EventForNone* classes are decorator classes for domain-specific events and contain the logic to determine what models will need to process them. The OASIS specification refers to these as *managed events*.

4.5.3 Global event broadcasting and time synchronization

The DVERT architecture supports distributed simulations that use multiple simulation engines, and these engines exchange relevant events as they are scheduled. Models can schedule events globally, and the engine will broadcast them across the middleware layer to other components and simulation engines before scheduling an *Event For None* locally. Global events do not require a destination model ID because the scheduling model will be unaware of the models on other simulation engines. Figure 22 shows the global event broadcasting process.

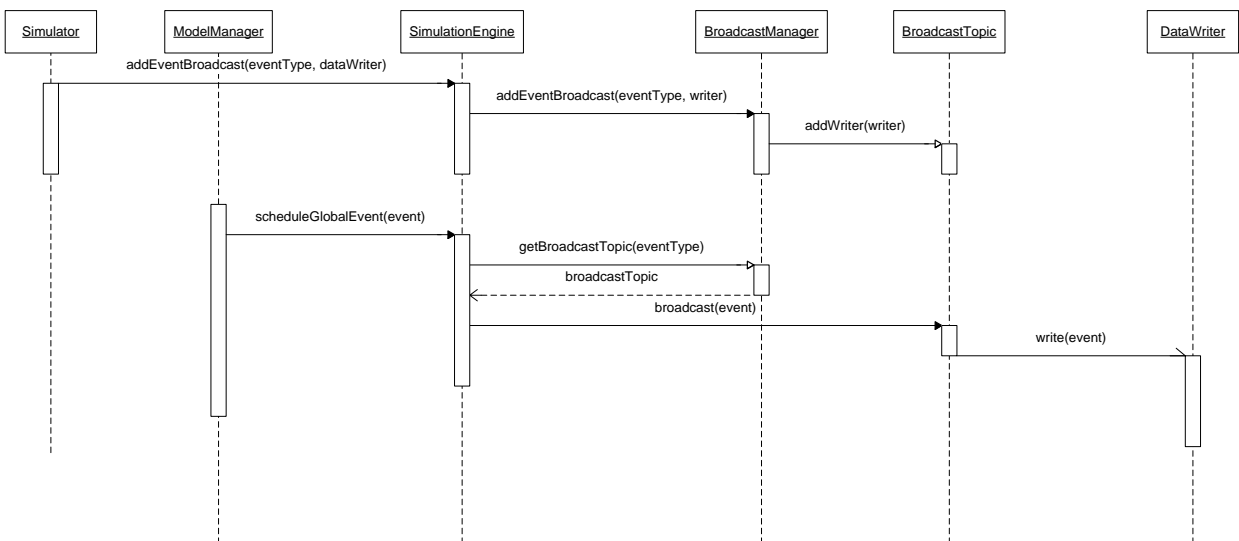


Figure 22 - Event Broadcast Sequence Diagram

The simulation engine's broadcast manager associates each event type with one or more DataWriters. A DataWriter represents the API to the middleware layer and allows the engine to send messages to other components. The DataWriters must be configured before the start of the simulation and registered with the engine. The GlobalEvent base class defines a method to convert the event to a message that can be sent over the middleware. When a model schedules a global event, the engine writes the event to each DataWriter that is registered for the event's type. Any components that are listening for global events can schedule them within their simulation engines. This presents a serious time synchronization problem, because simulation engines maintain their current simulation times separately.

Simulation engines process events in strict time-increasing order. Events can only be scheduled in the future. Each engine in a distributed simulation maintains its current time separately, and these times are usually out of sync. If a simulation engine receives a global event from the past, it would invalidate any processing the simulation engine has done since the time of the event. To solve this problem, the DVERT engine uses a lookahead mechanism to prevent any simulation engine from advancing too far ahead of the other engines in the simulation.

A lookahead is a promise not to schedule any more global events for a certain amount of simulation time [Fujimoto, 2000]. This allows other engines to advance to the current time plus the lookahead. Simulation engines send their current time and a lookahead every time they broadcast an event. Each engine remembers the most recent time of the other engines and does not process any local events beyond the minimum external time plus the lookahead time. Lookahead times must be chosen carefully so that at least one engine will always be able to process the next event. Because the engines do not know about each other at the beginning of the simulation, an initial discovery phase occurs

before the start of the simulation where engines broadcast one of each global message across the middleware.

4.6 Component Layer

In a distributed simulation, each simulation engine executes within a process. All processes in the DVERT architecture extend from a common *component* base class. A component is a process that responds to control messages, allowing users to start and stop the simulation's processes all at once. The DVERT component base class is inspired by the ROSA II component base class and uses a similar state machine, shown in Figure 23.

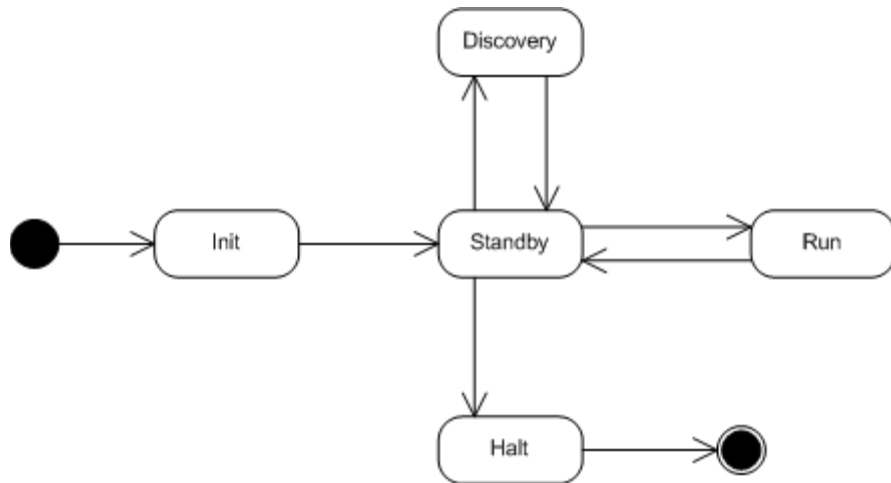


Figure 23: Component State Machine

To create a new component, developers extend the component base class and define functionality for each run state. Each state in the component's state machine corresponds to a function that will be handled by a component subclass. When components are created, the base class initializes the RTCL middleware layer and creates DataReader threads to listen for control messages. Components transition between run states in response to control messages, which might come from a user interface. The component base class contains the state transition logic. The standby state represents a transition between two run states and is where the component base class determines the next state.

Because the component base class initializes the middleware layer, developers don't have to explicitly instantiate RTCL in component subclasses. While in the init state, components perform any initialization required to prepare for the run state, such as creating threads and declaring DataWriters and DataReaders to communicate with other components. The discovery state is used by simulator components, which contain simulation engines. Simulation engines need to discover each other before the simulation begins in order to properly synchronize simulation time (4.5.3). The run state represents the component's main loop, and components remain in the run state until they release control or receive a halt command. The halt state allows components to perform any final actions before terminating, such as closing files and joining with threads.

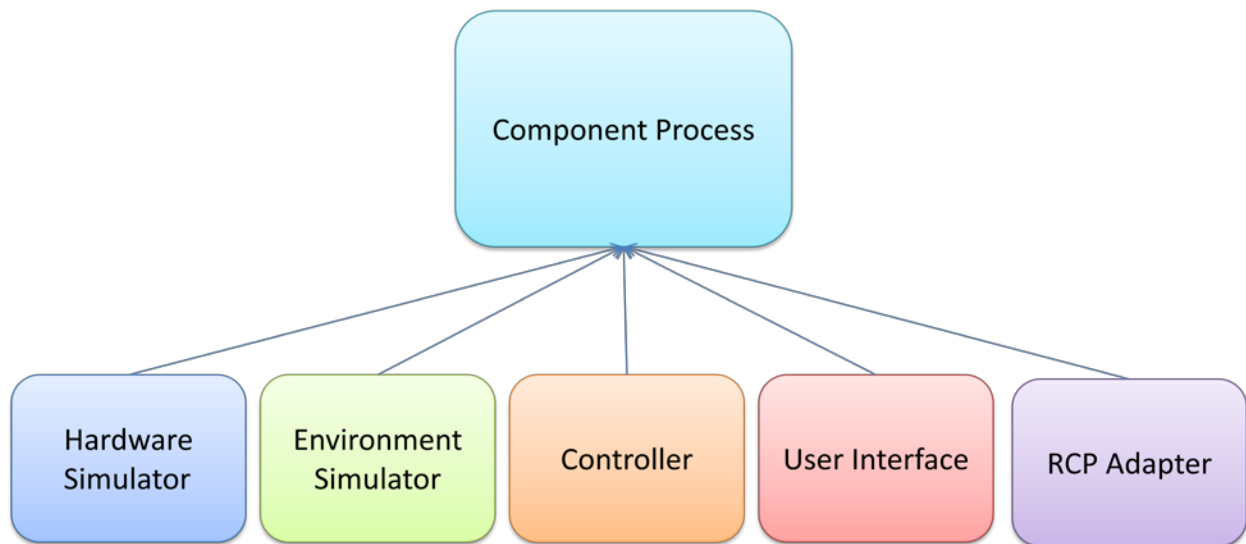


Figure 24 - Component and its Subclasses

Figure 24 shows the components that make up the DVERT distributed simulation. These components work together to form a complete radar simulator that can interface with the Radar Control Program and be controlled through a user interface.

4.7 Middleware Layer

The DVERT simulation architecture uses the ROSA Thin Communications Layer (RTCL) for inter-process communication. RTCL is described in section 2.3. Components do not communicate directly with other components. Components read and write messages through the middleware layer using the DataWriter and DataReader classes. The middleware layer uses a publish/subscribe messaging paradigm (2.3). In order to communicate, components only need to agree on a topic name for each type of message. Because publishers don't need to know about their subscribers, components are decoupled from each other. For example, the hardware simulator component does not need to know how many target simulator components it is communicating with. Target simulators can be added as needed to scale to additional targets, and the hardware simulator does not need to be changed.

The middleware layer encapsulates the transport layer used to distribute messages between publishers and subscribers. Components can communicate without needing to know whether they are on the same machine or on different machines connected by a network. As a result, the simulation architecture is flexible. Hardware and environment simulators can execute on the same or separate machines without changing any code or configuration.

4.8 Scalability of the Design

The key limitation of the simulator that this project seeks to address is scalability. The layered architecture described above makes distributed simulations possible. Components allow simulation engines to execute on separate machines and the middleware layer allows them to communicate. The primary bottleneck in radar simulation is that the simulator must perform complex computations for each target that is hit by a pulse in order to compute the energy and shape of the returns. The DVERT architecture and models are optimized to distribute this computation across as many machines and processors as possible.

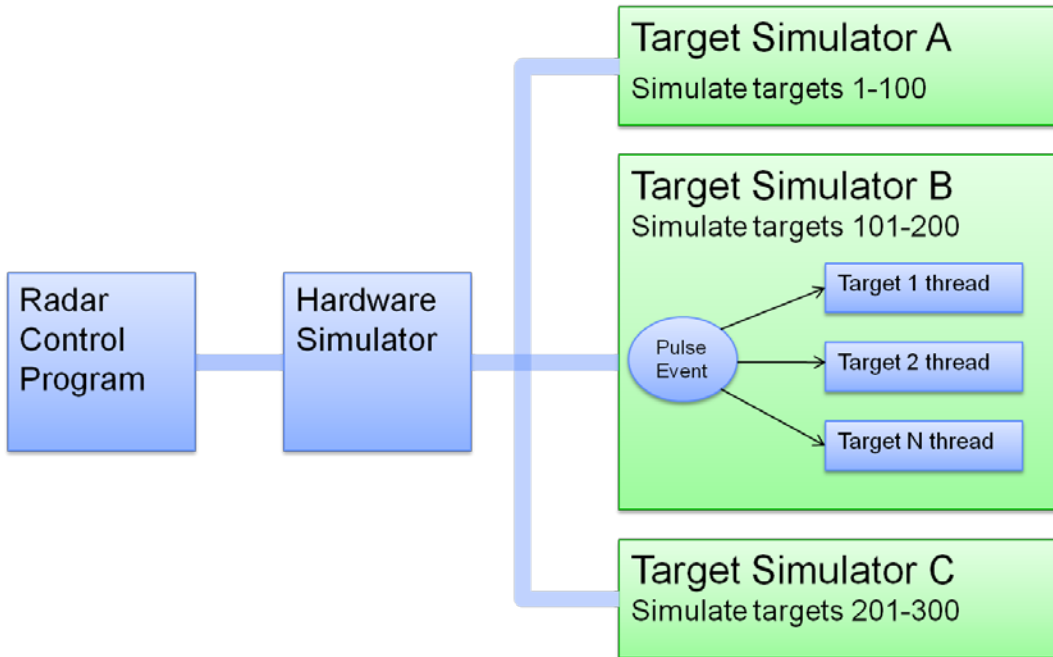


Figure 25 - Scalability of the DVERT simulator

In order to scale to a large number of targets, the DVERT architecture supports multiple target simulators running on separate machines. A large number of targets can be divided among the target simulators, reducing the amount of work that must be done by each computer. Figure 25 shows a simulation with 300 targets and three target simulators. Each target simulator runs on its own machine and is responsible for 100 targets, or a third of the total work. The hardware simulator contains the transmitter and receiver models and sends each pulse event to the target simulators.

All target models subscribe to pulse events through the simulation engine, as described in Section 4.5.2. When the simulation engine processes a pulse event, it must pass it to each target for processing. This process can be parallelized on computers with multiple cores or processors.

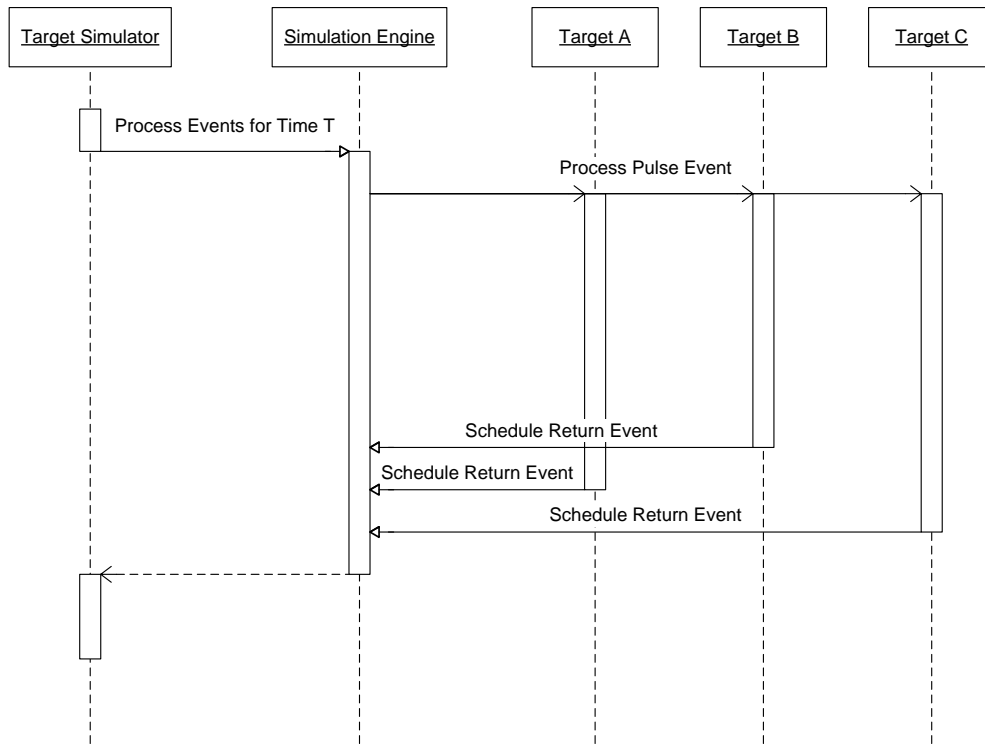


Figure 26 - Parallel event processing in the simulation engine

Figure 26 is a sequence diagram that shows how pulse events might be processed by targets in parallel. The engine assigns a thread to each target. In order to reduce the total amount of time required to process the pulse event, the targets process the pulse event in parallel within their own threads. The engine must wait for each target thread to finish processing the pulse before it can continue and advance simulation time. In order to reduce the overhead of creating and destroying threads, the engine can assign threads from a thread pool that can be as large as the maximum number of threads the operating system supports for a single process.

It is important to note that the engine has no knowledge of the specific types of models and events it manages. The parallel event processing optimization described above is possible for any event

that has multiple subscribers. The event will be processed by each subscriber at the same simulation time. In an event-based simulation, any events that share the same simulation time will not affect each other and can be processed in parallel. Due to time constraints, this feature of the simulation engine was not implemented, and is further described in Section 6.1.2.

When a single pulse hits N targets, it creates N returns. This could produce a large number of return messages in a simulation with thousands of targets. In order to reduce the number of messages sent across the network to the hardware simulator, a copy of the receiver model is located on each target simulator. The receiver collects local return events and adds them up. Each receiver then contains a subset of the total returns, which it sends to the aggregate receiver on the hardware simulator. The aggregate receiver sends the final result back the radar control program adapter by scheduling a global PulseAux event.

The distribution of targets across multiple machines reduces the amount of work each computer has to do. The parallel processing of pulse events by targets reduces the amount of work each processor or core must complete, reducing the overall time required to process a pulse. These factors, combined with the efficient message passing between simulation engines allows the design to scale to a large number of targets.

5 Results and Analysis

The results of the project are described in terms of the original metrics for success defined for the project (see Section 1.4). These included the creation of an extensible design that would pass Laboratory review, prototyping a minimal implementation to validate this design, testing to achieve 80% code coverage, and providing adequate documentation on how to use and extend the simulator. A summary of the final state of the project and a review of the project deliverables are discussed in the following sections.

5.1 Design Review

The primary deliverable for this project was a scalable and extensible simulator design that will meet the needs of the sponsor for many years into the future. The design, detailed within section 4, was iterated on over the course of seven weeks and was regularly updated to incorporate new domain knowledge and new use cases provided by the sponsor. Weekly design meetings were performed with radar and simulator experts at the Laboratory in order to produce an efficient simulation and an accurate representation of the radar domain. In order to validate the extensibility of this design, and, as a metric of success for the project, the team held a final design review with radar experts at the Laboratory.

The design review was conducted to verify that the simulator design supported all major use cases required by the Laboratory and that it would be extensible enough to support future radar simulation needs within the group. Eight members of the Ranges and Test Beds Group, including simulation and radar experts, attended the design review. The team presented the simulator design and received a variety of feedback from the Laboratory staff, which is detailed in Appendix C. The project group presented examples of how the radar hardware models could be extended to support different types of radar, such as spinning-dish, bi-static, and phased-array systems. The project team emphasized

the extensibility of the target base class, and described how the trajectory and radar cross section classes encapsulate important information about targets and make the design flexible.

Laboratory staff raised several minor concerns about the flexibility of the models in the design. The first issue was that the radar cross section of a target might need to be calculated at the receiver, and not at the target as in the design. Calculating the RCS at the target is acceptable for most simulations, but high-fidelity RCS models depend on the position and orientation of the receiver. Another minor design concern was the use of a low-fidelity pulse model that included only an average power and a width. Higher-fidelity pulse waveforms vary in time and have changing power and frequencies. The Laboratory staff was confident that the design would be flexible enough to support higher-fidelity pulse models.

Clutter and noise modeling within the simulator was also a point of interest. Members of the group were curious whether the simulator design could support clutter and noise models to interfere with radar returns at the receiver. While no actual clutter or noise models were provided in the simulator design, it was noted that the simulator could simply use additional targets in order to accurately model clutter from the environment. Noise could be modeled easily at the receiver by its environment class.

The engine received no criticism during the design review. This is largely because it was designed to be as generic as possible and, as a result, did not directly relate to domain-specific issues that were more likely sources of criticism.

Upon completion of the design review, the project group revised the simulator design in order to address the concerns and suggestions presented by the Laboratory staff. The revised design featured abstract transmitter and receiver base classes, and the team implemented specific transmitter and receiver subclasses to allow for more realistic pulse waveform models. The radar cross section

calculation remained in the target model, but the team verified that the design could support RCS calculations at the receiver by passing the target's RCS object to the receiver within the return event. Upon making these changes, the design passed Lincoln Laboratory review.

5.2 Implementation Results

The second major deliverable for the project was a working radar simulator. The initial goal of the implementation was to simulate 1000 point-source targets distributed across multiple machines. The simulation was intended to be multi-threaded to take advantage of multiple processors and cores. Finally, this implementation was intended to integrate directly with a ROSA radar control program, allowing the team to validate its output against the output of the simulator.

As the project progressed over 9 weeks, the project group determined that the scope of these original objectives was too large to accomplish within the given timeframe. However, the group was able to develop a stable implementation that satisfied a number of these criteria.

The majority of the work for the implementation involved building the necessary infrastructure required to create a distributed event-driven radar simulation. The team implemented the entire design for the distributed simulation engine (see Section 4.5), which supports a generic set of models and events interacting across multiple machines. The simulation engine supports configurable scenarios and can dynamically load a set of models and initial events. The team also implemented the component base class, which all the processes in the simulation extend from. The team developed a user interface component to allow users to start and stop the simulation. The team also used a graphical display to help visualize the simulator's output.

Using this simulation infrastructure, the team implemented the class hierarchy required to support a diverse set of radar hardware and target models, effectively creating a framework for developers at the Laboratory to extend using high-fidelity models in the future. The team implemented

an example K650Transmitter and K650Receiver model according to parameters from real transmitter and receiver hardware. The team created a constant radar cross section model and several trajectory models including fixed-point targets, linear motion, and oscillating motion back and forth between two points. The team successfully integrated each step of the radar range equation within the transmitter, receiver, and target models, producing a functional radar simulator.

The final implementation delivered to Lincoln Laboratory was capable of simulating multiple targets in a distributed, multi-process setting. The team tested the implementation using multiple environment simulators and multiple targets, and it consistently produced accurate returns. The team used a common simulator configuration, or scenario, for each acceptance test. The scenario is depicted in Figure 27, below.

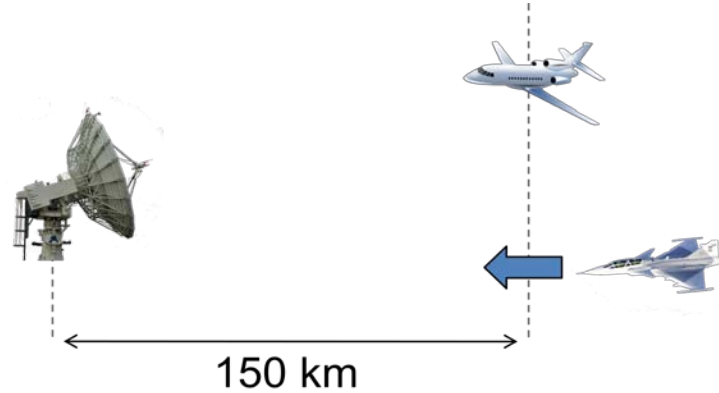


Figure 27- The common acceptance test used in testing the implementation. A stationary and a moving target are used to test the returns from a single transmitter.

The scenario features a pedestal with a single transmitter and receiver and two target models, one at a range of 150 km and one moving from a range of 200km to a range of 100km. One target is stationary, and the other is moving towards the radar. The two targets have constant, fixed radar cross-sections which are equivalent to each other. A target range of 150 km corresponds to a round-trip travel

time of 1000 microseconds per pulse. This made timing issues easier to debug and simplified the configuration of the receiver, which listens for returns for a short period of time starting exactly 1000 microseconds after a pulse is transmitted. This acceptance test was used to validate the successful operation of the simulator using multiple targets and multiple target simulators. The output produced by using this acceptance test was visualized using a real-time graphing tool to illustrate the returns. The images below show the progression of the returns as time passes.

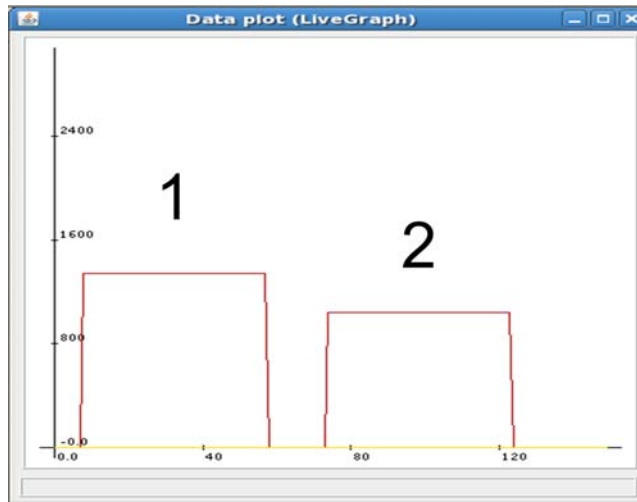


Figure 28 - The energy returns from two targets (joules) versus the simulation time (μ s).

Figure 28 displays the energy returns from two targets as seen by the radar receiver. The horizontal axis represents time, and the vertical axis indicates the amount of energy returned. The receive window, or the horizontal axis, begins at 1000 microseconds after the pulse is transmitted and lasts for 150 microseconds; this means the receiver is looking for targets at a range of approximately 150 to 170 km. Target 1 is stationary, at a distance of 150 km from the transmitter. Target 2 is moving towards the transmitter, but is currently at a farther distance and produces a lower energy return. This is the expected behavior of a radar return from these two targets, as the energy returned is inversely proportional to the target range (see Figure 6). Pulses sent by the transmitter are 50 microseconds wide,

and the returns have this same width. The returns from targets 1 and 2 arrive within approximately 100 microseconds of each other, indicating that the targets are at almost the same range from the radar.

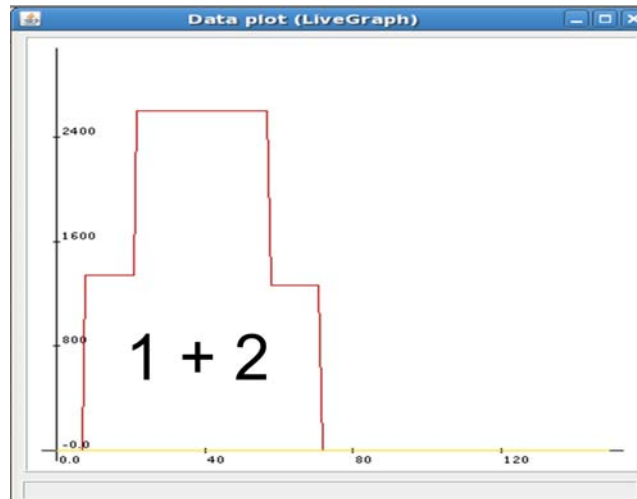


Figure 29 – The energy returns from the acceptance test after several seconds of elapsed time.

As time passes, the moving Target 2 eventually passes by Target 1, shown in Figure 29. A larger amount of energy is returned during the time that they overlap; this is the expected behavior of a radar return from two targets in close proximity. The simulator’s output matched the project group’s expectations for the behavior of an actual radar system. Additionally, they matched the expected behavior when analyzed by a radar expert from the group.

The same testing scenario was performed using one hardware simulator and two separate environment simulators with identical configurations. The return energy in this scenario was twice the energy returned in tests with only one environment simulator. This is the expected result, because the second simulation contained two of each target.

Due to time constraints, the team was not able to fully implement the simulator’s design. All tests were performed using shared memory as an RTCL middleware as opposed to a simulation over the network using DDS. This was due to difficulty configuring DDS to support large data loads over the

network. Due to time constraints, the team was not able to integrate the simulator with the ROSA radar control program (RCP). For testing purposes, the team implemented an adapter program to simulate control messages from the radar control program. The team was also unable to benchmark the real-time performance of the simulator. Finally, the team was unable to implement the entire multithreaded simulation engine design within the timeframe of the project.

Although the team was unable to incorporate all the features of the design in the implementation, the acceptance tests proved that the simulator was capable of a distributed radar simulation. The prototype radar simulator provided a valuable proof-of-concept for the design and is capable of supporting a diverse set of hardware and target models. The simulator will serve as a code base for Laboratory developers to extend in the future.

5.3 Testing Results

The third metric for the success of the project was a test suite containing unit tests that amount to at least 80% code coverage for all model and engine code. In order to accomplish this, the team used the CxxTest unit testing framework. The team wrote unit tests to test the individual parts of the engine and each model. The team also developed integration tests to ensure the engine and models interacted correctly.

























Name	Function ... ▲	Uncover...	Condition/...	Uncover...
rosa_interface_compon...	41% 	10 	23% 	23 
environment_simulator	70% 	3 	66% 	2 
hardware_simulator	70% 	3 	66% 	2 
libmodels	79% 	44 	83% 	17 
libengine	84% 	19 	80% 	10 
test	89% 	15 	98% 	1 

Figure 30- Project code coverage numbers provided by Bullseye Coverage

As shown in Figure 30, the team was able to achieve approximately 80% code coverage in the engine and models through a combination of unit tests and integration tests. The team developed unit

tests to verify the expected behavior of the transmitter, receiver, and target models and related classes including the environment, radar cross section, and trajectory classes. These tests allowed the team to refactor code within the model and engine classes and quickly determine if the classes were still functioning correctly. If changes caused a test to fail, it became immediately obvious that debugging was necessary. To make integration tests possible for the models, the team developed a mock object for the model manager class, which represents the interface between the models and the simulation engine. The mock model manager intercepts events when they are scheduled by models and allows unit tests to pass them directly to other models. Several unit tests used the mock model manager to pass events between the transmitter, receiver, and target models, ensuring that they integrated correctly.

The main integration test for the engine was an airport simulator modeling airplane arrivals and departures. The airport simulator is described in Section 4.5.1. The airport simulator tested the majority of the engine code and verified that the models were able to schedule and process events in the correct order.

In addition to unit tests and integration tests, the group also performed performance tests to ensure that the simulator contained no memory leaks and was capable of running for a long period of time without issues. The team ran the simulator overnight for a period of approximately 16 hours and found it was stable and was delivering the proper output. The size of the event queue in each simulation engine remained approximately constant throughout the simulation, showing that the simulator had no problem keeping up with control messages from the radar control program adapter. This also suggests that the memory used by the simulator is stable over long periods of time and that memory is being properly freed when it is no longer in use.

Including all mentioned tests, the code coverage of the existing model and engine code was measured to be greater than 80%.

5.4 Documentation

The final major deliverable for the project was the creation of appropriate documentation to accompany the project design and implementation. This documentation should be sufficient enough so that someone who is completely unfamiliar with the simulator can get it set up and installed, as well as extend the existing design and implementation to create scenarios specific to their simulation needs. Documentation was defined as one of the most important parts of the project, because the use of the simulator after the completion of the project is largely defined by how easily it can be understood and adopted by Laboratory staff.

As a response to this need, emphasis was placed on several documentation efforts for the project. The first of these was the use of Doxygen (see section 3.1.8) comments in code, which provide relevant information for those viewing the actual implementation code, as well as the ability to automatically generate documentation for the each class in the project. This generated Doxygen provides useful information such as descriptions of functions and data members, as well as diagrams and visuals showing the overall design of the project. Doxygen comments were inserted into every relevant file within the model and engine classes to ensure that future developers would be able to understand the inner workings of the simulator.

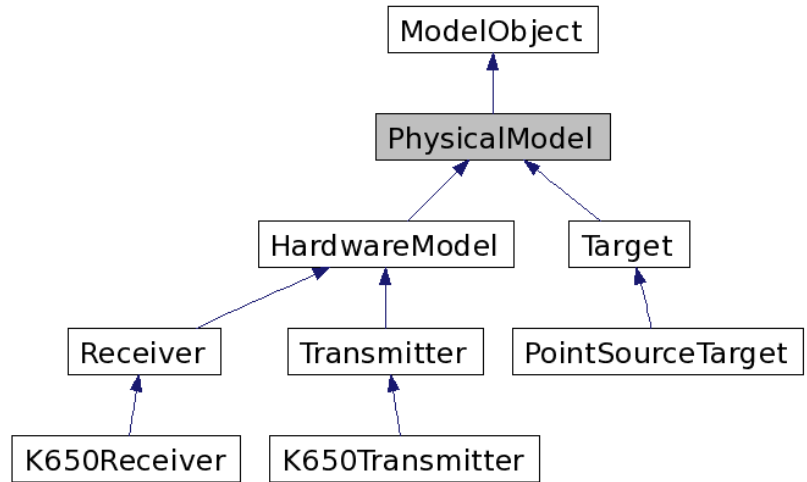


Figure 31 - A sample doxygen class diagram

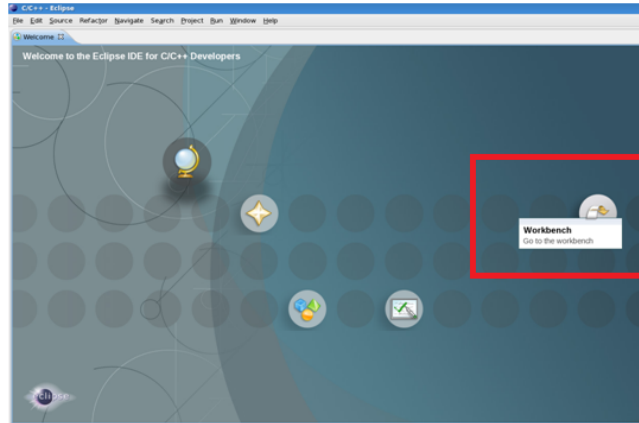
The second major documentation effort for the project was the creation of a simulator *User's Guide*, which detailed how to get the simulator installed and running. This includes a detailed list of the dependencies required for the project and how to acquire them, along with how to acquire the simulator package and get it running both within and outside of Eclipse. By the end of the guide, readers should be able to fully build and run the simulator.

Once you have Java, simply execute the file to start using Eclipse:

```
./eclipse
```

You will then be presented with a dialog which says to choose the location for your workspace. Enter a directory, press ok, and then you will be brought to the Eclipse main screen:

You'll be greeted by a start screen. Choose the option which says "Go to the workbench" to start.



From there, you'll enter the main Eclipse editing screen. You'll need to install the Eclipse SVN plugin "Subclipse" in order to download the simulator code from the repository. You can do this by going to:

Help->Install New Software

A dialog will appear. Click the *Add* button in order to add a new software site to download from.

Another dialog will appear. Fill it in with the following information:

Figure 32- A sample page from the User's guide, walking through the installation of the simulator

The third major piece of documentation associated with the project is a second project guide referred to as the *Developer's Guide*. This guide is written for those who already have the simulator up and running and are looking to extend the existing design and implementation to create their own custom models. It provides a sample list of use-cases and describes how the current simulator could be extended to support them. Examples include different types of radar hardware, more advanced targets, and different radar cross sections. The goal of the *Developer's Guide* is to help developers create new models within the simulator so that they can support new simulation scenarios to fit their specific needs.

5.5 Summary of Results

Over the course of nine weeks, the project team successfully met a large number of the metrics for success originally defined for the project. A thoughtful, extensible design which passed a design

review held with LL staff was developed. A working implementation to validate the fact that the design could be distributed and to confirm the correct behavior of basic model classes was created. A comprehensive testing suite, which acquired the metric of 80% code coverage originally stated for the project was developed. Finally, a large collection of documentation to ease the technical transfer of the project to Lincoln Laboratory and promote its future use was written.

6 Conclusion

Over the duration of the nine weeks allotted to the project, the project team leveraged expertise and resources from the Laboratory in order to accomplish the goal of designing a distributed radar simulator. Through the use of iterative design and collaboration with Laboratory experts, the team was able to design a set of domain-specific radar models and a robust, generic simulation engine. While the prototype simulator implementation had to be scoped down from its initial goal of supporting a large number of targets, it still served to validate the distributed design of the new simulator and prove that radar computation could be performed over multiple processes. The outstanding issues remaining with the project are detailed in the following section.

6.1 Outstanding issues

These issues represent the functionality defined for DVERT at the onset of the project, but which did not make it into the delivered version. As the project evolved, it became evident that some of these features were outside the scope of this project. Integration with the radar control program proved to be a much larger problem than originally scoped and was not directly relevant to the development of the simulator framework. Multi-threading was an optimization that, while desired, was not immediately necessary to produce a working prototype. Timing and synchronization of global events are already implemented within DVERT, but they still require slight refactoring. Finally, middleware configuration for network transport represents a minor task that was simply considered non-critical to development.

6.1.1 ROSA Interface

The original plan was to implement DVERT based around the same external interfaces used by the legacy simulator. This plan would enable DVERT to be used interchangeably with the simulator. Thus, DVERT would be able to take advantage of all the pre-existing components developed as part of the ROSA II framework, such as an assortment of GUIs for display and control. Additionally, directly

interfacing with the actual real-time control software would enable DVERT's performance to be examined in response to real-world inputs and deadlines.

As more research was conducted into the transport methods used to interface the simulator with the radar control software, it became evident that implementing the same interface would be outside the scope of the project. While ROSA II components communicate over RTCL, the simulator was developed before the development of the ROSA II framework. As such, a separate UDP based interface was developed as a ROSA II component to bridge the gap between the simulator and the rest of the radar control software. The Laboratory also indicated a desire to discard this interface in the future in favor of communicating directly over RTCL. However, the ROSA II components were still written to listen over the UDP interface.

To complicate matters further, the data structures being passed around had evolved over the course of the simulator's development. As the simulator was adapted to suit new projects involving hardware with new capabilities, additional fields were simply added to the existing data structures. This bloat has caused the data structures to balloon to many times their necessary size with dozens of fields that are specific to given hardware models.

A decision had to be made. On one hand, the UDP interface could be implemented, enabling the use of the existing components. However, the drawback was that the work would largely be wasted as the interface would be discarded in the future. Conversely, a direct interface to RTCL could be implemented, with the tradeoff that the existing control software wouldn't immediately support the interface.

After analyzing both options, a solution emerged. A Radar Control Program (RCP) adapter component was written for DVERT. This component acted as a mock external system, generating control messages as input to DVERT and listening for its output. This approach eliminated the need to

implement the interface between DVERT and the existing radar control software, yet still enabled the testing of DVERT's functionality. The RCP adapter sends simplified versions of real control message data structures to the simulator. The RCP adapter component was written such that the automatic generation of control messages could be replaced with either an RTCL or UDP listener to interface with the Laboratory's radar control software.

6.1.2 Multi-threading

While the current implementation takes advantage of running across multiple processes, the fact that each process contains a single thread prevents it from realizing the full parallelizability of a multi-core machine. Ideally, the simulation engine would process all of the events at a given simulation time in parallel. This process is described in section 4.8. However, this optimization introduces a number of difficulties within the engine as there are two key critical sections.

The first critical section is the event queue. If two threads try to push events to the queue simultaneously, the behavior would currently be undefined, potentially resulting in one of the events being dropped or the simulation crashing as a whole. A thread-safe implementation would ensure that only one thread would be capable of pushing or popping the queue, ensuring that all threads have access to a consistent copy of it.

The second critical section is the model manager. As a façade for the simulation engine, the model manager represents all of the resources of the simulation engine that a model could potentially access. Rather than individually making the various managers thread-safe, the same effect could be accomplished by modifying only the model manager.

Within each simulator process, there are separate threads for the execution of the main loop and for each of the RTCL subscribers that listen for messages from the middleware. This enables the simulator to continually execute its main loop without having to wait on incoming data. As previously

mentioned, the event queue is not currently thread-safe, so two subscribers could potentially write to it simultaneously, resulting in undefined behavior.

6.1.3 Timing and Synchronization

The timing system used within DVERT also remains to be finalized. While the final delivered product works, there are still border cases that remain to be debugged. These border cases could be made more visible and thus more easily addressable through the use of extensive unit and integration tests. While the majority of the engine is thoroughly tested for robustness, the timing system lacks the same level of coverage.

More importantly, the issue of deadlock and deadlock recovery was not addressed within the simulator. Referring back to section 4.5.3, if all of the simulation engines are unable to process an event, then the simulation has deadlocked. If a simulation engine is unable to process an event, it is unable to update the other simulation engines as to its current time. This inability to communicate causes the other simulation engines to be unable to process their events, causing a circular dependency. Currently, this problem is avoided through the careful choice of event timings within the implemented test scenarios.

A variety of different solutions exist for this problem. For example, in Parallel and Distributed Systems [Fujimoto, 2000], Fujimoto describes a deadlock detection and recovery algorithm. The detection algorithm revolves around constructing a tree. When simulation engines receive an event, they add themselves to the tree. If they become blocked and are a leaf node, they remove themselves from the tree and signal their parent. The simulation will have deadlocked when the controller node that sent the first event is the only remaining node in the tree. Once deadlock is detected, it can be

broken by processing the event with the smallest time stamp in the entire simulation¹. This time can be determined by having the controller request the smallest time stamp from each simulation engine and then broadcasting the smallest time it receives back to all of the engines.

6.1.4 DDS Configuration

Minor work is still required in the area of configuring RTCL to properly communicate over a network. Throughout development and testing, shared memory was used as the inter-process communication (IPC) middleware within RTCL. As all the processes ran on a single machine during development, shared memory was particularly well-suited for testing. However, once development was completed, there was a desire to test DVERT across multiple machines on a local network. Naturally, this test would require the use of a different middleware implementation, as shared memory only works for communicating between processes on a single machine. DDS was chosen as the intended networking middleware as it's both real-time and already supported by RTCL.

6.2 Future work

This section consists of features that were not originally part of the overall scope for the project, but which proved to be extremely attractive as optional features that could be implemented after the conclusion of the project.

6.2.1 Load balancing

A significant extension to this project would be load balancing of computation across the simulators of the distributed system. At the moment, targets are statically instantiated on each environment simulator at the start of the simulation and they remain on their original machines until the end of the simulation. Ideally, each machine in the simulation would be responsible for processing an equal number of targets. In this situation, processing would remain evenly spread across the

¹ This event is safe to process because if the simulation was a sequential simulation, the event with the smallest time stamp would be processed next.

machines, netting the largest gain in performance. In the worst case scenario, the majority of targets that require processing could reside on a single machine. Now, one simulation engine may be overtaxed, causing it to miss its deadlines, while others are lying dormant.

One method to prevent a situation like this from occurring is to distribute targets in a strategic manner before the start of a simulation. Targets can be broken up in a number of different ways to evenly distribute the computation of returns. For example, targets can be broken up by sector, range, or other parameters in order to distribute them evenly across a number of simulators. The diagram below illustrates some different types of static initial distributions:

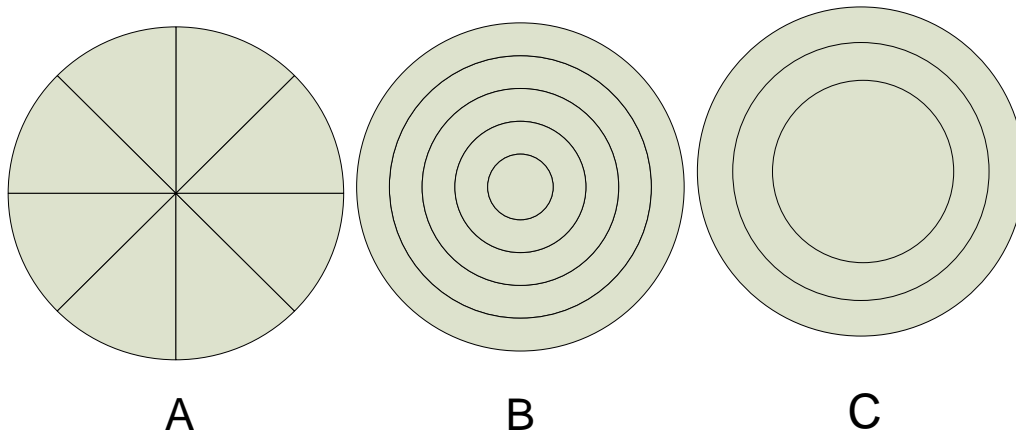


Figure 33 - Distributing target computation by sector (A), target range (B), or volume (C)

Another solution to this problem is the use of dynamic load balancing during the simulation. Its implementation would consist of a two stage process. The first stage is a detection stage. Before the system can begin balancing, it must be informed of the need to balance. Detection could occur through the publishing of error message by simulation components when they start to consistently miss deadlines. Similarly, the system could track the average number of models being processed at recent time steps and then check to ensure that no one machine has a higher than average number of targets.

Upon detecting the need for a rebalancing, transportation events could be scheduled on the underperforming engines. These transportation events would consist of serializing the object to be transferred and then scheduling an event to simultaneously remove it from the simulation engine on which it originated and then add it to the desired simulation engine.

6.2.2 Configuration Objects

Improving the method of configuring scenarios would greatly increase the value and usability of the simulator. At the moment, an interface, referred to as a configuration object, exists for initializing models within simulation engines. A configuration object essentially acts as a container for a collection of models that can be loaded into a simulation engine. Configuration objects are currently hardcoded into each specific simulation component and are passed into the component's simulation engine upon its creation. As a result, modifying a scenario requires that its host component be recompiled. Ideally, the simulator components could be distributed as executable binaries that reference an external file to generate their configuration objects at run-time. Run-time configuration would enable the rapid prototyping of simulation scenarios. Additionally, the simulator could be distributed as a binary with a collection of swappable XML files, thus eliminating time wasted recompiling. The project team briefly researched the feasibility of parsing XML files to instantiate models. Unfortunately, C++ lacks a crucial feature: built-in type reflection. Reflection would have enabled objects to be instantiated without hard-coding functions for each class that converted from some external schema to the internal class type.

6.2.3 Status Messages

Our implementation currently lacks any inter-process communication regarding the state of the components currently active in the system. Individual components report their state locally when changing states, but provide no information to the system as a whole regarding their status. This logging should be moved from a simple print statement to a separate status topic that is communicated over

RTCL. Hooks for this topic were implemented within the component base class to support this feature; however the simulator currently does not populate the information within that topic. Beyond the information contained within the base component, specifically its state, this status topic could also be used to communicate data that is specific to derived components. For example, components with active simulation engines could report statistics such as event queue size and information regarding the performance of their data writers and readers. By providing this information, the health of the system would be much more visible to operators, enabling at-a-glance updates of the system's status.

6.2.4 Simulator components as ROSA II components

The Laboratory also expressed interest in modifying the simulator components to be fully-fledged ROSA II components. Each ROSA II component has a finite state machine whose state can be changed by control messages passed over RTCL. Additionally, each ROSA II component publishes status messages about itself so other components can be aware of the state of the system. DVERT already makes use of a component base class that is functionally very similar to the ROSA II component base class. The DVERT component base class uses a state machine similar to the one contained in ROSA II components. With some minor refactoring, the DVERT component base class could be replaced with the ROSA II component base class. This refactoring would enable the simulator to be configured, launched, and controlled as a ROSA II component. While not necessary, it represents another step into fully integrating into the ROSA II framework.

6.2.5 Graphical User Interface

There are two separate aspects of the system that would benefit from the development of GUIs. The first facet is control and monitoring. Currently, the simulator is controlled via a text-based user interface. This user interface can initialize simulator components and start and stop the components. A very basic GUI, as seen in Figure 34, was implemented on top of the same interface used by the text-

based version. Assuming that status messages were implemented, this same GUI would ideally support the display of the information contained in the status messages. Having a central location for controlling the components of a simulator and for monitoring the health of the overall system would aid in both the use and development of the simulator.

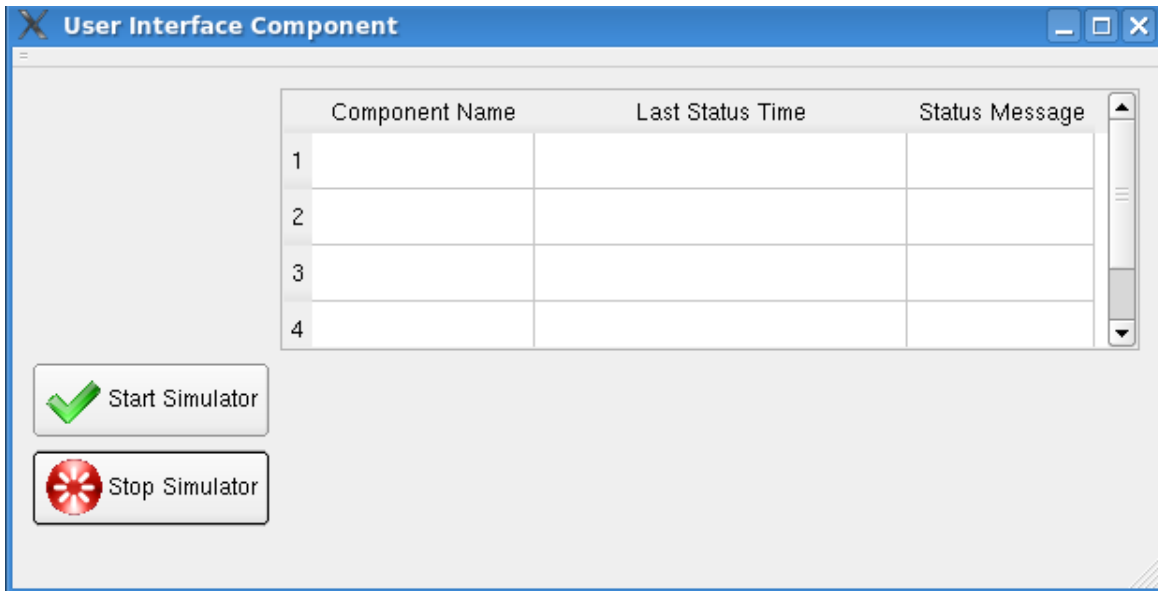


Figure 34 – A Basic simulator Control GUI

The second area where a GUI would be particularly applicable is the display of the return energy. If ROSA II integration were completed, development of a GUI for the display of output may be unnecessary as the existing displays could then be leveraged. However, as they are legacy displays, there may still be a desire to modernize them. Barring that integration, the need for a GUI drastically increases. To address this need during the project, a 3rd party tool was leveraged to graph the simulator's output in real-time. While this tool met the immediate needs of the project, a long term solution may include fully integrating the tool into a display component or potentially coding a display component from the ground up.

6.3 Concluding Thoughts

At the advent of this project, the project team had no experience with either general simulation or radar systems. However, by working closely with Lincoln Laboratory staff, the team was able to leverage the collective expertise of the Laboratory in order to design and implement a sophisticated simulator architecture. In nine weeks, the project team acquired enough domain-specific knowledge in order to produce an extensible, distributed simulator design as well as an implementation to validate this design.

Appendix A. Current ROSA Simulator Feature Tree

1. Target simulation
 - a. Up to 64 targets (on some computers); there is a desire in the group to simulate many more.
 - b. Characteristics are defined relative to a simulation start (liftoff) time.
 - c. Characteristics are defined prior to simulation start, currently via configuration file.
 - d. Center-of-mass translational motion models:
 - i. Trajectory models:
 1. Fixed point.
 2. Ballistic (including atmospheric drag).
 3. Time-tagged polynomial state vector (ECR or local RAE).
 - ii. Duration options:
 1. One-shot: target characteristics become valid at the start time and go invalid at the stop time.
 2. Repeating: translational motion model starts over when it reaches the stop time; example: a plane flying in a closed loop.
 - e. Inertial rotational motion about center of mass.
 - f. Multiple scattering centers.
 - g. Cross section models may vary with time.
 - i. Constant RCS.
 - ii. Gaussian: specify mean and std. dev.
 - iii. Swerling.
2. Environment simulation.
 - a. Refraction: multiple models in use; which one to use is configurable.
 - b. Atmospheric attenuation.
 - c. A low-fidelity clutter model exists but is rarely used.
 - d. An equally rudimentary surface clutter model exists and is also hardly ever used.
 - e. Multipath is modeled via a complex reflection coefficient, but time delays are ignored.
3. Hardware simulations are responsive to commands from the radar-control program and report their current state via entries in the AUX data.
 - a. Pedestal simulation responds to pointing commands
 - i. Elevation over azimuth mount.
 - ii. Uses position feedback loop for coarse pointing control and rate feedback for fine control.

- iii. Configurable position, rate, and acceleration limits.
 - iv. Configurable encoder lsb.
 - b. Antenna
 - i. Two model choices:
 1. Parabolic dish antenna with configurable multi-horn monopulse feed.
 2. Phased array with constant, predefined amplitude taper and subarray definition.
 - ii. Configurable gain.
 - c. The Master Timing System (MTS) used to be an actual system that provided hard triggers to the hardware, but is now just a collection of algorithms that runs on the control system for each piece of hardware. Because of the route followed in getting to this point, the simulator does not actually use these algorithms directly; rather, it simulates them.
 - i. Computes transmit time.
 - ii. Computes receive time.
 - iii. Keeps track of outgoing and incoming waveforms.
 - d. Transmitter simulation
 - i. Configurable peak power.
 - ii. Response to “triggers on/off” commands.
 - iii. Response to “inhibit” commands from RF safety controller (via radar-control RTP).
 - iv. Configurable transmit line losses.
 - e. Receivers simulation
 - i. Modeled as string of black boxes with configurable gain and noise figure and attenuators.
 - ii. Gain and noise power computed from the model.
 - iii. Includes automatic gain control algorithm that is waveform and target dependent; the target is selected by the control program.
- 4. Net result: simulates real-time radar data in presence (or absence) of simulated targets.
 - a. Format of real-time data is identical to live data.
 - b. Data rate is, on average, identical to live data. (In the event that the simulator cannot keep up, it drops the data returns)
 - c. Simulated returns from targets depend on hardware state (e.g., pedestal pointing and antenna beamwidth) and target characteristics (e.g., position and RCS).

Appendix B. Minimal Implementation Requirements As Specified By Sponsor

1. Target simulations:
 - a. Simulate as many targets as possible. The minimal goal for the implementation should be 1000.
 - b. Center-of-mass translational motion model: One-shot ballistic or time-tagged polynomial trajectories would be ideal, but simple fixed targets would suffice for the time being.
 - c. Constant RCS is adequate.
2. Environment simulation can be disregarded for now.
3. Hardware simulations should report their current state in the AUX data:
 - a. Although a fixed pedestal pointing would be acceptable (as long as it points the antenna at the targets), a model that responds to pointing commands will be more useful for testing.
 - b. The antenna can be modeled as a parabolic dish antenna with a monopulse feed. It need not be configurable. A simple look-up table of sum, traverse, and elevation response vs. angle offsets from boresight should suffice.
 - c. Timing:
 - i. The time between transmit pulses is determined by the commanded PRI.
 - ii. The time between each transmit pulse and the beginning of the corresponding receive window is computed from a commanded range, radial velocity, and radial acceleration. (In other words, figure out how long it would take the pulse to get back to the radar if it is reflected off a target having the commanded motion.)
 - iii. Optional: model ambiguous returns; i.e., a return from a given pulse does not have to arrive back at the radar before the next pulse is transmitted.
 - d. The transmitter simulation need not be configurable and can be treated as “always on.”
 - e. A receiver model with constant gain and noise power would be acceptable for the time being.
4. Net result: simulate real-time radar data in presence (or absence) of simulated targets.
 - a. Format of real-time data is identical to live data.
 - b. Data rate is, on average, identical to live data.
 - c. Simulated returns from targets depend on hardware state (e.g., pedestal pointing and antenna beamwidth) and target characteristics (e.g., position and RCS).

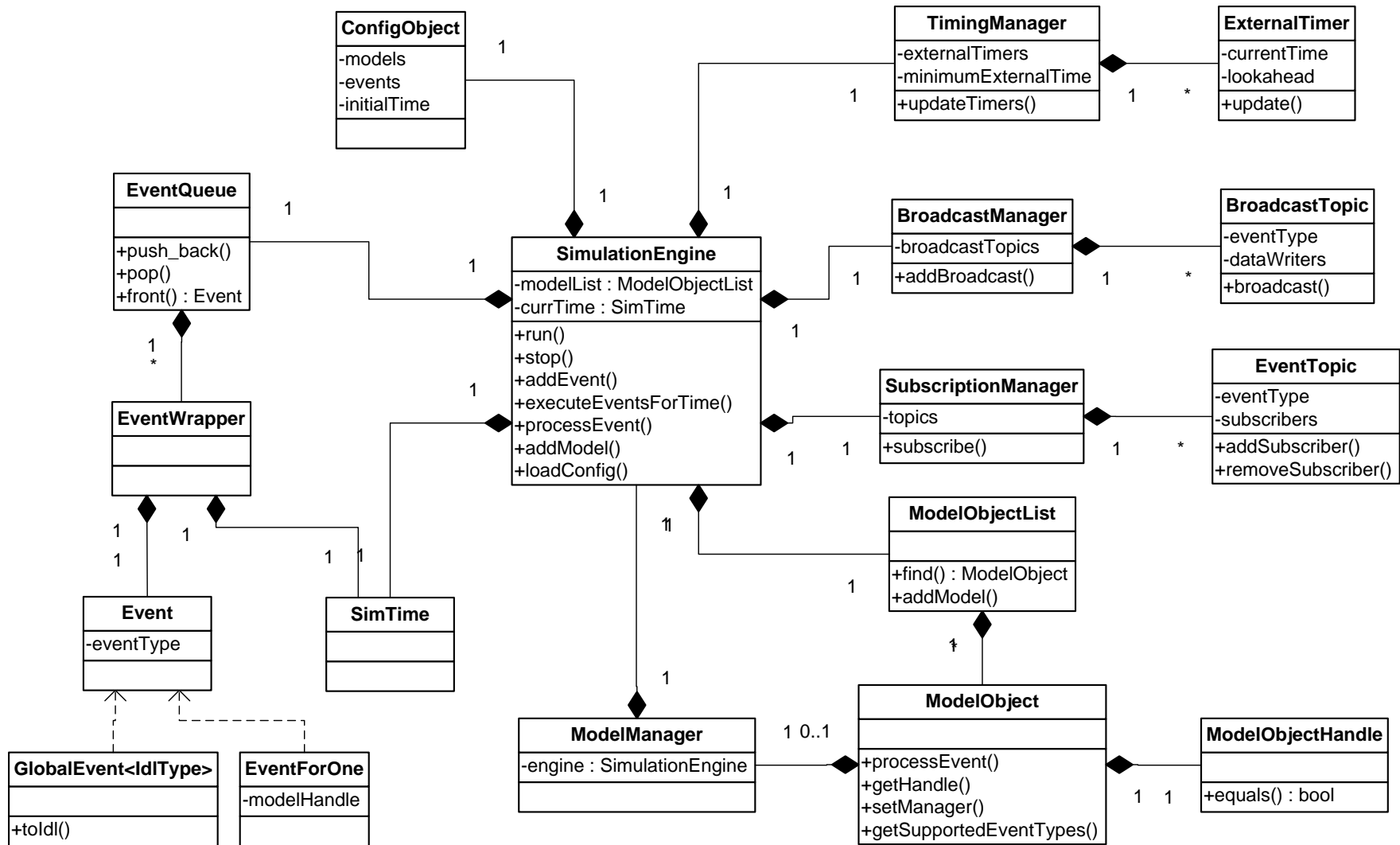
Appendix C. Design Review

Date: Thursday, September 30th, 2010

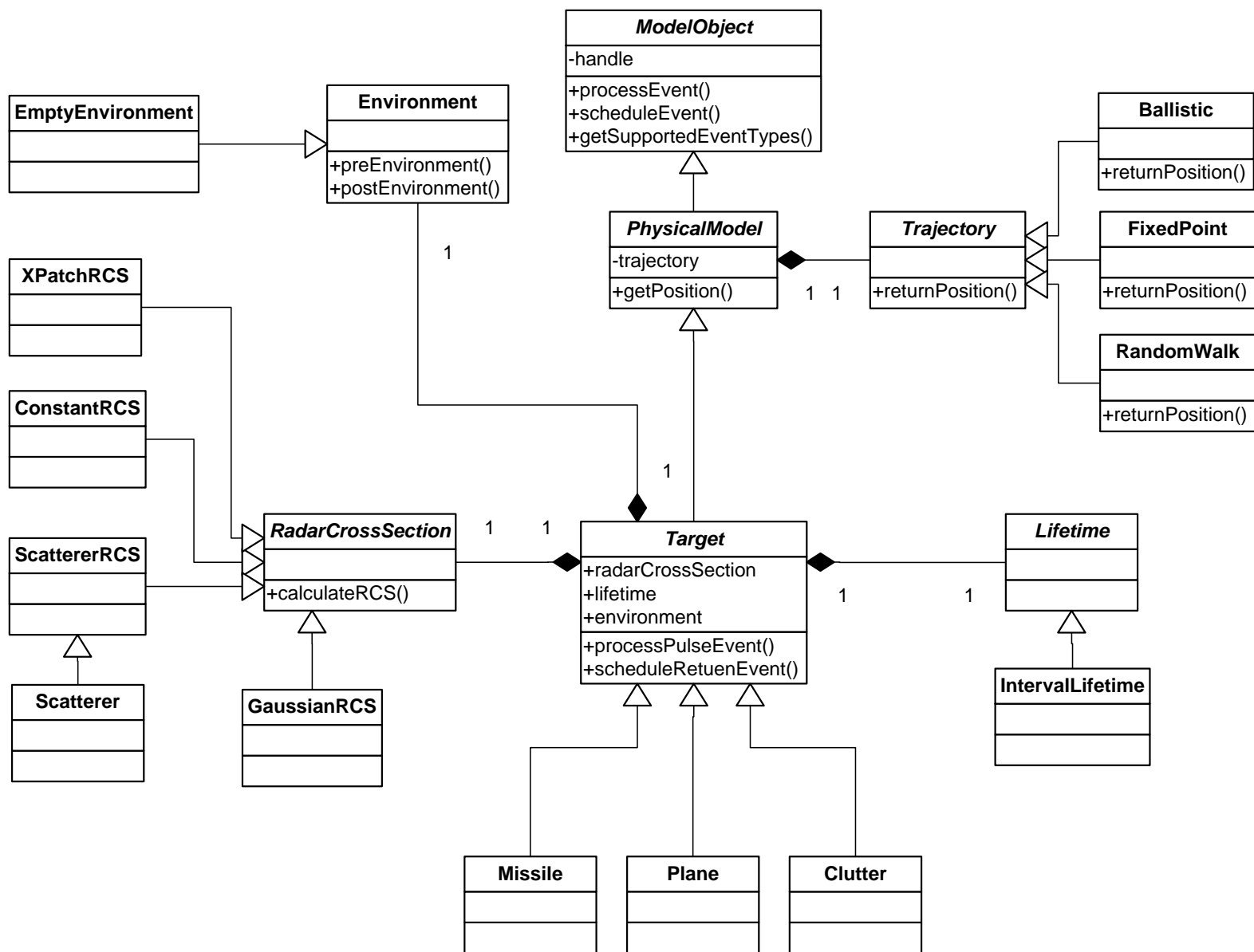
Comments:

- Radar Cross Section can be calculated at the target, but for the highest fidelity simulation it should be modeled at the receiver since the orientation and position of the receiver affects the cross-sectional view of the return.
- The old simulator has specific models for clutter and noise. The new simulator has nothing explicit for calculating these things. However, the new simulator could create actual targets to represent clutter and noise.
- Factors such as wave bandwidth are currently not a part of the pulse structure, but could be added easily.
- The transmitter and pulses created by it only represent the average power and frequencies being projected. A more accurate transmitter would use a sinusoidal function to model the changing power and frequency in a given range.
- In-phase and out-of-phase energy from returns is not being calculated. When some of the energy is returned to the receiver, it is expecting a specific frequency that it listens for. However, based on environmental effects and the distance to the target, there is a frequency shift that occurs. This is interpreted as a phase-shift by the receiver, causing some of the energy to be returned “out-of-phase” based on how much the frequency deviated from the expected frequency.
- Pulses should have durations when they are transmitted. We were originally modeling the pulse as being projected over an infinitesimally small period of time. However, a pulse will be transmitted for a specific amount of time, which affects the number of range gates that will receive energy returns. This can be added to our existing model very simply by giving pulses a duration or “width” and changing the calculations that are used to fill up the range gates.
- Trajectories can support a random walk by taking in input from a randomly generated table or by using a random function with a common seed in order to produce re-creatable data.
- Pulses could be represented as a more complex series of electronic samples. This could be accomplished using a sinusoidal function as described earlier, or by using a separate pulse structure to represent each “sample” that is being projected from the transmitter.
- The simulation engine was presented and explained with no negative comments. It was also explained that RTCL would be used as the wrapper around the middleware used for distributed communication.
- Despite a few small changes that need to be made, the Lincoln Laboratory staff appears excited to develop the simulator further. Comment from meeting attendee regarding overall design: “You presented a design which passed Lincoln Review.”

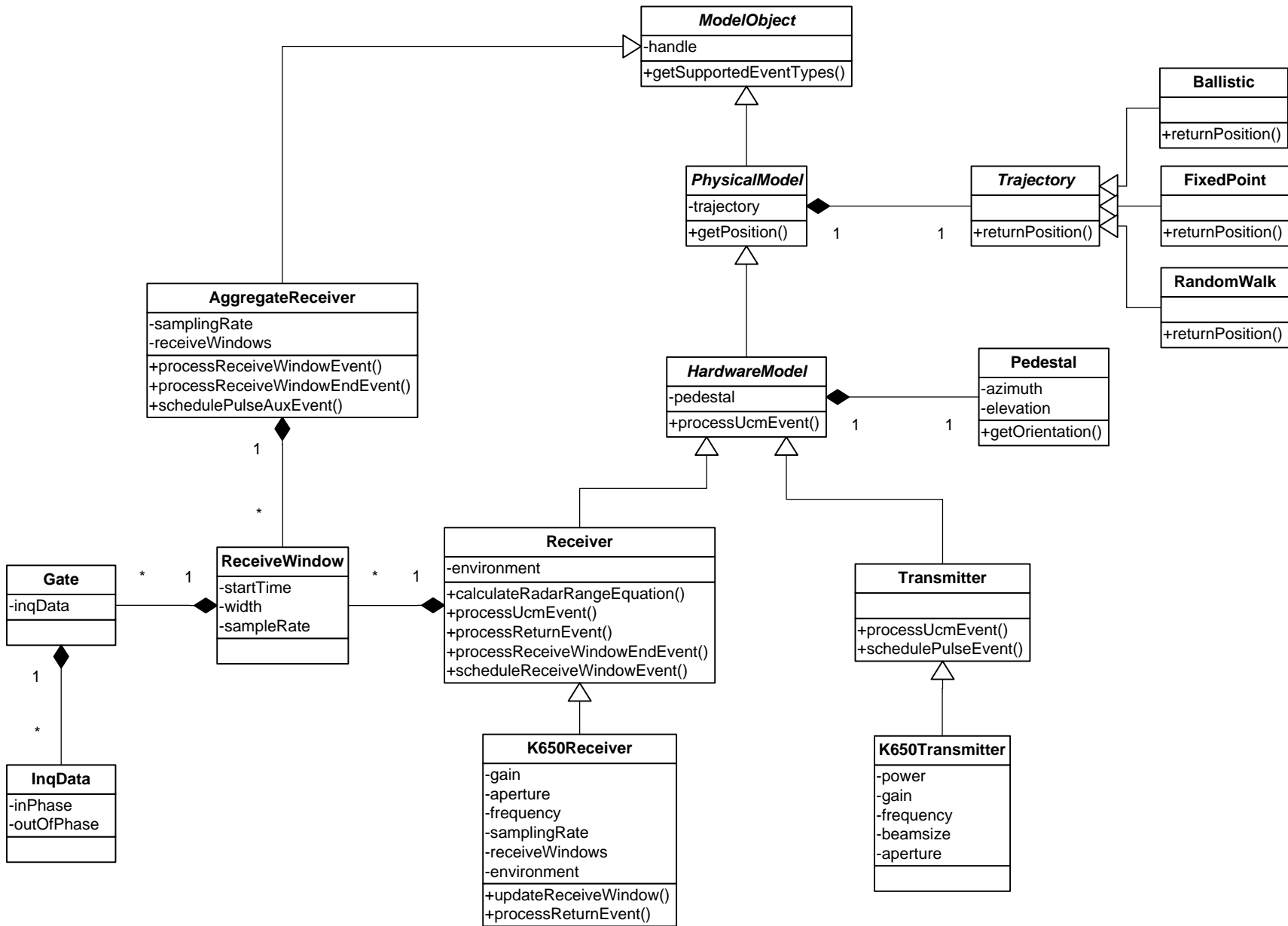
Appendix D. Simulation Engine Class Diagram



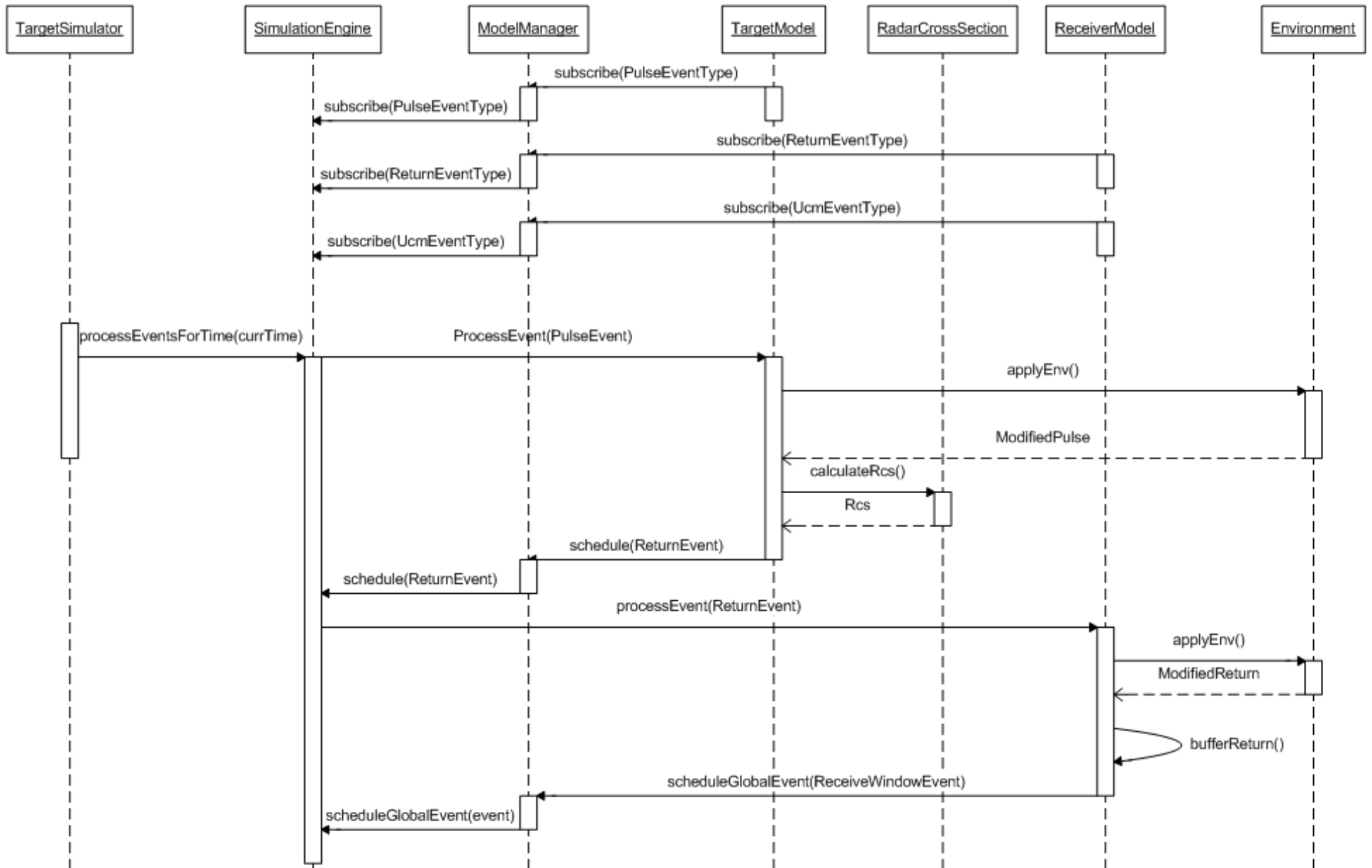
Appendix E. Target Model Class Diagram



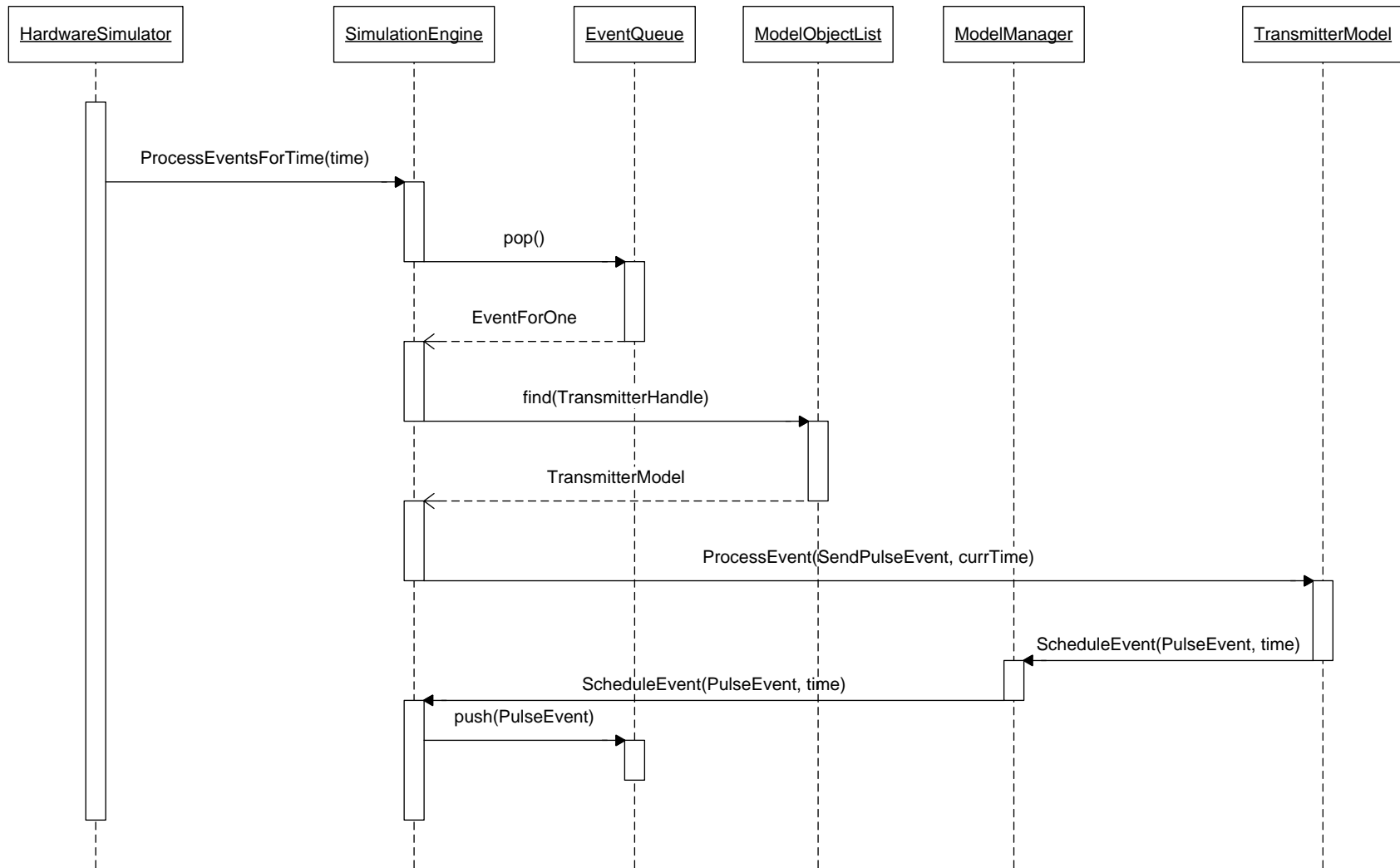
Appendix F. Hardware Model Class Diagram



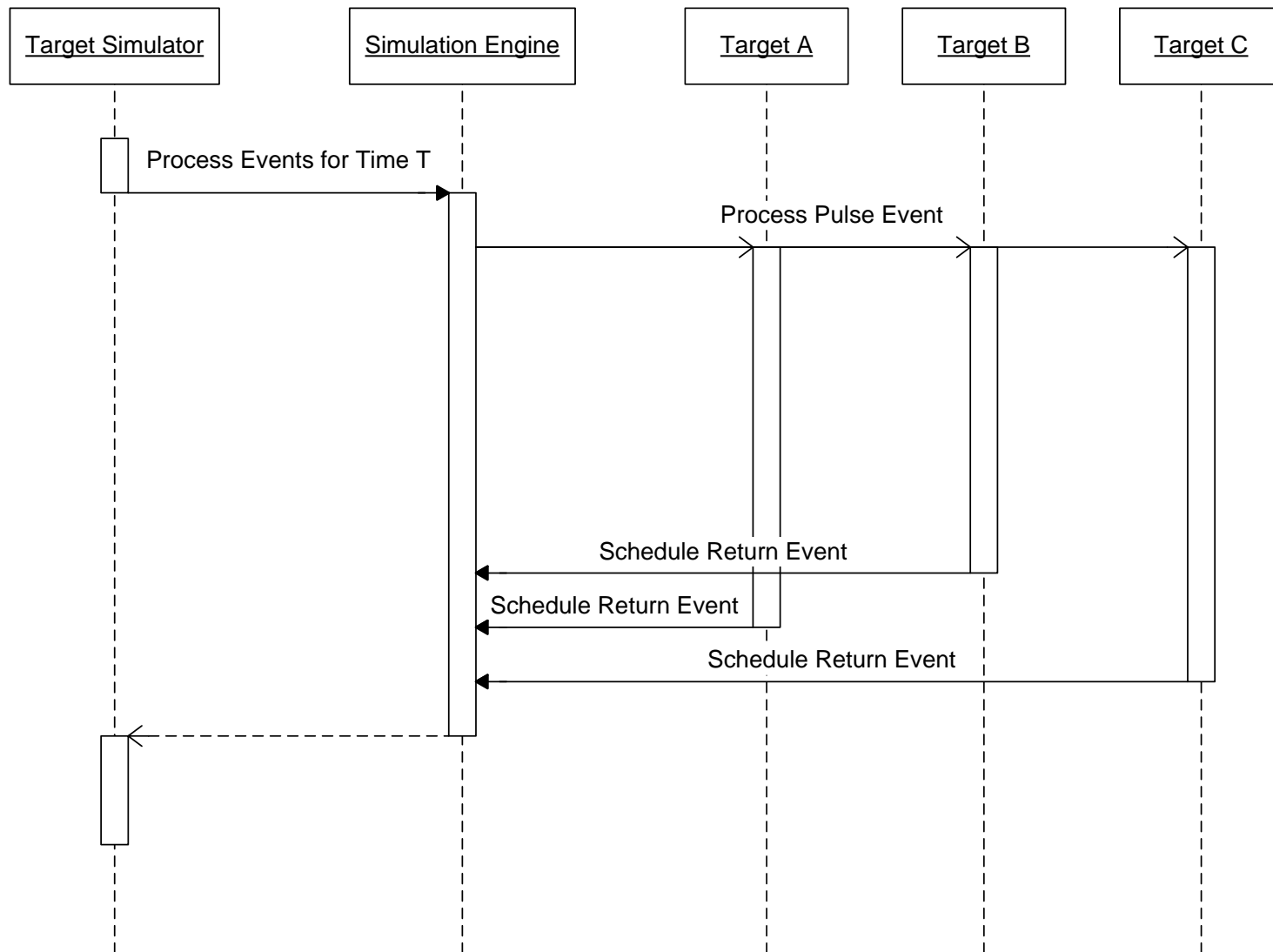
Appendix G. Target Simulator Sequence Diagram



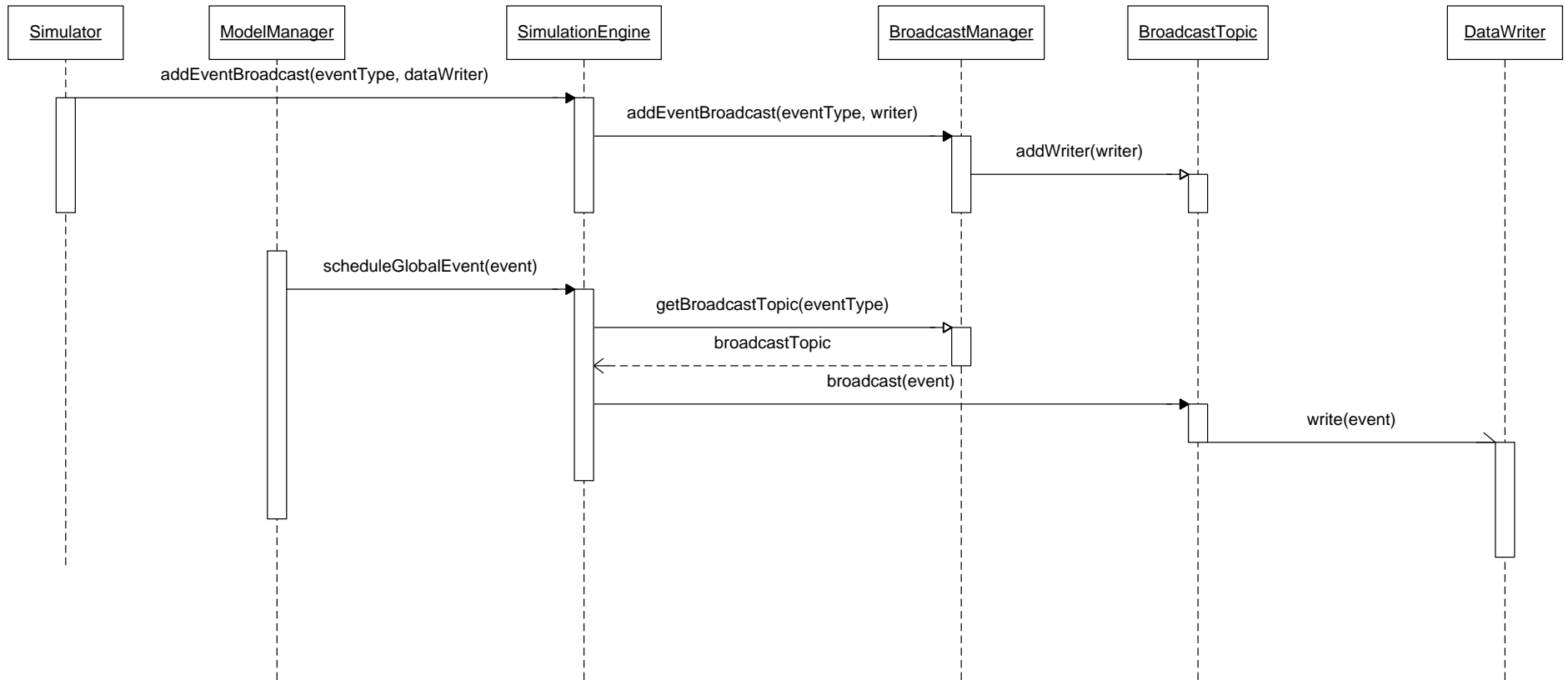
Appendix H. Hardware Simulator Sequence Diagram



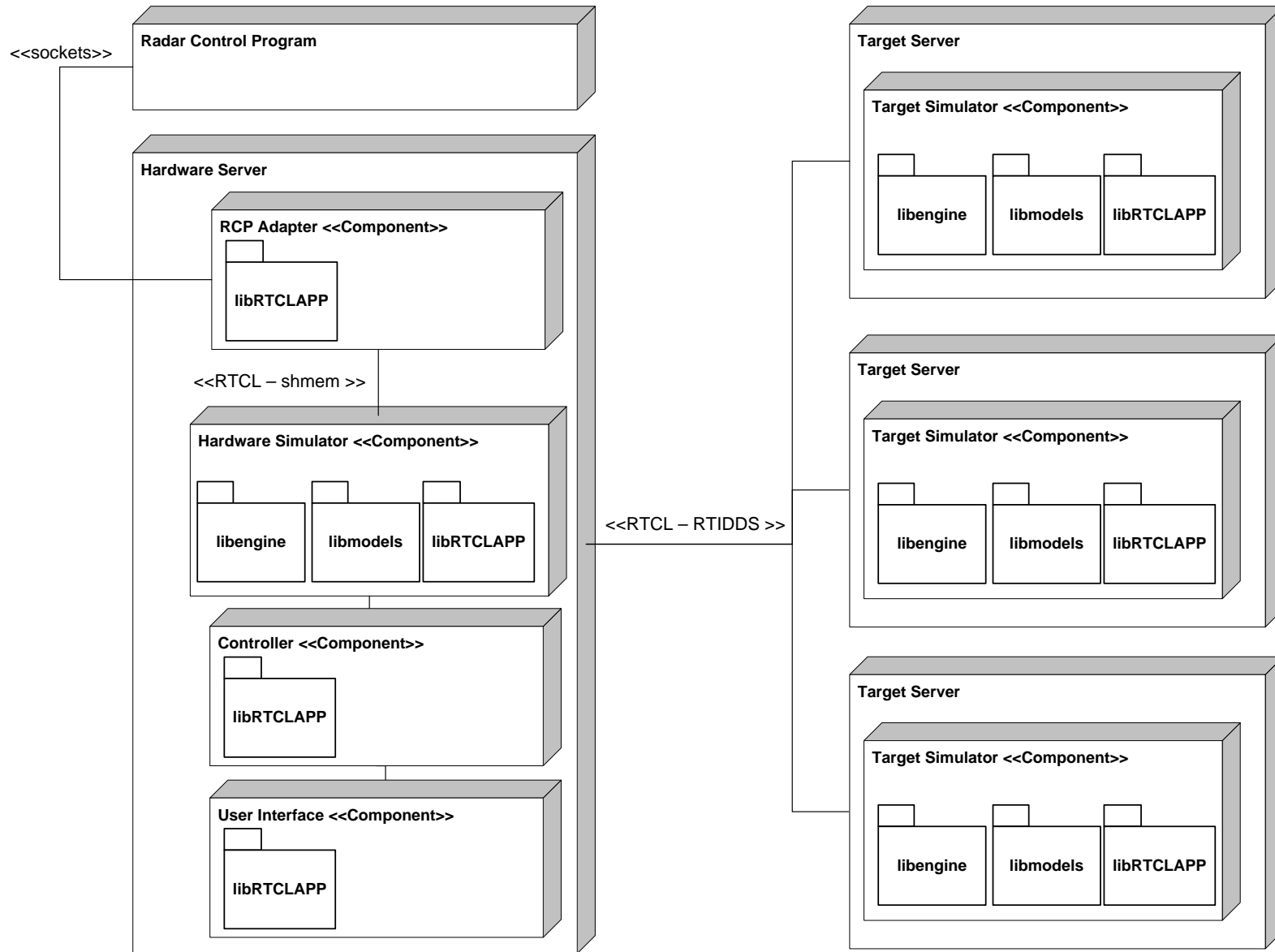
Appendix I. Target Simulator Parallelism Sequence Diagram



Appendix J. Simulation Engine Global Event Broadcast Sequence Diagram



Appendix K. System Deployment Diagram



Appendix L. Glossary

Aperture – Area of an antenna, measured in meters squared

Aux – Auxiliary data structure containing metadata regarding the current state of the radar hardware

Azimuth – The rotational position of an antenna about the vertical axis

Communications Middleware – Computer software that provides a set of services to facilitate inter-process communication

Component – A process with a finite state machine that responds to control messages

Distributed System – Any software system in which multiple computers are used to perform a subset of the total work

DVERT – “Distributed Virtual Environment for Radar Testing;” the simulation architecture described within this paper.

Elevation – The rotational position of an antenna about the horizontal axis

Event – A discrete action occurring at a finite time. The sole method of inter-model communication.

Interface – Point of interconnection between two systems

Lookahead – The amount of time into the future that a simulation engine promises to not send a message

Managed Event – Decorator class containing logic to determine which models will process domain-specific events

ModelObject – Common class that defines the interface from which all models inherit

Models – Objects within the simulation that contain domain-level knowledge

OASIS – A layered architecture for generic simulation developed at Lincoln Laboratory

Pulse – An electromagnetic wave generated by a radar transmitter

PulseAux – The combination of return energy and hardware metadata that is passed back to the RCP

Radar Cross Section – A measure of a target’s ability to reflect energy, measured in meters squared.

Range Gates – Time-tagged periods during which the receiver listens for return energy. Each period of time corresponds to a range that a target is estimated to be located at.

RCP – A real-time radar control program responsible for configuring radar hardware

Receiver – Radar hardware that receives electromagnetic returns

ROSA / ROSA II – A hardware and software modernization effort at Lincoln Laboratory

RTCL – An abstraction for communications middleware

Scheduling – Creating a new event

Simulation Engine – Simulation component responsible for scheduling events and keeping them ordered

Target – Any object that generates a return when illuminated by a pulse

Transmitter – Radar hardware that generates electromagnetic pulses

UCM – Message that contains hardware configuration data. Generated by the RCP

References

1. Alhir, Sinan Si. *UML in a Nutshell: A Desktop Quick Reference*. Cambridge: O'Reilly, 1998. Print.
2. Amdahl, G. M. "Validity of the single processor approach to achieving large scale computing capabilities." In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (Atlantic City, New Jersey, April 18 - 20, 1967). AFIPS '67 (Spring). ACM, New York, NY, 483-485.
3. Apache Software Foundation. *Apache Subversion*. <http://subversion.apache.org>. 2010. Computer Software.
4. AT&T Research. *GraphViz*. <http://www.graphviz.org>. 2010. Computer Software.
5. Brindley, Lana. *Release Notes for the Red Hat Enterprise MRG 1.2 Release Edition 3*. Red Hat, Inc. Web. 19 Oct. 2010.
6. Bullseye Testing Technology. *BullseyeCoverage*. <http://www.bullseye.com>. 2010. Computer Software.
7. D'Addario, Larry R. *Large Transmitting Arrays for Deep Space Uplinks, Solar System Radar, and Related Applications*. Tech. Web. 21 Sept. 2010. <<http://trs-new.jpl.nasa.gov/dspace/bitstream/2014/37533/1/05-2202.pdf>>.
8. Delaney, William P., and William W. Ward. "Radar Development at Lincoln Laboratory: An Overview of the First Fifty Years." *Lincoln Laboratory Journal* 12.2 (2000): 147-66. MIT Lincoln Laboratory. Web. 19 Oct. 2010. <http://www.ll.mit.edu/publications/journal/pdf/vol12_no2/12_2radardevelopment.pdf>.
9. The Eclipse Foundation. *Eclipse IDE for C/C++ Developers*. <http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/heliosr>. 2010a. Computer Software.

10. The Eclipse Foundation. *About the Eclipse IDE*. <http://www.eclipse.org/org/>. 2010b
11. The Eclipse Foundation. *The Eclipse Marketplace*. <http://marketplace.eclipse.org>. 2010c.
Computer Software.
12. Fitch, Kevin. *CxxTest*. <http://cxxtest.tigris.org>. 2009. Computer Software.
13. Fowler, Martin. "Test-Driven Development: A Conversation with Martin Fowler." Interview by Bill Venners. *Test Driven Development*. Artima, 2 Dec. 2002. Web. 19 Oct. 2010.
<<http://www.artima.com/intv/testdriven.html>>.
14. Free Software Foundation. *GNU Make*. <http://www.gnu.org/software/make>. 2010a. Computer Software.
15. Free Software Foundation. *The GNU Compiler Collection*. <http://gcc.gnu.org>. 2010b. Computer Software.
16. Free Software Foundation. *gcov-a code coverage program*.
<http://gcc.gnu.org/onlinedocs/gcc/Gcov.html>. 2010c. Computer Software.
17. Fricker, Sébastien. *Test Cocoon*. <http://www.testcocoon.org>. 2009. Computer Software.
18. Fujimoto, Richard M. *Parallel and Distributed Simulation Systems*. New York: John Wiley and Sons, 2000. Print.
19. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Reading, MA: Addison-Wesley, 1995. Print.
20. Hartsberger, Leon. *Eclipse C++ Unit Testing*. <http://sourceforge.net/projects/ecut>. 2008.
Computer Software.
21. Gleixner, Thomas. *Cyclictest*. <https://rt.wiki.kernel.org/index.php/Cyclictest>. May 2010.
Computer Software.
22. Hoffmann, Mark R. *EclEmma*. <http://www.eclEmma.org>. July 2010. Computer Software.

23. Kalibera, Tomas. *CD_x: A Family of Real-time Java Benchmarks*.
<http://d3s.mff.cuni.cz/publications/download/cdx09.pdf>. 2009. Computer Software.
24. McLaughlin, Brett, Gary Pollice, and David West. *Head First Object-Oriented Analysis and Design*. Sebastopol, CA: O'Reilly, 2007. Print.
25. Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Upper Saddle River, NJ: Addison-Wesley, 2005. Print.
26. MIT Lincoln Laboratory. *About the Lab*. <http://www.ll.mit.edu/about/about.html>. Copyright 2001-2010a. Accessed 19 October 2010.
27. Moore, Gordon E. "Cramming More Components onto Integrated Circuits." *Electronics Magazine* 19 Apr. 1965. Intel Corporation. Web. 1 Sept. 2010.
28. Nokia, Inc. *Qt- A cross platform application and UI framework*. <http://qt.nokia.com/products>. 2010. Computer Software,
29. O'Donnell, Robert M. "An Introduction to Radar." Lecture. Lexington. 18 June 2002. MIT Lincoln Laboratory. Web. 1 Sept. 2010.
<<http://www.ll.mit.edu/workshops/education/videocourses/intro radar/>>.
30. Oracle, Inc. *JavaDoc*. <http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html>. February 2004. Computer Software.
31. Phippard, Mark. *Subclipse*. <http://subclipse.tigris.org>. 2009. Computer Software.
32. Red Hat, Inc. *Red Hat MRG Real-time Kernel*. <http://www.redhat.com/mrg/realtime>. October 2010. Computer Software.
33. Rejto, S. "Radar Open Systems Architecture and Applications," *Radar Conference, 2000. The Record of the IEEE 2000 International*. pp.654-659, 2000.
34. Sangiolo, T. L. "Lincoln Space Surveillance Complex (LSSC) Modernization." *Proceedings of the*

- 2001 Space Control Conference*. 2001 Space Control Conference, Lincoln Laboratory, Lexington. 2001. Defense Technical Information Center, 8 May 2002. Web. 19 Oct. 2010. <<http://www.dtic.mil/srch/doc?collection=t3&id=ADA400867>>.
35. Secor, H. W. "Tesla's Views on Electricity and the War." *The Electrical Experimenter* Aug. 1917. *Tesla's Views on Electricity and the War*. Twenty-First Century Books. Web. 19 Oct. 2010. <<http://www.tfcbooks.com/tesla/1917-08-00.htm>>.
36. Sommerlad, Peter. *C++ Unit Testing Easier*. <http://accu.org/index.php/articles/1349>. 2006. Computer Software.
37. Tanenbaum, Andrew S., and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. 1st ed. Upper Saddle River, NJ: Prentice Hall, 2002. Print.
38. Thomas Publishing Company. *The Electro Magnetic Spectrum*. Digital image. *Manufacturing Electronic Microwave Components*. Thomas Publishing Company. Web. 19 Oct. 2010. <<http://www.thomasnet.com/articles/automation-electronics/electronic-microwave-manufacturing>>.
39. Toomay, J. C. *Radar Principles for the Non-specialist*. New York: Van Nostrand Reinhold, 1989. Print.
40. Vandevoorde, David, and Nicolai M. Josuttis. *C++ Templates: the Complete Guide*. Boston: Addison-Wesley, 2003. Print.
41. Van Heesch, Dimitri. *Doxygen*. <http://www.stack.nl/~dimitri/doxygen>. October 2010. Computer Software.