SENSORS IN OBJECT ORIENTED DISCRETE EVENT SIMULATION

Kirk Andrew Stork–Lieutenant, United States Navy
B.S.E., University of Washington, 1989

The growing cost of physical tests and evaluations of military systems has resulted in increased use of computer simulations to provide decision support information. Many such systems, such as weapons and countermeasure systems, rely on sensors. Hence, development of widely applicable computer models for sensors is vitally important. This research investigates the possibility of developing sensor simulations as components for use in models with varying fidelity and purpose. Development of abstractions is emphasized to maximize the applicability of components in a variety of modeling contexts. Concrete examples of reusable sensor components are demonstrated in working models and a preliminary design for a generalized modeling framework is proposed.

Master of Science in Operations Research, September, 1996

Advisor: Arnold H. Buss, Department of Operations Research
Second Reader: James Eagle, Department of Operations Research

Unclassified/A

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SENSORS IN OBJECT ORIENTED DISCRETE EVENT SIMULATION

by

Kirk A. Stork

September, 1996

Thesis Advisor: Arnold H. Buss

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September, 1996	Master's Thesis

4. TITLE AND SUBTITLE

SENSORS IN OBJECT ORIENTED DISCRETE EVENT SIMULATION

5. FUNDING NUMBERS

6. AUTHOR(S)

Stork, Kirk, A.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

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

Naval Research Laboratory, Code 5740
4555 Overlook Ave SW, Washington, DC 22193

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT(*maximum 200 words*)

The growing cost of physical tests and evaluations of military systems has resulted in increased use of computer simulations to provide decision support information. Many such systems, such as weapons and countermeasure systems, rely on sensors. Hence, development of widely applicable computer models for sensors is vitally important. This research investigates the possibility of developing sensor simulations as components for use in models with varying fidelity and purpose. Development of abstractions is emphasized to maximize the applicability of components in a variety of modeling contexts. Concrete examples of reusable sensor components are demonstrated in working models and a preliminary design for a generalized modeling framework is proposed.

14. SUBJECT TERMS

Modelling and Simulation, Object Oriented Programming, Java

15. NUMBER OF PAGES

125

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited

SENSORS IN OBJECT ORIENTED DISCRETE EVENT SIMULATION

Kirk A. Stork
Lieutenant, United States Navy
B.S.E., University of Washington, 1989

Submitted in parial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL
September, 1996

Author: _____
Kirk A. Stork

Approved by: _____
Arnold H. Buss, Thesis Advisor

_____
James Eagle, Second Reader

_____
Frank Petho, Chairman
Department of Operations Research

ABSTRACT

The growing cost of physical tests and evaluations of military systems has resulted in increased use of computer simulations to provide decision support information. Many such systems, such as weapons and countermeasure systems, rely on sensors. Hence, development of widely applicable computer models for sensors is vitally important. This research investigates the possibility of developing sensor simulations as components for use in models with varying fidelity and purpose. Development of abstractions is emphasized to maximize the applicability of components in a variety of modeling contexts. Concrete examples of reusable sensor components are demonstrated in working models and a preliminary design for a generalized modeling framework is proposed.

THESIS DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

TABLE OF CONTENTS

I. INTRODUCTION . . . 1

A. MISSILE COUNTERMEASURES . . . 2

B. CURRENT EFFORTS . . . 2

II. A CONCEPTUAL MODEL OF ACTIVE SENSORS . . . 5

A. WORLD VIEWS . . . 5
1. Strategic Planner . . . 5
2. Tactics Developer . . . 6
3. Procurement Professional . . . 6
4. System Designer . . . 7

B. OPERATIONAL DESCRIPTION . . . 7

C. OBJECT DESIGN . . . 8

D. DISCRETE EVENT DESIGN . . . 10
1. The Active Sensor Entity . . . 11
2. Active Sensor Signal . . . 12
3. Responder . . . 12
4. Sensor Owner . . . 13
5. Sensor User . . . 13

E. ABSTRACT IMPLEMENTATION . . . 14
1. The Sensor Interfaces . . . 14
2. The Signal Interfaces . . . 15
3. The Responder Interfaces . . . 17
4. Other Interfaces . . . 17

III. DEMONSTRATIONS . . . 19

A. SCENARIO . . . 19

B. COOKIE CUTTER SENSOR . . . 20
1. Model . . . 20
2. Simulation . . . 21
a. CCActiveSensor Class . . . 21
b. CCActiveSensorSignal Class . . . 22
c. CCActiveSignalResponder Class . . . 23
d. Main Program . . . 23
3. Results and Verification . . . 24

C. MONOPULSE RADAR . . . 25
1. Model . . . 26
2. Simulation . . . 27
a. MonopulseRadar class . . . 27
b. PulseTrain class . . . 27
c. PulseTrainResponder class . . . 28
d. Main Program . . . 28
3. Results . . . 28

IV. APPLICATION . . . 31

A. DESIGN GUIDELINES . . . 31

B. ARBITRATION . . . 32
1. Referee . . . 33
2. Environment . . . 34
3. Opportunities . . . 34

C. LOCATION . . . 35
1. The Position Abstract Class . . . 35
2. The Positionable Interface . . . 36

D. MOVEMENT . . . 37
1. The Velocity Abstract Class . . . 37
2. The Moveable Interface . . . 38

E. FURTHER DEVELOPMENT . . . 38

V. CONCLUSIONS . . . 41

LIST OF REFERENCES . . . 43

BIBLIOGRAPHY . . . 45

APPENDIX A. ABSTRACT SENSOR COMPONENT LISTINGS . . . 47

APPENDIX B. COOKIE CUTTER SENSOR DEMONSTRATION LISTINGS . . . 61

APPENDIX C. MONOPULSE RADAR DEMONSTRATION LISTINGS . . . 83

INITIAL DISTRIBUTION LIST . . . 109

LIST OF FIGURES

2.1 Conceptual Active Sensor Component 8
2.2 Inheritance Graph for Abstract Sensor Entities 9
2.3 Event Graph for the Abstract Active Sensor 11
2.4 Event Graph for Abstract Sensor Signal 12
2.5 Event Graph for Abstract Sensor Signal Reflector 13
2.6 Event Graph for Abstract Sensor Owner 13
2.7 Event Graph for Abstract Sensor User 14
2.8 Event Graph for Abstract Sensors 15
2.9 Expanded Sensor Interface Inheritance Tree 16
2.10 Expanded Signal Interface Inheritance Tree 16
2.11 Expanded Responder Interface Inheritance Tree 17

3.1 Barrier Search Scenario 19
3.2 Cookie Cutter Probability of Detection 20
3.3 Cookie Cutter Model Inheritance Graph 22
3.4 Cookie Cutter Model Results 25
3.5 Monopulse Radar Model Results 29

ACKNOWLEDGEMENT

The author would like to express his thanks to the many people behind the scenes whose help was instrumental in the completion of this thesis. First to Dr. Arnold Buss, who's expertise and experience in computer modeling shaped so many design decisions. Our many philosophical discussions provided immeasurable motivation and inspiration. To Dr. James Eagle for remarkable patience and firm grounding in the theoretical aspects of search and detection. To the Naval Research Laboratory, especially Dr. George Weisbach for funding a valuable research experience, and Dr. William Hardenburg for his expert tutelage on radar theory. Finally, to my wife Elizabeth for her loving support and patience without which this research could not have been completed.

I. INTRODUCTION

Military organizations of the world constantly develop new war fighting technology. Advances in our own technologies produce the potential for operational and organizational improvements that are unforeseen. However, advances in the capabilities of potential adversaries present challenges that must be answered by our systems and organizations.

One area of concern is the advancement of anti-ship missile technologies available to potential enemies of the United States. As the cost of technology drops and the availability of technology rises, the challenges faced by our shrinking resources grows. We are faced with an urgent need to quickly improve ship missile defenses, or at least to understand our vulnerabilities. Research in this field is vigorous.

Because our missiles are expensive and foreign missiles are unavailable, test programs for missile defense systems are often infeasible. Even when feasible, physical testing on a range is expensive. Consequently, the use of simulation models to support design, procurement and research funding decisions is ever increasing. This increased use of simulation to support important decisions motivates the study and construction of simulation tools that are applicable to test and evaluation as well as to tactical development and combat effectiveness studies.

Simulation modeling can provide decision support at all stages of a system's development and employment. During concept development, medium to low resolution models can provide insight into future needs. Once needs are identified, simulation may be used by engineers to prototype designs, and by physicists to test theoretical concepts. When prototypes become available, the test and evaluation community can use simulation to plan physical testing, and then to extrapolate physical test results with calibrated simulations. When a system is selected for procurement, simulation may help define tactical concepts prior to the system's introduction into the inventory. Finally, when a system is employed, a simulation is can assess the effectiveness of organizations equipped with the system. Systems are often improved during their lifecycles, and all these uses of simulation may be repeated to develop, test and evaluate the improved system.

Rapidly increasing computer capacity and programming skill has made this scenario possible. As computing capabilities grow, the demand for better and more complex software grows. But high demand and limited resources have made software development an expensive undertaking. The expense of software, such as simulation models, necessitates a program for model and code reuse to minimize the demands placed on limited programming resources while satisfying the growing requirements.

In this thesis, we examine simulation support for sensor systems. After a brief discussion of current efforts associated with the sensing models used for soft-kill missile countermeasure system support, we develop a model for active sensor components. The conceptual model we develop is then demonstrated in a number of contexts to evaluate the possibility of a code reuse scheme in real projects, including countermeasure evaluations. Finally, we discuss the supporting structure needed to make a component, such as our sensor model, reusable on an organizational scale.

A. MISSILE COUNTERMEASURES

Missile countermeasures are divided into two broad categories. Hard-kill systems, which aim to destroy the missile before it reaches its target, and soft-kill systems, which attempt to confuse the missile into pursuing a false target. In practice, the two systems interfere with each other, making them difficult to employ together.

Soft-kill countermeasure systems can be active or passive, the most common being decoys, chaff and radar jamming devices. Chaff is well understood, but continues to be studied in the presence of new missiles. Radar jammers are of continued interest because of new missile radars, and because they interfere with our own sensor systems. Decoys are of particular interest since our improving ability to manipulate radar signals can make them highly effective.

B. CURRENT EFFORTS

The Ship's Electronic Warfare Systems Division of the Naval Research Laboratory (SEWS) provides simulation support for soft-kill countermeasure acquisition. SEWS has a long history of providing support for countermeasure and radar system development and procurement for the USN, and is home to a large body of expertise in radar engineering.

Originally, modeling efforts at SEWS supported research on the radar reflection characteristics of existing passive missile decoys, ships and aircraft. These efforts have provided important performance evaluations that have contributed both to design and to the procurement process.

Recently, focus has shifted to include evaluation of countermeasure effectiveness, including countermeasures still under design. Today, the effort is expanding towards support of tactics development and vulnerability assessment. The shift of interest towards evaluation of effectiveness, in addition to evaluation of performance, has spawned a relationship between SEWS and the Department of Operations Research at the Naval Postgraduate

School. This thesis is the second[1] to explore application of Operations Research techniques to the problems being addressed at SEWS.

As we will discuss in Chapter II, these changing demands for simulation support are a typical phenomenon encountered as a system or concept evolves. Operations Research techniques will increase in applicability as modeling requirements move further from system performance analysis and closer to analysis of effectiveness.

Two models are currently used for production level research at SEWS. VIEWS [Hardenburg, 1995], a medium resolution physical model of radar was developed in the 1970's to be a high resolution model for radar research is used directly for countermeasure support. C Routines Utilizing Ships Environments and Missiles (CRUISE_Missile) [Fletcher, 1996], the successor to VIEWS chronologically, is an even higher resolution physical radar system model.

CRUISE_Missile the high resolution model, simulates signal processing at the level of components on circuit boards, and simulates a ship as several thousand corner radar reflectors. Although CRUISE_Missile was developed for other purposes, it has been used to model missile-ship engagements with chaff. A single run of a single ship, single missile engagement requires hours to complete.

CRUISE_Missile is intended to so accurately model every aspect of the real systems that a single run is sufficient to obtain useful information. CRUISE_Missile has been very successful for its intended purpose but, as will be discussed in Chapter II, it is difficult to extend such a model to uses other than pure engineering. Since systems are modeled by simulating individual electrical components, the job of modeling a new missile seeker is tantamount to building that seeker by hand, a time consuming and expensive task.

The VIEWS model is lower resolution, and after two years of renewed development is the principle model for countermeasure analysis at NRL. VIEWS was resurrected and modernized to fill the need for a lower resolution model for the effectiveness analyses describe above. VIEWS is is a better candidate for extension to operational testing because of it's lower resolution and stylized approach to signal processing. Consequently, VIEWS is has been ported from FORTRAN to C++ to ease development of this extension and to facilitate more complex scenarios.

VIEWS models the target ship as several corner reflectors, and models signal processing at the process level. Presently, VIEWS models a single ship, DDG-51, a single missile, and several countermeasures. The stylized model of signal processing used in VIEWS facilitates modeling of additional missile seekers because it is process based; i.e., it models

[1] The first thesis is "Simulation of a Radar Detection Model Using the NPS Platform Foundation," Aaron S. Ellison, March 1996

the characteristics of the seeker, rather than the circuits. The simplified ship model also eases extension of the overall model because fewer corner reflectors must be placed and managed.

From an operational perspective, and perhaps even a combat modeling perspective, VIEWS is promising. It runs on inexpensive hardware and is fast enough to be used as a subroutine in a scenario testing model for tactics development. For combat models, VIEWS could be used to generate databases for lower resolution models, following the ATCAL/COSAGE [DTIC, 1980, unsighted] methodology.

The remainder of this thesis is organized to examine and develop a reusable model component for active sensors, and then to address the problem of reusing information, experience and computer code for simulation support of military systems as they mature. In Chapters II and III we will focus on a single aspect of the countermeasure simulation problem, sensors, and provide a concrete example of a reusable model component. We then discuss a proposed software framework in Chapter IV.

II. A CONCEPTUAL MODEL OF ACTIVE SENSORS

Development of a component, such as a sensor, requires a conceptualization of the component to be modeled. We have several goals in this chapter. First, we will describe the important features of an active sensor from the viewpoint of a number of simulation consumers. From this we develop an operational description that will serve as a guide in the following section, where we develop the sensor component conceptual model.

To facilitate construction of a reusable component, the conceptual description, or abstraction, will be designed to accommodate simulations at any resolution and all qualitative categories of modeling. The abstraction will accommodate all identified world views without imposing any particular resolution or viewpoint, and should be understandable by people not necessarily expert in sensor system design.

A. WORLD VIEWS

We identify four potential "consumers" (i.e., users) of our sensor model and attempt to define each consumer's idea of an active sensor by listing the important characteristics of the sensor from that consumer's point of view. This exercise will identify the common elements of the different viewpoints and, perhaps more importantly, the differences, to ensure our operational description accommodates all viewpoints.

1. Strategic Planner

The strategic planner sees a sensor as a device carried by some platforms which detects, and possibly locates, certain things in the vicinity of its owner. Characteristics of interest might be:

- Accuracy.
- Portability.
- Controllability.
- Susceptibility to environmental conditions.
- Susceptibility to counter detection.

The strategic planner is interested in Measures of Effectiveness associated with the information the sensor provides, rather than the details of the sensor's implementation.

2. Tactics Developer

The tactics developer, who is also interested in battle effectiveness, sees a sensor similarly to the strategic planner, but at a lower organizational level. Important characteristics for this simulation consumer might be:

- Accuracy.
- Power consumption.
- Human resource consumption.
- Susceptibility to environmental conditions.
- Controllability.
- Susceptibility to counter detection.

Like the strategic planner, the tactics developer is interested in Measures of Effectiveness associated with the information provided by the sensor, but he may also be interested in some of the details of implementation. He is interested in what the sensor does, what information it provides, and the quality of that information, but he may also be interested in some of the details of how that information is obtained.

3. Procurement Professional

The procurement professional sees a sensor as a device which has a specified purpose, and a set of specified capabilities. Because he is usually managing a contract with explicit requirements, he is interested in Measures of Performance, such as:

- Signal strength.
- Bearing and range resolution.
- Detection degradation due to environmental conditions.

and in Measures of Effectiveness, such as:

- Probability of detecting a specified target at a specified range.
- Probability of system failure.
- Probability of counter detection.

4. System Designer

The sensor designer has a much more detailed view. To him, an active sensor is a device which detects and locates certain things in the environment by emitting an energy signal, waiting for that signal to return, and then processing the returned signal to derive information. The system expert is tasked with adding to the advantages of the system and reducing the disadvantages, by meeting a set of design specifications for performance, such as:

- Power requirements.
- Signal strength.
- Antenna gain.
- Detection thresholds.

The needs of all of these simulation consumers can be satisfied by using an appropriate set of abstractions. The abstractions do not satisfy any needs themselves, but allow the construction of components that do. To define the interfaces required in the framework, we must settle on a single operational description.

B. OPERATIONAL DESCRIPTION

The most detailed description need not be adopted for the purposes of modeling a generic sensor. Indeed, as we are designing an abstraction of the sensor, there are many details that are both unnecessary and unwanted. As stated previously, our goal is to develop an abstraction that fits into models of many styles and resolutions. As we will show in Chapter III, adopting this abstraction greatly enhances the possibility that components will be applicable in some unknown future model without limiting the detail that can be achieved. We will adopt the following generalized operational description of an active sensor:

> An active sensor is a device which translates data into information. The data is acquired by emitting a signal and waiting for it to return; upon its return, the signal is processed by the sensor to generate information, which is sent to the sensor's users for an unspecified use. The sensor may have various modes of operation which are controlled by its owner via some method of issuing commands.

The entities shown in Figure 2.1, represent the sensor and all of the entities the sensor communicates with. The sensor exchanges information with its users, takes commands from its owner, and sends and receives signals. The abstract sensor component is analogous to object oriented Software ICs, due to Cox [Cox and Novobilski, 1991], and can be developed with no knowledge of an overall model in which it will eventually be used.

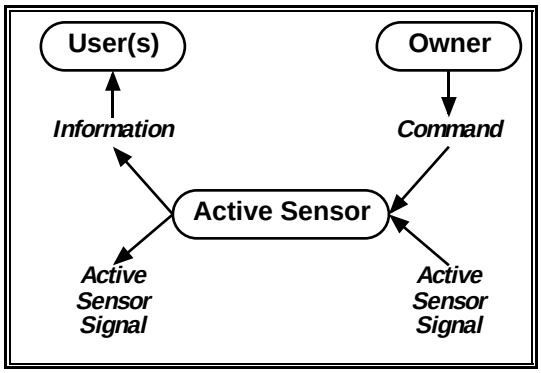


Figure 2.1. An active sensor and the other simulation entities it interacts with.

C. OBJECT DESIGN

Because we are developing the conceptual model for eventual implementation in an object oriented program, we begin design by defining the important objects, or entities in the description. We will focus on abstract objects; that is, we will specify the behaviors of each entity, but say nothing of how those behaviors will be carried out.

The sensor exists in an environment, is perhaps mounted on a platform of some kind, has an owner and, possibly, several users. The signals that carry information between entities and sensors are created by the sensor but act according to their own rules. The signal interacts with the environment and other entities as arbitrated by an entity we designate the Referee (Entity 10 below). Figure 2.2 will be a useful guide to the descriptions of these entities.

Entity 1
Signal. An object to abstract the notion of data that moves through space and time. Signals may represent concrete energy signals, such as sound waves or radar waves, or abstract signals, such as messages. In some cases, the signal may interact with the Referee to determine its own behavior.

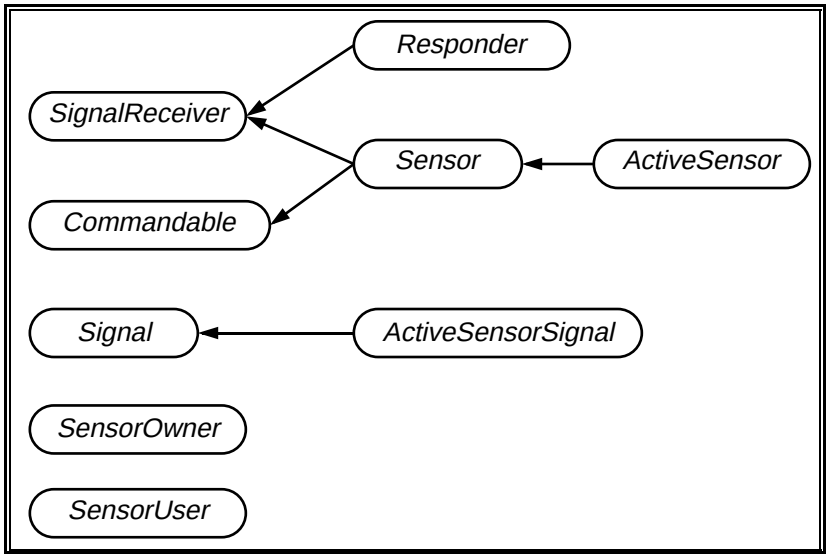


Figure 2.2. Inheritance Graph for Abstract Sensor Entities.

Entity 2
Signal Receiver. An object that can receive signals. Behavior resulting from reception of a signal is unspecified. There need be no guarantee that a particular Signal Receiver will understand all signals.

Entity 3
Commandable. An object that can receive commands. Behavior resulting from reception of a command is unspecified. There need be no guarantee that a particular Commandable entity will understand all commands.

Entity 4
Sensor. A special kind of Signal Receiver that translates signal data into information. Also, a special kind of Commandable entity. Signals are processed and the results are sent to the sensor's users, perhaps only on request. The Sensor may have change modes of operation when it receives a command from its owner.

Entity 5
Active Sensor. A special kind of Sensor that can generate its own signals. Because it is a Sensor, the Active Sensor is also a Signal Receiver and a Commandable entity.

Entity 6
Active Sensor Signal. A special kind of Signal which is created by active sensors. The Referee (Entity 10) is used to discover what other objects a signal should interact with. The

Referee is used in this manner to hide information from other entities, namely the Sensor, who should have no knowledge of the entities it is searching for, apart from the information conveyed by its signals. The result of interaction with another entity will be defined by the properties of that entity and the particular signal, but is unspecified.

Entity 7
Responder. A special type of Signal Receiver that responds to a signal by generating another signal. The intended use of Responders is to represent the signal reflecting properties of an entity.

Entity 8
Sensor Owner. An entity that can own a sensor. Such an entity must know how to issue commands to the sensor and possibly query the sensor for information about it's operational status. Because this ability is a feature of all types (e.g., passive, bistatic) of sensors, we do not specify this entity to be an *active* sensor owner. The owner of a sensor must also have position information.

Entity 9
Sensor User. An entity that can accept the information output by a sensor. Such an entity may also know how to ask the sensor for information. Because all types of sensors will output similar information to their users, and because the abstract model should accommodate all types of sensors, we do not specify this entity to be an *active* sensor user

Entity 10
Referee. An entity that arbitrates the activities of other entities. The Referee enables encapsulation by providing the means for entities to get the information they need to implement behavior, while keeping that information hidden until it is needed. Hiding of this information is essential to long term software stability, and greatly simplifies the structure of other simulation entities. Thus, when the signal needs to know what entities it will interact with in its lifetime, it must ask the referee.

D. DISCRETE EVENT DESIGN

Because we are developing the conceptual model for eventual implementation in a discrete event simulation, we must define the events for the entities we have defined. We will use event graphs, due to Schruben [Schruben, 1983], to graphically describe the processes being modeled. Event graphs are directed graphs in which the nodes represent events, and the edges represent scheduling. A concise introduction to event graphs can be found in

[Buss, 1995]. The graphs shown here are necessarily incomplete due to their generality. Implementations of concrete sensor examples will add the details, such as time delays and scheduling conditions that are necessary for the particular implementation.

1. The Active Sensor Entity

The active sensor itself has four events that are defined by the inheritance relationships shown in Figure 2.2 and one additional event, as shown in Figure 2.3. The implementation of inherited behaviors may be specified in parent classes or in the particular sensor being implemented.

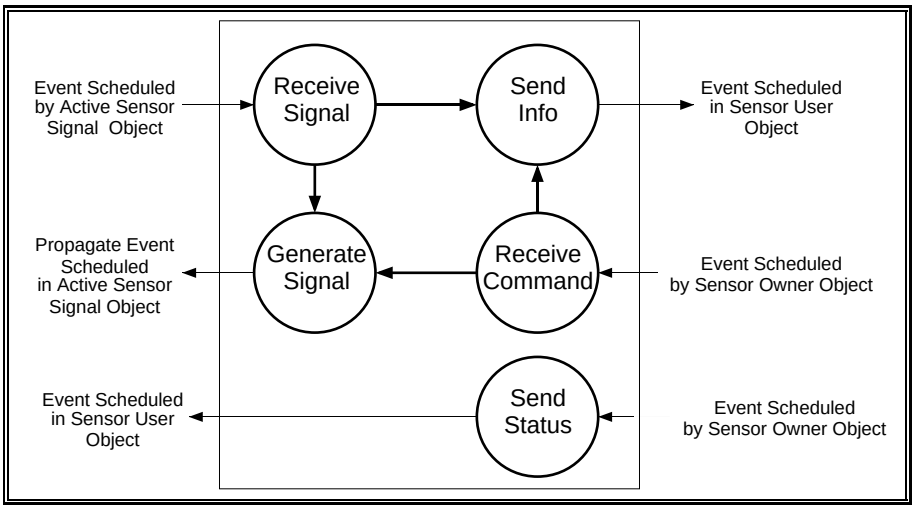


Figure 2.3. The generalized event graph for an active sensor. Specific sensors will have conditions, timing rules and parameters on the arcs. Specific sensors will also have state variables on the nodes, with rules for updating them.

Behavior 1 (Active Sensor)
Generate Signal. Instantiate a signal object of a particular kind, establish it's parameters and tell it to propagate.

Behavior 2 (Signal Receiver)
Receive Signal. Accept a returning signal, extract data from it, and process that data.

Behavior 3 (Sensor)
Send Information. Give the information gained through processing signals to a user.

Behavior 4 (Sensor)
Send Status. Process a request from the owner to report operating status.

Behavior 5 (Commandable)
Receive Command. If the command is one understood, then carry it out, otherwise, do nothing.

2. Active Sensor Signal

The active sensor signal only has one inherited event and one event of its own, as depicted in Figure 2.4.

Behavior 6 (Signal)
Propagate. Ask the referee for a list of the entities that might be encountered based on signal properties. Then, tell each entity in the list to reflect the signal. Depending on the resolution of the particular signal model, signal parameters given to the reflecting object may be calculated based on environmental conditions, which are obtained via the referee.

Behavior 7 (Active Sensor Signal)
Return to Sender. Use environmental information provided by the Referee to determine properties upon arrival back at the sensor. Tell the sensor to receive the signal. Note that the returning signal is new instance of the signal type originally propagated by the sensor.

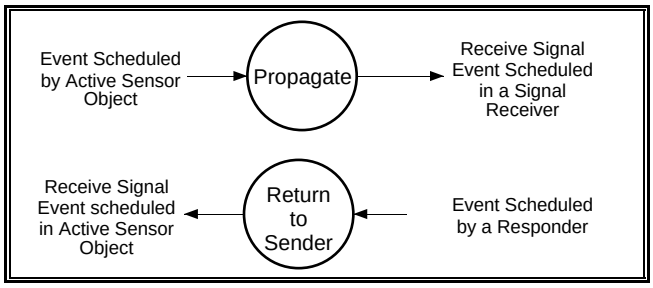


Figure 2.4. Generalized event graph for the abstract sensor signal entity.

3. Responder

Responders have only the one event inherited from Signal Receiver, which is as shown in Figure 2.5.

Behavior 8 (Signal Receiver)
Receive Signal. Receive a Signal and create a new one. The new Signal properties will be based on the properties of the first, and the properties of the Responder define the details of the interaction. Send the new signal back to it's point of origin by telling the new signal to return to sender.

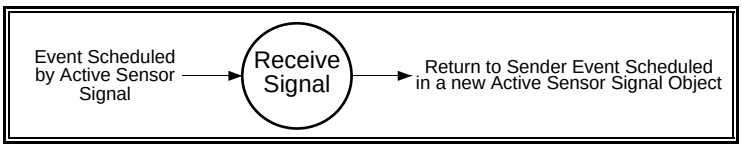


Figure 2.5. Generalized event graph for the abstract sensor signal reflector entity.

4. Sensor Owner

Figure 2.6 shows the single event for the Sensor Owner.

Behavior 9 (Sensor Owner)
Receive Sensor Status. Receive a message from a sensor containing the Sensor's operational status. This may trigger other, unspecified actions.

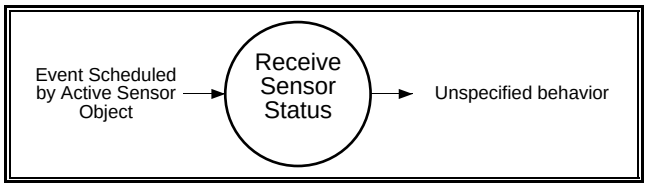


Figure 2.6. Generalized event graph for the abstract sensor owner entity.

5. Sensor User

Since Sensors might provide information without warning, or could experience a delay in satisfying a request for information, the Sensor User needs a method for receiving information, as shown in Figure 2.7.

Behavior 10 (Sensor User)
Receive Sensor Information. Receive a message from a sensor containing sensor contact information. This may trigger other, unspecified actions.

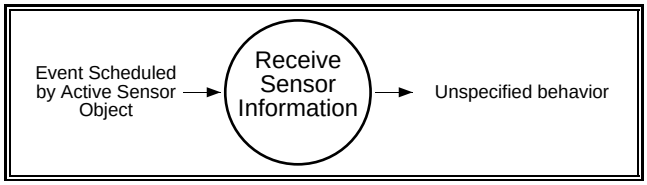


Figure 2.7. Generalized event graph for the abstract sensor user entity.

In the next section, we translate the abstract objects and activities into a programming interface behind which the actual behaviors will be implemented.

E. ABSTRACT IMPLEMENTATION

Our implementation is in the new computer language, Java, from Sun Microsystems [Gosling and McGilton, 1996]. The Bibliography lists books and documents found on the World Wide Web which are useful in learning about the language. Since Java's syntax is similar to C and C++, anyone with a passing familiarity with either of those languages should be able to follow the source code examples shown in this chapter and the next.

Java, like any good object oriented language, can directly represent abstractions such as those we have described. The Java constructions to do this are called interfaces and abstract classes. An interface is a Java construct that defines constants and specifies methods that must be implemented by any class that claims to conform to the interface. Interfaces support single and multiple inheritance from other interfaces. Unlike an interface, an abstract class may implement some of the methods it declares. Abstract classes are always part of a single-inheritance tree, but, like concrete classes, may implement any number of interfaces. Neither interfaces nor abstract classes may be instantiated.

Defining the interfaces will complete the design of the abstract sensor component and provide the foundation for concrete examples. Complete code listings for the interfaces presented here can be found in Appendix A. The completed, though still general, event graph for the abstract model is shown in Figure 2.8 on page 15.

1. The Sensor Interfaces

The inheritance tree shown in Figure 2.2 on page 9, contains four interfaces to be implemented by any Active Sensor. Figure 2.9 on page 16 expands the pertinent portions of Figure 2.2 to include the method declarations in each interface. Due to inheritance, the Active Signal interface specifies five methods that must be implemented by any object that claims to implement that interface:

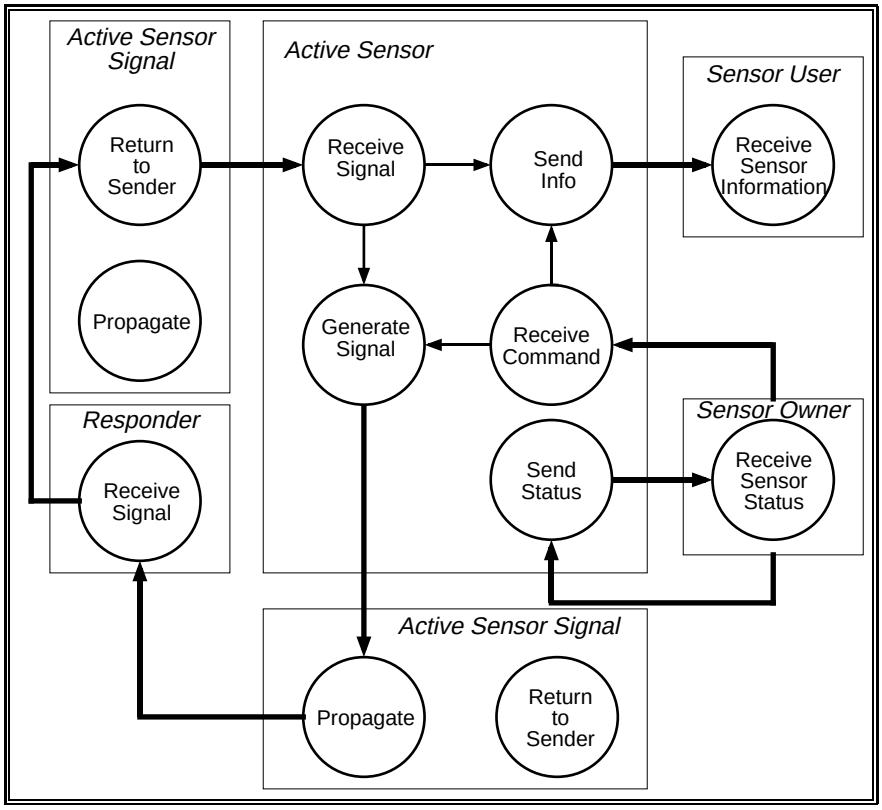


Figure 2.8. Complete generalized event graph for abstract sensors.

```
public void receiveSignal ( Signal      s );
public void receiveCommand( Command     c );
public void sendInfo      ( SensorUser  u );
public void sendStatus    ( SensorOwner o );
public void generateSignal();
```

2. The Signal Interfaces

As with the sensor interfaces, we expand Figure 2.2 to show the methods declared in each interface in Figure 2.10 on page 16. The resulting methods required of any Active Signal are:

```
public void propagate     ();
public void returnToSender();
```

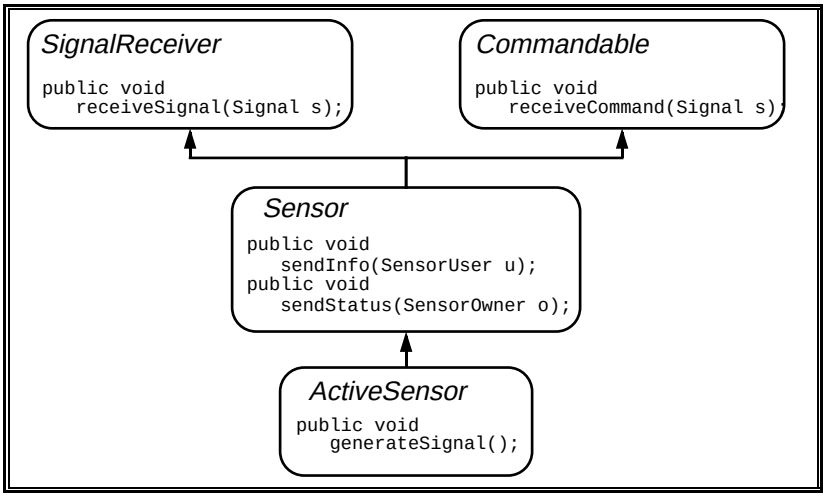


Figure 2.9. Expanded Sensor Interface Inheritance Tree.

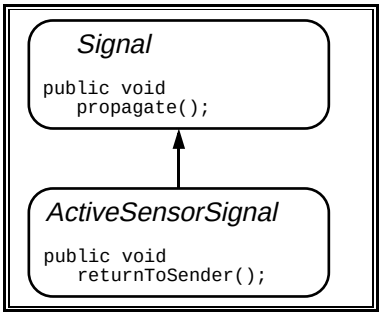


Figure 2.10. Expanded Signal Interface Inheritance Tree.

3. The Responder Interfaces

Again, we expand Figure 2.2 to show the methods declared in the Responder interface in Figure 2.11. The resulting methods required of any Responder are:

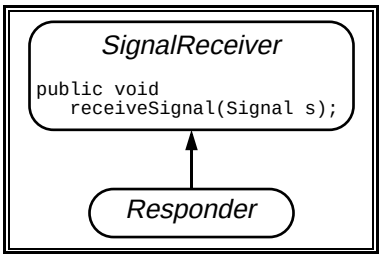


Figure 2.11. Expanded Responder Interface Inheritance Tree.

```
public void receiveSignal();
```

4. Other Interfaces

The remaining interfaces are not members of an inheritance hierarchy. The Sensor Owner interface declares one method:

```
public void receiveSensorStatus( SensorStatus s );
```

and the Sensor User interface declares another:

```
public void receiveSensorInfo( SensorInfo i );
```

III. DEMONSTRATIONS

In this chapter we develop several models to demonstrate the use of the abstract sensor component in a number of modeling contexts. The demonstrations are intended to show that the abstraction can serve in both high and low resolution models to support very different types of analysis.

The descriptions here focus on the sensor component, and the large body of supporting code will not be discussed at length here. In Chapter IV, we will discuss a supporting framework that should be developed to make components such as the sensor generally useful.

A. SCENARIO

The demonstrations model a so-called barrier search scenario. The barrier search was chosen because it has mature analysis techniques with which to verify the new model. Additionally, the barrier search does not require a sophisticated position or motion model and is straightforward to implement.

In a barrier search problem (see Figure 3.1), there are two players: a target and a searcher. The target attempts to travel down a channel with speed u. The searcher creates a barrier across the channel by traveling back and forth between two points, A and B so that its sensors traverse the entire channel width, L. The searcher has speed v.

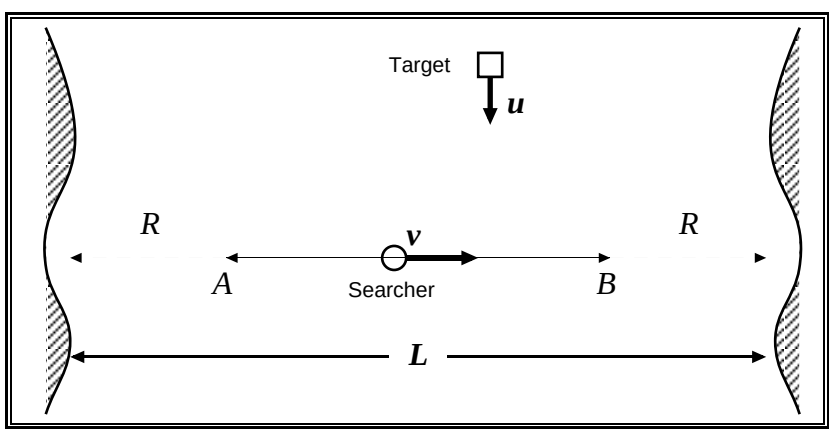


Figure 3.1. Barrier Search Scenario. A searcher with a sensor of effective radius R travels back and forth between points A and B with speed v. The target moves down the channel with speed u.

The demonstration models discussed in the next two sections model the barrier search. In both models the searcher begins on the left and patrols the barrier until some number of targets have attempted to cross. Targets arrive at a location which is uniformly distributed over the length of the barrier. Target arrivals occur when the previous target is detected or successfully penetrates the barrier. Target and searcher have fixed speeds throughout the experiment.

The barrier search is modeled with two sensor component models using the component developed in Chapter II. The first is a deterministic "Cookie-Cutter" glimpsing sensor. The second is a glimpsing radar based on the theoretical physics of radar.

B. COOKIE CUTTER SENSOR

Cookie cutter sensors are the foundation of search and detection theory in Operations Research. The cookie cutter sensor has radius R, and detects a target with certainty if the range, r, to the target is less than R, as shown in Figure 3.2 and in Equation 3.1.

$$P_d = \begin{cases} 1 & \text{if } r \leq R \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

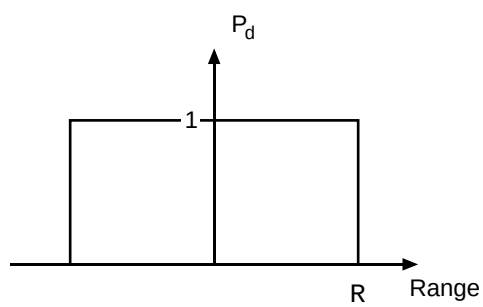


Figure 3.2. Probability of Detection for a Cookie Cutter Sensor.

1. Model

The barrier search scenario using cookie cutter models are developed analytically in [Washburn, 1989] and [OASG, 1977]. The perspective taken in this development is operational; the goal being to determine the probability of detection given values for the parameters. We assume the target arrives at the barrier at a position which is uniformly distributed over the channel width, L. The searcher begins to traverse the channel from

the left when the first target arrives. In this circumstance, the searcher has a maximum probability of detecting the target as defined in Equation 3.2.

$$P_d \leq \min\left\{1,\ \frac{2R}{L}\sqrt{1+\left(\frac{v}{u}\right)^2}\right\} \tag{3.2}$$

The long run probability of detection is more closely approximated by Equation 3.3.

$$P_d \approx \begin{cases} 1-\left[\frac{1}{\lambda(\lambda+1)}\left(\lambda-\frac{\sqrt{r^2+1}-1}{2}\right)^2\right], & r \leq 2\sqrt{\lambda\left(\lambda+1\right)} \\ 1, & r > 2\sqrt{\lambda\left(\lambda+1\right)} \end{cases} \tag{3.3}$$

where,

$$\begin{aligned} r &= \tfrac{v}{u} \\ \lambda &= \tfrac{(L-2R)}{2R} \end{aligned}$$

2. Simulation

To simulate the scenario with the sensor component developed in Chapter II, concrete classes which implement the sensor component interfaces must be constructed. Listings for these classes can be found in Appendix B. Listings for supporting code to handle discrete event simulation and entity motion can be found by following links from `http://dubhe.cc.nps.navy.mil/~ahbuss` on the World Wide Web.

The cookie cutter sensor is simple to implement using the sensor component model. Four classes must be defined: a Sensor, a Signal, a Responder and a Platform. As shown in Figure 3.3, these entities correspond to the ones in the abstract model. In some cases, the concrete classes inherit from classes or implement interfaces that have not been discussed; Listings for these classes are available at `http://dubhe.cc.nps.navy.mil/~ahbuss`, and are discussed in greater detail in [Buss and Stork, 1996].

a. CCActiveSensor Class

The cookie cutter active sensor implements the methods of interfaces Commandable and ActiveSensor, as well as the methods required of any simulation entity. It responds to only two commands: one to activate, and one to passivate the sensor. When the sensor is activated, it schedules a `generateSignal` event to occur after a delay. The delay is set when the sensor is instantiated with an argument to it's constructor.

When the `generateSignal` event occurs, the sensor instantiates an object of class CCActiveSensorSignal, passing as parameters to the constructor a reference to the sensor itself, the position at generation time and the maximum range of the signal. The

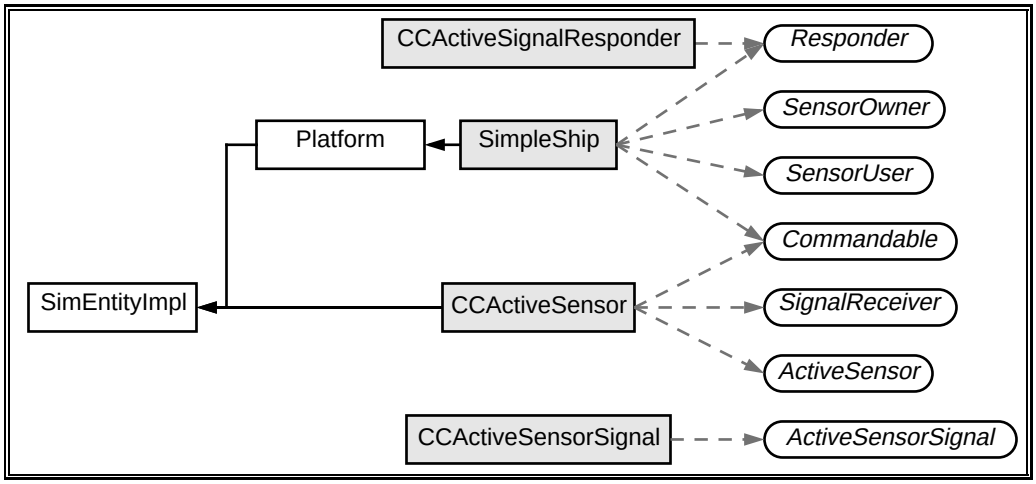


Figure 3.3. Cookie Cutter Model Inheritance Graph. Entities that schedule events which pass simulation time inherit from the Simkit [Buss and Stork, 1996] class SimEntityImpl. The sensor component interfaces from Chapter II are implemented by the entities to establish the sensor-related behaviors. The class SimpleShip implements the Responder interface because it may own an instance of a Responder, namely the CCActiveSignalResponder.

signal is then told to propagate. Finally, a `generateSignal` event is scheduled to occur after the delay time.

When the sensor receives a signal, it checks to ensure the signal is it's own, and records the fact that a detection was made. It also cancels any further signal generation, effectively ending the simulation trial.

b. CCActiveSensorSignal Class

The signal class for the cookie cutter sensor simply stores the information passed to it's constructor: who made it, where was it made, and what is its maximum range. When told to propagate, the signal asks the Referee for a list of the players that respond to signals. For each player in the list, the signal gets its position and calculates the range from the originating point. If that range is less than the maximum range of the signal, the player is told to `receiveSignal`.

A responder creates the signal with a different constructor which takes the original signal together with a position as arguments. Information is copied from the original signal, and no further reference to that signal is retained.

When told to `returnToSender`, the signal tells its creator (the CCActiveSensor instance that created the original signal) to `receiveSignal`.

c. CCActiveSignalResponder Class

The responder only has one method, `receiveSignal`. When this method is called by a signal, this responder simply instantiates a new signal of the same type, using the position of the responder's platform and the original signal as arguments to the signal constructor. The new signal is then told to `returnToSender`.

d. Main Program

In Java, all code must be part of some class, so there is a main class in addition to the sensor related classes that runs the model. The method of batch means [Law and Kelton, 1991] is used to find the fraction of trials in which the target is detected. The algorithm followed by the main class to run the simulation is as follows:

1. Instantiate two objects of class SimpleShip: searcher and target
2. Instantiate a CCActiveSensor object with maximum range R, and add it to the searcher
3. Instantiate a CCActiveSignalResponder object and add it to the target
4. Instantiate two data collectors, one for batch results and one for trial results
5. For $i = 1$ to $i =$ number of batches
   (a) reset the trial data gatherer
   (b) Issue a patrol command to the searcher
   (c) For $i = 1$ to $i =$ number of trials per batch
      i. Place the first ship at the left-hand barrier waypoint position
      ii. Place the second ship at a position uniformly distributed between the left and right ends of the barrier, and north of the barrier a distance R
      iii. Issue a course/speed command to the target
      iv. Issue an activate command to the searcher
      v. Start the simulation clock
      vi. Record detection or non-detection in the trial data collector
   (d) Record the average number of detections from the batch in the batch data collector
6. Output results

The program is invoked (under UNIX), with the following command line:

```
java CCBarrier <batches>          <trials per batch>
               <search speed>     <target speed>
               <sensor range>     <glimpse interval>
               <barrier length>   [seed]
```

or

```
java CCBarrier <filename>
```

Where,

`batches`	= Number of batches to run
`trials per batch`	= Number of target attempts per batch
`search speed`	= Searcher speed in knots
`target speed`	= Target speed in knots
`sensor range`	= Sensor range in nm
`glimpse interval`	= Time between sensor glimpses in seconds
`barrier length`	= Width of the channel in nautical miles
`seed`	= Long integer random seed (optional)
`filename`	= Name of an input file

If a filename is specified, that file has a single line containing the arguments as listed, separated by white space.

Alternatively, this program is available as an applet on the World Wide Web that can be run in a Java enabled web browser, such as Netscape 3.0 or Microsoft Internet Explorer 3.0. The applet version provides a graphical interface with labeled fields to fill in and produces the same output as the program. The applet can be found by following links from `http://dubhe.cc.nps.navy.mill/~ahbuss/`.

3. Results and Verification

The model proved to be insensitive to the value for the glimpse interval parameter up to about 960 sec. However, runtime is dramatically reduced by using larger values. The results below correspond to the following set of parameters:

Number of batches	= 500
Trials per batch	= 50
Search Speed	= 12(knots)
Target Speed	= 7(knots)
Sensor Range	= 10(nm)
Glimpse Interval	= 480(sec)
Barrier Length	= 80(nm)
Seed	= 2116429302

This run resulted in an average detection rate of 0.4706 with variance of the mean 0.0044. This gives a 95% confidence interval for the average detection rate of $(0.340, 0.601)$. A histogram of the batch results of are shown in Figure 3.4. To verify the results, Equation 3.3 was solved with the same parameters:

$$\begin{aligned} v &= 12(\text{knots}) \\ u &= 7(\text{knots}) \\ R &= 10(\text{nm}) \\ L &= 80(\text{nm}) \end{aligned}$$

resulting in $P_d \approx 0.476$, which falls within the confidence interval of the simulation results.

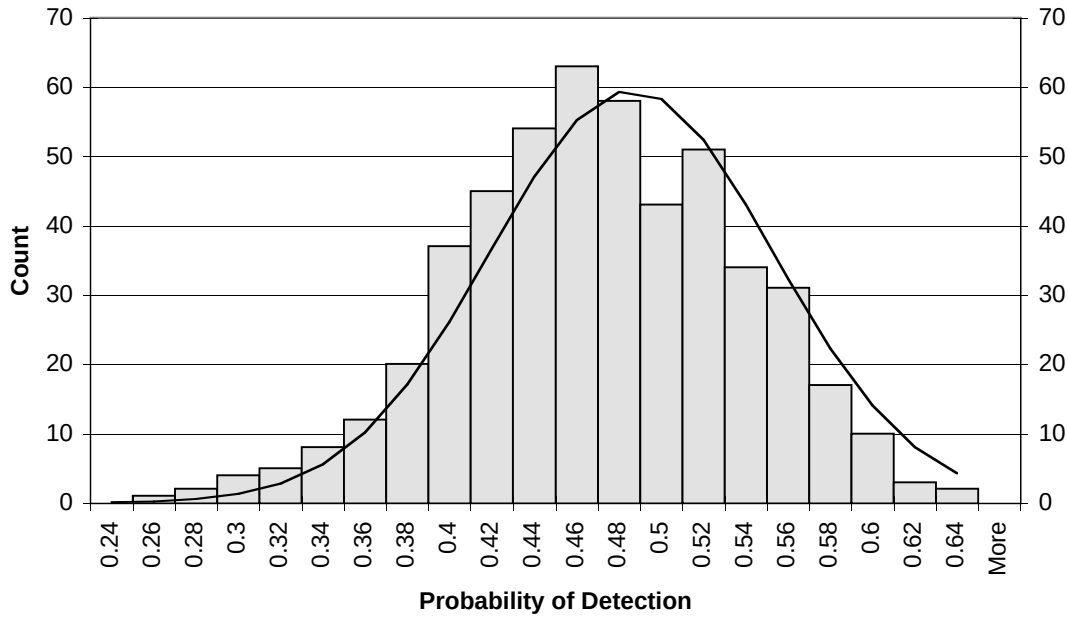


Figure 3.4. Cookie Cutter Model Results. This histogram counts the number of batches resulting in a given probability of detection. The probability of detection should be normally distributed, as shown by the pdf plotted with the histogram.

C. MONOPULSE RADAR

Missile countermeasures must be evaluated with a more sophisticated sensing model than the Cookie Cutter described above. The Cookie Cutter sensor is sufficient for the barrier search because the MOE is simple, i.e., the probability that the searcher detects the target. With countermeasures, the MOE changes from a simple determination of detection or non-detection, to one of discrimination, i.e., the probability that the sensor correctly identifies it's target when several are present.

We have two goals in this demonstration of a Monopulse Radar model. First, we wish to reuse the barrier scenario to demonstrate the ability to reuse a large body of supporting code. In fact, the main program has only minor modifications, which we will highlight below. Second, we wish to develop a sensor model that operates at a higher level of detail, capturing some of the physics of radar operation.

1. Model

The model is still highly stylized—the sweep of the radar is not modeled explicitly, but simulated with an omnidirectional signal containing only the number of pulses that would be incident on the target if the signal was a sweeping beam. Signal generation is timed to coincide with the sweep rate to approximate the number of glimpses a directed, sweeping radar would get. Finally, the signal used simulates a pulse train, rather than individual pulses.

To model the physics of waves, the model is based upon The Radar Range Equation (Equation 3.4), which can be found in any book on radar.

$$\frac{S}{N} = \frac{PGA\sigma_{rcs}En}{\left[4\pi R^2\right]^2 \left[\sigma^2_{noise} + \sigma^2_{clutter}\right] \left[L_{sys}L_{atm}\right]} \tag{3.4}$$

where,

$\frac{S}{N}$	= Signal to Noise ratio
P	= Average transmitted power
A	= Antenna aperature
G	= Antenna gain
σ_{rcs}	= Target radar cross section
E	= Integration efficiency
n	= Number of pulses integrated
R	= Slant range to target
σ^2_{noise}	= rms noise power
$\sigma^2_{clutter}$	= rms clutter power
L_{sys}	= System losses
L_{atm}	= Atmospheric losses

Equation 3.4 is rearranged in Equation 3.5 into terms that will be useful in the implementation.

$$\frac{S}{N} = \left[\frac{P}{4\pi R^2}\right] \left[\frac{\sigma_{rcs}}{4\pi R^2}\right] \left[\frac{AEn}{\sigma^2_{noise}}\right] \tag{3.5}$$

Where we have removed variables that will not be modeled.

The first term of Equation 3.5 is the signal power incident upon the target. The second term, when multiplied by the first produces the signal power returning to the radar

receiver. Finally,when the third term is applied, the result is the signal to noise ratio (SNR) as processed by the radar system.

2. Simulation

Several assumptions are made in the simulation. First, we assume a constant threshold SNR for the radar system below which a returning signal will not constitute a detection. Second, the target is assumed to be a so-called Swerling case 3 target [Swerling, 1976], which models a radar reflector that is dominated by a single large scatterer and many smaller independent scatterers. This model simulates a large scatterer surrounded by ocean clutter. Receiver noise is assumed to be normally distributed. Finally, we will ignore atmospheric and system losses.

As previously stated, the simulation will use a glimpse approach similar to the original Cookie Cutter model. To be consistent with the assumptions of the Swerling target model, probabilistic independence between successive glimpses is assumed. The source code listings for this demonstration can be found in Appendix C.

a. MonopulseRadar class

The MonopulseRadar class is similar to the CCActiveSensor, requiring modification of only two methods. First, the `generateSignal` method generates a new signal type, PulseTrain, rather than the CCActiveSensorSignal. Second, the `receiveSignal` method applies the final term of Equation 3.5 to the returning signal and compares the result to a fixed threshold value to determine if a detection has been made.

b. PulseTrain class

The PulseTrain class is a new Signal type, implementing the ActiveSensorSignal interface. To incorporate the physics of wave propagation, reception by potential contacts is no longer instantaneous. Instead, the PulseTrain class inherits from the SimEntity class and defines simulation events to schedule signal arrival at the potential contact after a delay computed from the range to the contact. Similarly, the returning signal arrives at the sensor after a delay.

The PulseTrain is instantiated with all the information needed for subsequent calculations, and told to propagate as before. After retrieving the potential contact list from the Referee, it calculates the range to each potential contact, and if the contact is within the range scale setting of the radar, it schedules an arrival based on the range and the speed

of light. When the arrival event occurs, the PulseTrain calculates the power incident upon that Responder, and calls the responder's `receiveSignal` method.

Returning PulseTrains are instantiated by the Responder with the σ_{rcs} of the Responder as an argument. Term two of Equation 3.5 is applied, and the result stored. The range to the sensor that created the original signal is calculated and arrival is scheduled after the appropriate delay. When the arrival event occurs, the PulseTrain calls the sensor's receiveSignal method.

c. PulseTrainResponder class

The PulseTrainResponder is instantiated with a mean value for σ_{rcs}. When the `receiveSignal` method is called, it instantiates a copy of the received signal with a value of σ_{rcs}, which is used in term two of Equation 3.5. The value used is a chi-square random variate with four degrees of freedom which models a target with a single dominant radar scatterer and several smaller scatterers.

d. Main Program

The main program is virtually identical to the Barrier Search program of the previous demonstration. The differences are:

1. The searcher is equipped with a MonopulseRadar instead of a CCActiveSensor.
2. The target is equipped with a PulseTrainResponder rather than a CCActive-SignalResponder.

3. Results

The Monopulse Radar model was run with the same parameters as the Cookie Cutter model. Additional parameters used for the run correspond to the operational parameters of the SPS-68 surface search radar as published in [Streetly, 1996]:

$$\begin{aligned} P_{ave} &= 10(\mathrm{W}) \\ A &= 1.37\ \mathrm{m}^2 \\ E &= 0.96 \\ \sigma_{noise} &= 6(\mathrm{dB}),\ \text{average} \\ N &= 18(\text{pulses/glimpse}) \end{aligned}$$

The model run resulted in an average detection rate of 0.4540 and variance 0.00451 (see Figure 3.5). This produces a 95% confidence interval of (0.3224, 0.5857). As expected, the confidence interval for the radar is lower than the perfect cookie cutter, but still contains the theoretical result of Equation 3.3.

Figure 3.5. Monopulse Radar Model Results. This histogram counts the number of batches resulting in a given probability of detection. The distribution of an ideal sensor is also plotted.

IV. APPLICATION

The models presented in the previous chapter demonstrate the usefulness of the abstract model developed in Chapter II. The component model we have developed is applicable to the missile countermeasure problem, but not immediately useful in the absence of a larger framework. In particular, it was not possible to develop a motion model sophisticated enough for the missile-ship engagement scenario in the time available for this research.

Since time did not permit a more directly applicable implementation, we will discuss what is needed to use the sensor component in more sophisticated future models. We do so by sketching out the other components that would make up a framework. We will focus, as we did in Chapter II, on abstractions.

The framework envisioned in this chapter was inspired by an existing library, the Naval Postgraduate School Platform Foundation [Bailey, 1995]. Recent research employing the Platform Foundation, including a thesis related to the problem of simulating radars [Ellison, 1996], has highlighted the usefulness of simulation libraries in developing models quickly. It has validated the concept of reusable and extensible computer code, and provided important insights into the problem of designing a true framework. However, the Platform Foundation was developed to be an extensible application, rather than a framework or Application Programming Interface (API).

Readers familiar with the Department of Defense High Level Architecture (HLA) [DMSO, 1996], will notice some differences in design philosophy between it and the framework we propose. Whereas HLA is being designed to facilitate cooperation between existing and future models, the framework proposed here is designed to facilitate rapid development of small to medium sized models for a *specific purpose*.

While the HLA object model is a sound approach to an immense problem, the modeler's ability to hide information is limited by the requirement to broadcast information about an entity without concern for how that information will be used. The broadcast mechanism is only necessary to support cooperation between legacy and future models, but adopting this mechanism as the centerpiece for model cooperation forces future models to conform when they could otherwise benefit from the full benefits of object oriented design.

A. DESIGN GUIDELINES

The proposed framework, the Abstract Military Modeling Toolkit (ammt), was conceived for a much smaller problem than the HLA, although it shares many of the same difficulties. The intended users of ammt are organizations like NPS, SEWS, and test and

evaluation support organizations that have a need for simulation models but limited resources. To this end, design was guided by a few general principles:

- **Language.** ammt should be implemented in a portable, easy to learn language with a larger programmer base. Compilers for this language should be freely available.
- **Object Oriented.** ammt should be designed as an object oriented class library to maximize the benefits provided by software engineering.
- **Abstract.** At its core, ammt should be a set of abstract classes or interfaces. This abstract core can establish a sound design base that does not impose the restrictions of a concrete implementation. Any number of concrete implementations could be developed for particular modeling contexts.
- **Distributed.** ammt should be designed with distribution in mind, and eventually should directly support both distributed object libraries and distributed execution.

The remainder of this chapter examines a few critical areas that should be addressed by ammt. The list is incomplete, but sufficient to guide a full design and implementation. We will discuss three specific areas of interest for military modeling:

- **Arbitration.** An mechanism to effect the principle of least privilege, or information hiding.
- **Location.** A generalized model for locating entities in physical space.
- **Movement.** A generalized model for physical movement.

B. ARBITRATION

Simulation of real entities, such as soldiers, ships and aircraft, involves managing a complex and unforeseen set of interactions between those entities. In an object oriented program, it is desirable to use an event model, similar to that used in modern graphical user interface (GUI) programming. That is, entities exist with some state, and wait for events to occur. In a GUI, an entity such as a button, merely exists—it is only the externally generated event of a mouse click that causes the button to actually do anything.

A simulation analogy to the GUI Button is an entity, such as a ship. A simulation model of the ship exists with some state until an event occurs that results in state-changing

action by the ship. For example, the ship may be cruising at some course and speed and receive a communication message that causes it to maneuver. Or, the ship may encounter an environmental interaction, such as running aground in shallow water.

This approach has many advantages. For instance, the designer of the ship model need not be omniscient. So long as the ship object responds to the appropriate events, it need not know how to determine whether or not it has run aground. This allows information to be maintained in a safe place, separate from the ship. Just as the ship object has no knowledge of the contents of a message before the message arrives, it should have no knowledge of the impending grounding.

Software stability and extensibility is enhanced by hiding information as described above. Book keeping of information, such as the existence of shallow water at some location, is not duplicated across all entities that might be interested, so the chances for error in future code is reduced. The memory requirements of entities are also reduced, though perhaps at a price paid in execution time.

Additionally, hiding of information makes "cheating" more difficult. Cheating arises when a programmer, faced with a deadline, has "back door" access to information that would normally be considered private to some other object. The "quick and dirty" solution is to use the back door, with full intentions of correcting the poor code later. Later, when the implementation of the object that "owns" the information is changed, the cheating code breaks because the quick and dirty solution was forgotten.

These issues motivate a structure that maximizes the hiding of information. However, the shallow water example above serves to raise the question of where information should reside. Who *should* know where the shallow water is? The answer adopted in this preliminary design of the ammt lies in arbitration of such interactions by two simulation entities that will be part of every model constructed with ammt: The Referee, and the Environment.

1. Referee

The Referee is envisioned as a god-like entity that is kept informed of everything that transpires, although it is not necessarily aware of information internal to simulation entities. All interactions between other entities will in some way involve the Referee. The sensor component developed in Chapter II serves as an example: Signals interact with the Referee to determine what other entities will be affected by the signal.

Such a scheme requires all players in the simulation to register with the Referee when they are created and to unregister when they are destroyed. It also requires players

to register their activities, or state changes, so the Referee can schedule events, or opportunities, that are unforeseen by the player. The Referee will make use of another simulation entity, the Environment, to determine some such events, and to provide information to players about environmental conditions.

2. Environment

The Environment class holds information about conditions in a physical region. It can schedule state changes in itself, such as changes in the weather or season. There may be several Environment instances in a simulation, particularly if the simulation models activity in a large physical region.

The Environment class would also be the place where topographical and domestic information, such as soil type and vegetation, would be maintained. Players will not directly interact with the environment, instead, they interact with the Referee, which gets the information from the Environment and passes it along to the player. This indirection is imposed to allow construction of components who's only direct link to the rest of the simulation is the Referee.

3. Opportunities

An Opportunity embodies the notion of something that might happen. Opportunities are the sole responsibility of the Referee, who schedules and cancels them according to it's own rules.

As an example, assume there are two ships, each with a course and speed that will eventually lead to collision. Assuming the ships have no sensors, they have no knowledge of each other, and so they are unable to interact. If this were the complete model, nothing would ever happen.

But, in ammt the ships register their position, course and speed with the Referee when they maneuver. The Referee then calculates the closest point of approach (CPA) and notices that the ships will collide if they remain on course and speed. This situation causes the Referee to schedule an Opportunity for the two ships at the time of CPA. If the ships do not maneuver, the Opportunity will occur at it's scheduled time, sending a message to both ships that they have collided. If one of the ships maneuvers before the Opportunity occurs, then that Opportunity is canceled, the new states are examined by the Referee, and a new Opportunity might be scheduled.

The Opportunity concept is general enough to handle all interactions that are unforeseeable by simulation players. However, this mechanism will potentially result in the

scheduling of a large number of events. Further, because the Referee must make opportunity determinations whenever any player changes state, the Referee could easily become a large and unwieldy burden on the model.

C. LOCATION

The idea of location is particularly vexing for simulation modeling. Many systems exist for specifying an object's physical location in space, examples include the zones of TACWARS, the network node locations of JWEAPS, grid squares, hexagons and continuous coordinates. Further, simulation players have more than one kind of location. For example, a ship has an organizational location in a fleet, a soldier has a location in a chain of command, and a radar contact has a location in a radar's parameter space. We will discuss only one such location, physical, but the final design should address many.

1. The Position Abstract Class

Because ammt is to be abstract, it is possible to delay some of the problems related to positioning. We can define an abstract class, Position, that contains no data, but embodies the notion of a displacement vector. Entities whose state includes physical position would then have a member variable of type Position. Concrete entities would necessarily be designed to work with certain specializations of Position, but the abstract entities of the ammt do not need that specific information.

Because Position is abstract, entities that work with Positions cannot know how to perform operations on them. Instead, the abstract class, Position, should declare a number of standard operations that will be needed by simulation entities regardless of the specific implementation. These operations return either boolean values, or values in the units of the positioning system. Examples of such operations are:

- add a Position and a Displacement
- subtract a Displacement from a Position
- find the distance between two Positions
- find the charted distance between two Positions
- find the slant range between two Positions
- find the direction from one Position to another
- find the elevation angle from one Position to another

- multiply a Position by a number
- divide a Position by a number
- determine if one Position is within a region defined by other Positions
- determine if one Position is between two bearings from another Position
- determine if one Position lies to the north of another Position
- determine if one Position lies to the east of another Position

It is often useful to identify entities who have a particular attribute, and define operations that can be performed on such entities. In Java, the common practice is to define an interface with the adjective form of the word that describes the attribute. Following this practice for objects that have the Position attribute, we call the interface "Positionable".

2. The Positionable Interface

Positionable entities are those that have a position in space. Like the attribute itself, the interface imposes no specific system of positioning, but merely requires support for certain operations.

Since Positionable objects are simply objects that have the Position attribute, the simplest definition the Positionable interface would specify only accessor methods. An accessor method is one that allows setting and getting of the attribute. In Java, we could write the entire interface as follows:

```
public interface Positionable
{
   public void position( Position p);    // setter
   public Position position();           // getter
}
```

However, this system allows any object to get the Position of any Positionable object, perform operations on that Position, and set the the Positionable objects Position to a new value. This is undesirable, since entities such as ships should not be moved around by entities such as radar signals.

Instead, the operations defined for Positionable objects can parallel the ones for Position, with a few exceptions:

- find the distance between two Positionable objects

- find the charted distance between two Positionable objects
- find the slant range between two Positionable objects
- find the direction from one Positionable object to another Positionable object or Position
- find the elevation angle from one Positionable object to another Positionable object or Position
- determine if a Positionable object is within a region defined by a set of Positions
- determine if a Positionable object is between two bearings from another Positionable object or Position
- determine if a Positionable object lies to the north of another Positionable object or Position
- determine if a Positionable object lies to the east of another Positionable object or Position

In implementation, these methods are simple redirections to the parallel method in the Position instance variable. Notice that these methods only provide information, and not the ability to change the attribute.

D. MOVEMENT

Physical movement can be modeled in many ways, from Newtonian physics that models forces on objects to affect motion, to highly stylized instantaneous jumps from one location to another. The ammt framework should neither impose nor exclude any such motion model. Furthermore, each concrete example of physical positioning, such as grids or continuous coordinates, will require at least one corresponding system for movement. In the abstract, however, it is only necessary to declare the operations.

As for location, above, there should be abstract objects to contain the state information, and interfaces to distinguish objects that have the new attribute. Using the terms of object oriented design, an object that can move "is a" Positionable object. Hence, new state variables and behaviors should be based on the existing state information. Movement requires at least one additional state variable, Velocity.

1. The Velocity Abstract Class

Velocity is an abstract class that encapsulates the notion of changing position as a function of time. Like the Position class, Velocity should provide for operations so that other objects can treat it abstractly:

- add two Velocities
- subtract two Velocities
- find the angle between two Velocities
- multiply a Velocity by a number
- divide a Velocity by a number
- determine the magnitude of a Velocity

2. The Moveable Interface

Moveable objects have both a Position and a Velocity. The Moveable interface inherits from Positionable, and adds methods for accessing the motion state of the implementing object:

- determine the speed of a Moveable object
- determine the direction of a Moveable object's motion
- determine the relative velocity of one Moveable object with respect to another

E. FURTHER DEVELOPMENT

The issues addressed in this chapter are only a beginning. The abstract framework components described here only comprise a part of the framework that is required, and even those will require several design iterations. In particular, the Referee deserves considerable thought and development, since each new paradigm added to the ammt framework will necessitate expansion of the Referee's abilities.

Additional paradigms that should be incorporated into the ammt include, but are not limited to:

- *Motion*

  Most modeling contexts require the notions of the orientation of objects, rotation, and angular velocity. Constraints on motion based on the type of platform are also important, for instance, submarines can't fly.

- *Communication*

  The idea of a communication message should be provided. Early ideas for this follow the Signal/Signal Receiver construction used in the sensor component. The Signal approach allows for modeling of signal interception.

- *Passive Sensors*

  The Active Sensor component developed in Chapter II was designed to be expanded to passive sensors. A critical missing element is generation of signals by entities other than sensors, and coordinating the reception of those signals by the Referee.

- *Weapon Systems*

  Although many weapon systems, such as missiles, can be modeled using the same structure as other platforms, there will be need for weapons that cannot. One example might be directed energy weapons.

  Weapon systems typically have complex supporting systems, such as fire control, which should be generalized for the ammt if possible.

- *Command Structure*

  Military organizations are centered around a chain of command. The chain of command should be modeled by ammt to support decision modeling. Decisions are made based on *available* information, so an operational commander should be modeled to be at a location, or on a platform, and to act only on information available from his sensors and communications equipment.

- *Operational Status*

  Platforms may not always have all of their designed functionality. For instance, a sensor system may fail, making it unavailable until it is repaired. The concepts of failure, damage and repair should be abstract components of ammt.

No code was written that actually implements the design described in this chapter. Instead, it is a sort of designers notebook compiled from the experience of writing the supporting code for the models presented in Chapter III. The ammt code, such as it is, is provided at `http://dubhe.cc.nps.navy.mil/~ahbuss` on the World Wide Web. We look forward to further development efforts.

V. CONCLUSIONS

This research set out to explore the problem of providing simulation support for sensing systems. In the process, we developed some previously unavailable tools in a new computer language, Java. In the course of this research we perhaps found more questions than answers, but the insights our model gives for the sensing problem will help guide future modeling efforts.

We have shown that sensor components can be designed to accommodate a diverse set of modelers. While the sensor components implemented in this thesis do not fit any existing framework, the design methodology and abstract concepts are generally applicable, and independent of any computer language.

We developed a robust and general abstract component model for sensors. We then used that model to construct several customized components at different resolutions that work within the same supporting framework. While our abstract model will certainly benefit from further development, the power of abstract model development is evident.

We have given an example of a method for communication between modelers and computer programmers. The conceptual sensor model of Chapter II requires no programming expertise to understand. However, the form of the abstract component is easily understood and implemented by an Object Oriented programmer. Because programmers discuss their code in similar terms, the modeler who learns about Object Oriented design is also better equipped to understand the programmer's discussions of the implementation.

We have demonstrated an agenda for code and concept reuse. Without such a program, future software development will be slow and costly in an environment of rapidly changing technology and force structure. This will impact all areas of software development, including simulations.

Software design is as important as implementation. Software development is an iterative process, including the software design phase. Because changes in software design impact a potentially huge body of code, we should recognize its importance and ensure that software engineers are involved in the process of constructing simulation models. This is already common practice for large monolithic models, but should also be adopted by organizations that need smaller custom models. The proposed framework, when available, will impose a solid software engineering discipline and could relax the need for software engineering expertise at small organizations.

Clarity and extensibility of models can be more important than run-time efficiency. Computer hardware and runtime is cheap compared to software development. Simulation

models should be written to be as generic as possible, due to unknown and unforseeable needs of future analysis.

System experts should focus on component building, rather than model building, to support simulation throughout the system's lifecycle. *Simulation* experts should focus on modeling. A framework, such as the one proposed, can facilitate such an arrangement with potentially great rewards.

There is a need for simulation support that spans the lifecycle of systems. Military systems remain in service longer than ever before and undergo numerous revisions. Simulation support is required throughout this lifecycle, and should thus be viewed as an integral part of the system.

A *single* model cannot span the entire lifecycle. Furthermore, since modeling requirements vary widely depending on the customer, it is impossible to construct one model that suits all needs. The need for separate custom models necessitates development of components that can serve in more than one model. Perhaps more importantly, reusable components are needed to allow for rapid assembly of supporting code to for use in custom models for new components.

In this thesis we developed models that demonstrate the feasibility of constructing small simulations with a reusable component framework. The approach requires thorough development of components in abstract terms. It is noteworthy that the most important part of the process, design of the model, does not require programming expertise. The abstract nature of the components suggests that the approach could be applied in larger scale simulation models as well, and that development of a more complete framework, such as the one proposed, is worthy of further investigation.

LIST OF REFERENCES

[Bailey, 1995] M. Bailey. The Naval Postgraduate School Platform Foundation. Technical Report NPS-OR-94-005, Department of Operations Research, Naval Postgraduate School, Monterey, CA, 1995.

[Buss and Stork, 1996] Arnold H. Buss and Kirk A. Stork. Simkit: A Toolkit for Simulation on the Internet. Working paper, 1996.

[Buss, 1995] Arnold H. Buss. A Tutorial on Discrete-Event Modeling with Simulation Graphs. In C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman, editors, *Proceeding of the 1995 Winter Simulation Conference*, 1995.

[Cox and Novobilski, 1991] Brad J. Cox and Andrew J. Novobilski. *Object Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 2nd edition, 1991.

[DMSO, 1996] Department of Defense High Level Architecture Object Model Template. Internet URL: ⟨`http://www.dmso.gov/`⟩, August 1996.

[DTIC, 1980] Defense Modeling and Simulation Organization. *Combat Sample Generator User's Manual*, 1980.

[Ellison, 1996] Arron Ellison. Simulation of a Radar Detection Model using the NPS Platform Foundation. Master's thesis, Naval Postgraduate School, 1996.

[Fletcher, 1996] Dr. Charles Fletcher. E-mail correspondence. Ship Electronic Warfare Systems, Naval Research Laboratory, September 1996.

[Gosling and McGilton, 1996] James Gosling and Henry McGilton. The java language environment. Internet URL: ⟨`http://www.sunsoft.com/`⟩, 1996.

[Hardenburg, 1995] Dr. William Hardenburg. Personal interviews. Conducted by the author, December 1995.

[Law and Kelton, 1991] Averill M. Law and W. David Kelton. *Simulation and Modeling Analysis*. McGraw-Hill, 1991.

[Schruben, 1983] Lee Schruben. Simulation Modeling with Event Graphs. *Communications of the ACM*, 26:957–963, 1983.

[Streetly, 1996] Martin Streetly, editor. *Jane's Radar and Electronic Warfare Systems 1996-1997*. Sentenel House, 8th edition, 1996.

[Swerling, 1976] P. Swerling. Probability of Detection for Fluctuating Targets. In *Detection and Estimation Applications to Radar*. Dowden, Hutchinson & Ross, Inc. Stroudsberg, PA, 1976.

[OASG, 1977] OASG. *Naval Operations Analysis*. Naval Institute Press, 1977. Operations Analysis Study Group of the United States Naval Academy.

[Washburn, 1989] Alan R. Washburn. *Search and Detection*. ORSA Books, care of Ketron, Inc., 2nd edition, 1989.

BIBLIOGRAPHY

Cornell, Gary and Horstman, Cay, S., *Core Java*, SunSoft Press, 1996.

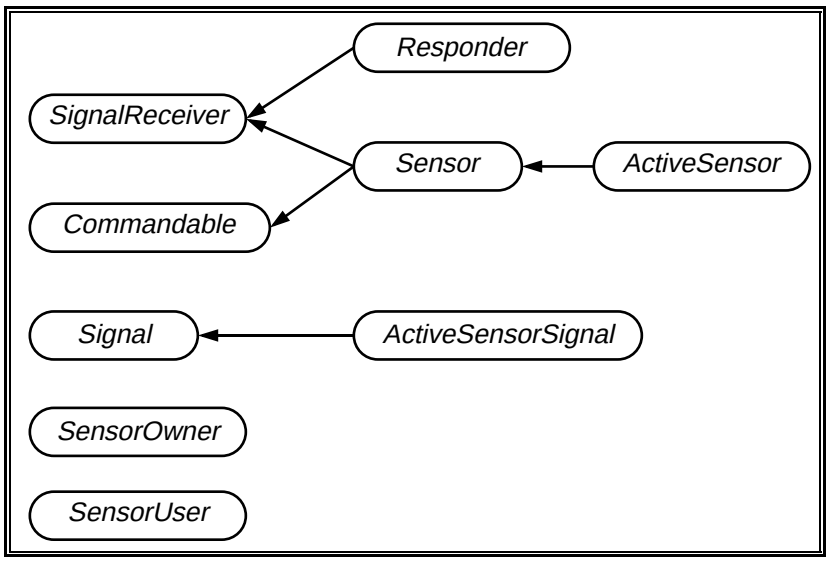
Cox, Brad, J., Planning the Industrial Software Revolution, In IEEE Software Magazine *Software Technologies of the 1990's*, IEEE, 1990.

Flanagan, David, *Java in a Nutshell*, O'Reilly & Assciates, Inc., 1996.

APPENDIX A. ABSTRACT SENSOR COMPONENT LISTINGS

This appendix contains the Java interface files described in Chapter II. These files are part of the Java package `simkit.javasim.ammt` which is available in it's entirety at the URL below.

All source code is available from professor Buss's web pages at `http://dubhe.cc.nps.navy.mil/~ahbuss/`.



```
// FILE: SignalReceiver
package simkit.javasim.ammt;

import scalar.*;
import misc.*;
import simkit.*;
import simkit.javasim.*;
import simkit.awt.*;
import g2d.*;
import jgl.*;

/**
* Base interface for all simulation entities that
* can receive signals of any kind.
* <br>
* @author Kirk A. Stork
* @version 1.0
**/

public interface SignalReceiver {

/**
Receive a signal.
<br>
If the signal is unkown, do nothing or print a warning,
otherwise, handle the signal.
**/

   public void receiveSignal( Signal s );
}
```

```
// FILE:  Responder.java
package simkit.javasim.ammt;

import scalar.*;
import misc.*;
import simkit.*;
import simkit.javasim.*;
import simkit.awt.*;
import g2d.*;
import jgl.*;

/**
* Interface for objects that respond to signal
* reception by instantiating a new signal and
* sending it sowmewhere.
* <br>
* This interface is provided for typing purposes.
* @author Kirk A. Stork
* @version 1.0
**/

public interface Responder
   extends SignalReceiver
{
}
```

```
// FILE Sensor.java
package simkit.javasim.ammt;
import scalar.*;
import misc.*;
import simkit.*;
import simkit.javasim.*;
import simkit.awt.*;
import g2d.*;
import jgl.*;

/**
* Base interface for all kinds of sensors.
* <br>
* The methods defined in this interface handle two
* commands, and a request for the sensor's information.
* Sensor information is information about the contacts
* this sensor has detected, which will be sent to
* the user specified.  The information is not returned
* because the request may require expending simulated
* time.
*
* @author Kirk A. Stork
* @version 1.0
**/

public interface Sensor
   extends    SignalReceiver
{
/**
Process a request for information.

This method is called by sensor users when they
want the current contact information this sensor
posesses.  The request is processed and when this object
is ready, it calls the SensorUser's receiveSensorInfo
method to deliver the information.
**/

   public void sendInfo( SensorUser u);

/**
Turn this sensor on.
<br>
When this method is called, the sensor should be turned on,
```

```
if the sensor is capable of being turned on.
**/

   public void activate();

/**
Turn this sensor off.
<br>
When this method is called, the sensor should be turned off,
if the sensor is capable of being turned off.
**/

   public void deactivate();

   public void receiveCommand( SensorCommand c);
}
```

```
// FILE: ActiveSensor.java

package simkit.javasim.ammt;
import misc.*;
import simkit.*;
import simkit.javasim.*;
import simkit.awt.*;
import g2d.*;
import jgl.*;

/**
* Basse interface for active sensors.
*
* Active sensors are sensors that emit signals to
* do their work.
*
* @author Kirk A. Stork
* @version 1.0
**/
public interface ActiveSensor
   extends Sensor
{
/**
Tell this sensor to generate a signal.

This is included in the interface
for the case when a single signal
is to be generated by a command.
**/

   public void generateSignal();

/**
Change the sensor state to standby.

Standby is a state sometimes available for
sensors that have a long warmup period, or that
are used intermittantly.
**/

   public void standBy();
}
```

```
// FILE Signal.java
package simkit.javasim.ammt;
import jgl.*;

/**
* Base interface for signals.
* <br>
* A signal is a simulation entity that, in the abstract
* sense, carries information from place to place.
* Communication and sensors are the first uses of this
* interface, although others are expected.
* @author Kirk A. Stork
* @version 1.0
**/

public interface Signal
{
/**
Tell this signal to propagate.
<br>
The specific meaning of propagation is highly implementation
specific.  The intention is for the signal to interact
with the referee to discover how to get where its going.
This might involve passage of simulation time.
**/

   public void propagate();

/**
Set a property of the signal.
**/

//   public void putProperty( String property, Object value);

/**
Get a property of the signal, return null silently if
the asked for property does not exist (or print a warning).
**/

//   public Object property(String property);

/**
Return a copy of the properties in this signal.
```

```
This method is used when a responder needs to construct
a signal in response to a signal interaction.
**/

//   public HashMap properties();

/**
Call back for the SignalArrival event.
<br>
@see SignalArrivalEVT
**/

   public Sensor creator();
// public void handleArrivalEVT ( SignalReceiver receiver);
   public void dispose();
}
```

```
// FILE Signal.java
package simkit.javasim.ammt;
import jgl.*;

/**
* Interface for Active Sensor Signals.
* @author Kirk A. Stork
* @version 1.0
**/

public interface ActiveSensorSignal
   extends Signal
{
/**
Return a reference to the creator of this signal.

This method is used when a responder needs to construct
a signal in response to a signal interaction.
**/

//   public SignalSender creator();

/**
Indicate this signal to be the product of a signal/responder
interaction.
<br>
This method should cause the signal to return to the
active sensor that created the signal whos interaction
instantiated this signal.  No other interactions should
occur between this signal and other entities.
*/
   public void returnToSender();
}
```

```
// FILE SensorOwner.java
package simkit.javasim.ammt;
import jgl.*;

/**
* Base interface for entities that can own sensors
* <br>
* A sensor can have only one owner.  Ownership implies
* the ability to send the sensor operational commands.
*
* @author Kirk A. Stork
* @version 1.0
**/

public interface SensorOwner
{
/**
Accept sensor status information.
<br>
This is called by sensor objects when they want
to tell their owner that sensor status has changed.
**/

   public void receiveSensorStatus( SensorStatus s);
}
```

```
// FILE SensorUser.java
package simkit.javasim.ammt;
import jgl.*;


/**
* Base interface for entities that can use sensor
* information.
* <br>
* @author Kirk A. Stork
* @version 1.0
**/

public interface SensorUser
{
/**
Accept sensor information.
<br>
This is called by sensor objects when they want
to deliver contact information to this user.
**/

   public void receiveSensorInfo( SensorInfo info );
}
```

APPENDIX B. COOKIE CUTTER SENSOR DEMONSTRATION LISTINGS

This appendix contains the Java classes described in Chapter III for the Cookie Cutter Sensor barrier search model.

All source code is available from professor Buss's web pages at `http://dubhe.cc.nps.navy.mil/~ahbuss/`.

```
// FILE: CCActiveSensor.java
import scalar.*;
import misc.*;
import simkit.*;
import simkit.javasim.*;
import simkit.awt.*;
import g2d.*;
import simkit.javasim.ammt.*;
import jgl.*;
import java.io.*;


public class CCActiveSensor
   extends       SimEntityImpl
   implements    ActiveSensor
{
   Length           maxRange_;
   Platform         owner_;
   boolean          wait_for_data_requests_;
   String           status_;
   PrintStream      out;
   DataAccumulator  myStats;
   boolean          detected;
   double glimpse;
   CCBarrier  theModel_;

   public CCActiveSensor(  String         name,
                           Platform       owner,
                           boolean        wait_for_data_requests,
                           Length         maxRange,
                           CCBarrier      theSim )
  {
      super(name);
      theModel_ = theSim;
      owner_ = owner;
      wait_for_data_requests_ = wait_for_data_requests;
      maxRange_ = maxRange;
      myStats = null;
      try{
         out = new PrintStream(new FileOutputStream("CCActiveSensor.log"));
      } catch ( Exception e) {
         System.err.println(e);
      }
      detected = false;
```

```
}

public void doSensorCommand(SensorCommand c) {
   if ( tracing ) Trace.msg( Trace.MSG,
                             "command() in: " +
                             this.toString() +
                             " argument: " +
                             c.toString()
                             );

   String status = (String)(c.get("Change"));
   if ( status != null ) {
      status_ = status;
      changeStatus();
   } else {
      System.err.println("WARNING: Sensor command not understood");
   }
}

private void changeStatus() {
   if (status_.equals("Active")) {
      generateSignal();
      CCGenerateSignalEVT  e = new CCGenerateSignalEVT(this);
      e.waitDelay(new Time(0.0));
      detected = false;
   }

   if (status_.equals("Passive")) {
      this.interrupt("CCGenerateSignalEVT");
   }
}

public void setGlimpse(double interval) {
   glimpse = interval;
}

public void generateSignal() {
   if ( detected ) { return; }
      CCActiveSensorSignal s =
      new CCActiveSensorSignal( this, owner_.position(), maxRange_);
      s.propagate();
      CCGenerateSignalEVT  e = new CCGenerateSignalEVT(this);
      e.waitDelay(new Time(glimpse));
}
```

```
   public void receiveSignal( Signal s ){
      if ( s instanceof CCActiveSensorSignal ) {
         Object result = ((CCActiveSensorSignal)s).interactionLocation();
         if ( result != null &&  s.creator() == this ) {
            detected = true;
            theModel_.doEndSimEVT(this);
         }
      }
   }

   public void standBy() {
   }

   public void activate() {
   }

   public void deactivate() {
   }

   public void sendInfo(SensorUser u) {
   }

   public void receiveCommand( SensorCommand c) {
      doSensorCommand(c);
   }


   public void reportTo( DataAccumulator stats ) {
      myStats = stats;
   }

   public void reset() {
      if ( detected ) {
         myStats.getSample(1.0);
      } else {
         myStats.getSample(0.0);
      }
      detected = false;
   }

}  // class SimpleRadar
```

```
class CCGenerateSignalEVT extends SimEvent
{
   private boolean localdisposed;

   public CCGenerateSignalEVT(CCActiveSensor o) {
      super(o);
      localdisposed = false;
   }

   public void onRun() {
      ((CCActiveSensor)myOwner).generateSignal();
   }

   public void onInterrupt(){}

   public void dispose() {
      super.dispose();
      if (localdisposed) return;
      localdisposed=true;
   }

   public void finalize() {
      super.finalize();
      if ( localdisposed) return;
      this.dispose();
   }

}
```

```
// FILE: CCActiveSensorSignal.java
import scalar.*;
import misc.*;
import simkit.*;
import simkit.javasim.*;
import simkit.awt.*;
import g2d.*;
import simkit.javasim.ammt.*;
import jgl.*;
import java.util.*;


public class CCActiveSensorSignal
   implements ActiveSensorSignal
{
   private CCActiveSensor creator_;
   private Time creationTime_, interactionTime_;
   private Position creationLocation_, interactionLocation_;
   private Length maxRange_;
   private   boolean CCActiveSensorSignalDisposed;

   public CCActiveSensorSignal( CCActiveSensor  owner,
                                Position        startingPoint,
                                Length          maxR ) {
      creator_             = owner;
      creationTime_        = TimeMaster.SimTime();
      creationLocation_    = startingPoint;
      maxRange_            = maxR;
      interactionLocation_ = null;
      interactionTime_     = null;
      CCActiveSensorSignalDisposed = false;
   }

   public CCActiveSensorSignal( CCActiveSensorSignal original_signal,
                                Position loc )
   {
      creator_     = original_signal.creator_;
      creationTime_ = original_signal.creationTime_;
      creationLocation_ = original_signal.creationLocation_;
      maxRange_ = original_signal.maxRange_;
      interactionTime_ = TimeMaster.SimTime();
      interactionLocation_ = loc;
      CCActiveSensorSignalDisposed = false;
   }
```

```
public Sensor creator() {
   return creator_;
}

public Position creationLocation() {
   return creationLocation_;
}

public void propagate() {
   Responder       cst;
   Position  cst_offset, cst_pos;

   for( Enumeration e = Referee.responders();
        e.hasMoreElements();) {

      cst         = (Responder)(e.nextElement());
      cst_pos     = ((Positionable)cst).position();
      cst_offset  = (Position)(cst_pos.subtract
                           ( creationLocation_ ) );
      double dist = cst_offset.length();

      if ( dist <= maxRange_.value() ) {
         cst.receiveSignal( this );
      }
   }
   this.finalize();
}

public String toString() {
   return super.toString();
}

public Position interactionLocation() {
   return interactionLocation_;
}

public void returnToSender() {
   ((SignalReceiver)creator_)
      .receiveSignal(this);
   this.finalize();
}

public void dispose() {
```

```
      if (CCActiveSensorSignalDisposed) return;
      creator_             = null;
      creationTime_        = null;
      creationLocation_    = null;
      maxRange_            = null;
      interactionLocation_ = null;
      interactionTime_     = null;
      CCActiveSensorSignalDisposed = true;
   }

   public void finalize() {
      dispose();
   }
} // class CookieCutterRadarSignal
```

```
// FILE: CCActiveSignalResponder.java

import scalar.*;
import misc.*;
import simkit.*;
import simkit.javasim.*;
import simkit.awt.*;
import g2d.*;
import simkit.javasim.ammt.*;
import jgl.*;
import java.util.*;


public class CCActiveSignalResponder
   implements Responder
{

   private Positionable owner_;

   public CCActiveSignalResponder(SignalReceiver owner)
      throws MoveNotSupportedException
   {
      if ( owner instanceof Positionable ) {
         owner_ = (Positionable)owner;
      }
      else throw new MoveNotSupportedException();
   }

   public void receiveSignal(Signal s) {
      if ( s instanceof CCActiveSensorSignal ) {

         CCActiveSensorSignal response =
            new CCActiveSensorSignal(
                  (CCActiveSensorSignal)s,
                  owner_.position());
         response.returnToSender();
      }

   }
}
```

```
// FILE: SimpleShip.java
import scalar.*;
import misc.*;
import simkit.*;
import simkit.javasim.*;
import simkit.awt.*;
import g2d.*;
import simkit.javasim.ammt.*;
import jgl.*;
import java.util.*;
/**
Target ship player for Model1.
*/

public class SimpleShip
   extends Platform
   implements SensorOwner, SensorUser, Commandable, Responder
{
   private Time nextManeuverTime, legtime;
   private Bearing course1, course2;
   private int curCourseNo;
   private Length leg;
   private Array waypoints;
   private CCBarrier theModel;

   public SimpleShip( String name, CCBarrier theSim ) {
      super(name);
      nextManeuverTime = null;
      responders_ = new HashSet();
      sensors_    = new HashSet();
      theModel = theSim;
      Referee.registerPlayer(this);
   }


   private void doReportWhen( ManeuverCommand c) {
      Length l = (Length)(c.get("Distance Travelled"));
      Time delay = l.divide(speed());

      WaypointArrivalEVT e = new WaypointArrivalEVT(this);
      e.waitDelay(delay);
   }

   public void doWaypointArrivalEVT() {
```

```
   theModel.doEndSimEVT(this);
}

private void doStandardPattern(ManeuverCommand c) {
   if ( ((String)(c.get("Pattern"))).equals("Barrier")) {
      doBarrierPattern(c);
   }
   else {
      System.err.println("Unknown Maneuver command received by " +
                          this);
   }
}


private void doBarrierPattern( ManeuverCommand c) {
   waypoints = new Array();
   waypoints.add(new Position(position()));
   waypoints.add(((Position)(c.get("End"))));
   setSpeed((Velocity)(c.get("Speed")));
   course2 = ((Position)(waypoints.at(1)))
                .bearingFrom((Position)(waypoints.at(0)));
   course1 = ((Position)(waypoints.at(0)))
                .bearingFrom((Position)(waypoints.at(1)));
   leg = new Length(((Position)(waypoints.at(0))).
              subtract((Position)(waypoints.at(1))).
                length());

   legtime = leg.divide(speed());
   setCourse(new Bearing(course2));
   curCourseNo = 2;

   SimpleShipManeverEVT e = new SimpleShipManeverEVT(this);
   e.waitDelay(legtime);
}

public void doManeuverEVT() {
   updatePosition();

   switch(curCourseNo) {
      case 1:
         // this means the next course should be to point 1
         this.setCourse(new Bearing(course2));
         curCourseNo = 2;
         break;
```

```
         case 2:
            // this means the next course should be to point 2
            this.setCourse(new Bearing(course1));
            curCourseNo = 1;
            break;
         default:
           System.err.println("Oh-Oh, something is wrong");

      }
      SimpleShipManeverEVT e = new SimpleShipManeverEVT(this);
      e.waitDelay(legtime);
   }

   public void receiveSensorStatus(SensorStatus s){}
   public void receiveSensorInfo(SensorInfo i){}

//==========================================================================
// COMMAND HANDLING
//==========================================================================
   public void receiveCommand( Command c ) {
      if ( c instanceof ManeuverCommand ) {
         doManeuverCommand( (ManeuverCommand)c );
      } else
      if ( c instanceof SensorCommand ) {
         doSensorCommand( (SensorCommand)c );
      }

   }
//==========================================================================
// SENSORS
//==========================================================================
   protected HashSet sensors_;

   public void addSensor( Sensor s) {
      sensors_.put(s);
   }

   public void removeSensor( Sensor s) {
      sensors_.remove(s);
   }


//==========================================================================
```

```
// RESPONDERS
//==========================================================================
   protected HashSet responders_;

   public void addResponder( Responder r) {
      responders_.put(r);
   }

   public void removeResponder( Responder r) {
      responders_.remove(r);
   }


//==========================================================================
// SIGNAL RECEPTION
//==========================================================================
   public void receiveSignal( Signal s ) {
      for ( Enumeration e = responders_.elements();
                        e.hasMoreElements();) {
         ((Responder)(e.nextElement())).receiveSignal(s);
      }

      for ( Enumeration e = sensors_.elements();
                        e.hasMoreElements();) {
         ((Sensor)(e.nextElement())).receiveSignal(s);
      }

   }
   public boolean knowsCommand( Command c ) {
   // this has to get more useful, but for now its ok
      if ( ( c instanceof SensorCommand) ||
           ( c instanceof ManeuverCommand ) ) {
         return true;
      }
      return false;
   }

   public void doManeuverCommand( ManeuverCommand c ) {

      String maneuverKind = (String)(c.get("Kind"));

      if (maneuverKind.equals("Course/Speed Change")) {
         this.setCourse(((Bearing)(c.get("Course"))));
         this.setSpeed(((Velocity)(c.get("Speed"))));
```

```
      }
      if (maneuverKind.equals("Standard Pattern")) {
          doStandardPattern(c);
      }
      if (maneuverKind.equals("Report When")) {
          doReportWhen(c);
      }
   }

   protected void doSensorCommand( SensorCommand c ) {

      String sensorType = (String)(c.get("SensorType"));

      if (sensorType.equals("All")) {
         for ( Enumeration e = sensors_.elements();
               e.hasMoreElements(); ) {

            ((Sensor)(e.nextElement())).receiveCommand(c);
         }
      } else {
         System.err.println(
             "Don't know how to handle sensor commands for " +
             "sensors of kind " + sensorType);
      }
   }

}  // class SimpleShip




class SimpleShipManeverEVT extends SimEvent
{
   public SimpleShipManeverEVT(SimEntityImpl owner) {
      super(owner);
   }

   public void onRun() {
      ((SimpleShip)(myOwner)).doManeuverEVT();
   }

   public void onInterrupt() {}
}
```

```
class WaypointArrivalEVT extends SimEvent
{
   public WaypointArrivalEVT(SimEntityImpl owner) {
      super(owner);
   }

   public void onRun() {
      ((SimpleShip)(myOwner)).doWaypointArrivalEVT();
   }

   public void onInterrupt() {}
}
```

```
// FILE CCBarrier.java
import scalar.*;
import misc.*;
import simkit.*;
import simkit.javasim.*;
import simkit.awt.*;
import g2d.*;
import simkit.javasim.ammt.*;
import java.io.*;
import java.util.*;

/**
Simple Barrier Search simulation
*/

public class CCBarrier extends SimEntityImpl
{
   static int nTrials, trialNo, i;
   static Date startTime, endTime;
   static Histogram batchStats =
          new Histogram("Batch Means", true);
   static  PrintStream       out;
   static Format mf = new Format("%12.6f");


   public static void main( String args[])
      throws Exception
   {
      if ( args.length   != 6 ) {
         System.out.println("Wrong number of arguments");
         System.out.println(
            "  usage: java CCBarrier <Number of Batches> "+
            " <Number of Trials per batch>" +
            " <Searcher Speed> <Target Speed>" +
            " <Glimpse Interval> <seed>");
         System.exit(1);
      }
      TimeMaster.reset();
      out = new PrintStream(new FileOutputStream("CCBarrier_Results.dat"));

      int batches = Integer.parseInt(args[0]);
      nTrials = Integer.parseInt( args[1] );
      double ss = (Double.valueOf(args[2])).doubleValue();
      double ts = (Double.valueOf(args[3])).doubleValue();
```

```
    double glimpse = (Double.valueOf(args[4])).doubleValue();
    long seed = Long.parseLong(args[5]);
    startTime = new Date();
    trialNo = 0;
    CCBarrier theModel = new CCBarrier( ts, ss);

    theModel.targetPositRand.SetSeed(seed);
    theModel.theSensor.setGlimpse(glimpse);
    for ( i = 0; i < batches; i++){
    trialNo = 0;
    while( trialNo < nTrials ) {
       theModel.runSim();
       theModel.theSensor.reset();
       trialNo++;
    }
    endTime = new Date();

    System.out.println("Number of Trials:           " +
                        theModel.searchSuccessStats.count());
    System.out.println("Avg Detection Rate:         " +
                        theModel.searchSuccessStats.mean());
    out.println(i + "\t" + mf.form(theModel.searchSuccessStats.mean()) +
                "\t" + mf.form(theModel.searchSuccessStats.variance()));


    System.out.println("Cumulative Run Time = " +
                      ((endTime.getTime() - startTime.getTime())/1000.0) +
                      " sec");
    System.out.println("Next Seed = " +theModel.targetPositRand.seed());

    batchStats.getSample(theModel.searchSuccessStats.mean());
    theModel.searchSuccessStats.reset();
    }

    batchStats.makeHistogramPlotDisplay(/*bin width*/0.1);
    batchStats.makeMovingAvePlotDisplay();
    batchStats.makeReportDisplay();
}




 SimpleShip        searcher, target;
 SimpleEnvironment theEnvironment;
```

```
Command           c;
double            d, dd,searchSpeed_,targetSpeed_;
RandomStream      targetPositRand;
DataAccumulator   searchSuccessStats;
CCActiveSensor    theSensor;
ClockFrame        clock;
Position startSearchAt;


public CCBarrier(double targetSpeed, double searchSpeed) {
   searcher             = new SimpleShip("USS Searcher", this);
   target               = new SimpleShip("USS Target", this);
   theEnvironment       = new SimpleEnvironment(1.0);
   searchSuccessStats   = new DataAccumulator(
                              "Search Success", false);
   targetPositRand      = new RandomStream();
   targetSpeed_=targetSpeed;
   searchSpeed_=searchSpeed;
   d = 0;
   double dd;

   theSensor = new CCActiveSensor( "Cookie Cutter MK I",
                                    searcher,
                                    false,
                                    Length.fromNMValue(10.0),
                                    this );

   theSensor.reportTo( searchSuccessStats);

   searcher.addSensor(theSensor);

   target.addResponder( new CCActiveSignalResponder(
                                 target));
   try{
      out = new PrintStream(
                new FileOutputStream("CCBarrier.log"));
   } catch ( Exception e) {
      System.err.println(e);
   }
   searcher.setPosition( new Position(
                                 Length.fromNMValue(10.0),
                                 new Length(0.0),
                                 new Length(0.0) ) );
   c = new ManeuverCommand();
```

```
      c.put("Kind", "Standard Pattern");
      c.put("Pattern", "Barrier");
      c.put("End", new Position(   Length.fromNMValue(70.0),
                                        new Length(0.0),
                                        new Length(0.0)));
      c.put("Speed", Velocity.fromKnotValue( searchSpeed_));
      searcher.receiveCommand( c );

      c = new SensorCommand();
      c.put("Change", "Active");
      c.put("SensorType", "All");
      searcher.receiveCommand( c );

   }


   public void runSim() {


      dd = targetPositRand.Uniform(0,80);

      target.setPosition( new Position( Length.fromNMValue(dd),
                                        Length.fromNMValue(10.0),
                                        new Length(0.0) ));


      c = new ManeuverCommand();
      c.put("Kind", "Course/Speed Change");
      c.put("Course", new Bearing( 180.0, 0.0, 0.0));
      c.put("Speed", Velocity.fromKnotValue( targetSpeed_));
      target.receiveCommand( c );

      c = new ManeuverCommand();
      c.put("Kind", "Report When");
      c.put("Distance Travelled", Length.fromNMValue(20.0));
      target.receiveCommand(c);

//      Time delay = (Length.fromNMValue(20.0)).
//                   divide(Velocity.
//                   fromKnotValue( targetSpeed_));
//      System.out.println("Barrier penetration will take " + delay);
//      SimEvent e = new EndSimEVT(this);
//      e.waitDelay(delay);
```

```
      TimeMaster.startSimulation();
   }


   public void doEndSimEVT(Object caller) {
//      searcher.interruptAll();
      target.interruptAll();
//      theSensor.interruptAll();
//      this.interruptAll();
      TimeMaster.stop();
   }
}

//=====================================================
class EndSimEVT extends SimEvent
{
   public EndSimEVT(SimEntityImpl owner) {
      super(owner);
   }

   public void onRun() {
      ((CCBarrier)(myOwner)).doEndSimEVT(this);
   }

   public void onInterrupt() {}
}
//=====================================================
```

APPENDIX C. MONOPULSE RADAR DEMONSTRATION LISTINGS

This appendix contains the Java classes described in Chapter III for the Monopulse Radar model. Please note that the SimpleShip class from the Cookie Cutter model was reused for this model without changes.

All source code is available from professor Buss's web pages at `http://dubhe.cc.nps.navy.mil/~ahbuss/`.

```
// File MonopulseRadar.java
import scalar.*;
import misc.*;
import simkit.*;
import simkit.javasim.*;
import simkit.awt.*;
import g2d.*;
import simkit.javasim.ammt.*;
import java.io.*;
import java.util.*;

public class MonopulseRadar
   extends SimEntityImpl
   implements ActiveSensor
{
   Platform         owner_;
   boolean          wait_for_data_requests_;
   String           status_;
   PrintStream      out;
   DataAccumulator  myStats;
   boolean          detected;
   Time glimpseInterval_;
   Length rangeScale_;
   PulseTrainSim theModel_;
   Power   transmitPower_;
   int  nPulses_, nReturns;
   double antennaGain_, antennaAperature_,
          integrationEfficiency_, threashold_;
   PulseTrain s_;
   RandomStream noiseStream;
   double noisedB_;

   public MonopulseRadar(  String         name,
                           Platform       owner,
                           boolean        wait_for_data_requests,
                           Length         maxRange,
                           Power          transmitPower,
                           double gain,
                           double efficiency,
                           double aperature,
                           double threashold,
                           int            nPulses,
                           double meanNoiseDB,
                           PulseTrainSim  theSim )
```

```
{
    super(name);
    noisedB_ =meanNoiseDB;
    noiseStream = new RandomStream((int)7);
    owner_ = owner;
    wait_for_data_requests_ = wait_for_data_requests;
    rangeScale_ = maxRange;
    theModel_ = theSim;
    myStats = null;
    detected = false;
    transmitPower_ =transmitPower;
    nPulses_=nPulses;
    antennaGain_ = gain;
    antennaAperature_ = aperature;
    integrationEfficiency_ = efficiency;
    threashold_=threashold;
    glimpseInterval_ = new Time(0.0);
    try{
       out = new PrintStream(new FileOutputStream("Radar.log"));
    } catch ( Exception e) {
       System.err.println(e);
    }
 }



 public void doSensorCommand(SensorCommand c) {
    if ( tracing ) Trace.msg( Trace.MSG,
                              "command() in: " +
                              this.toString() +
                              " argument: " +
                              c.toString()
                              );

    String status = (String)(c.get("Change"));
    if ( status != null ) {
       status_ = status;
       changeStatus();
    } else {
       System.err.println("WARNING: Sensor command not understood");
    }
 }

 private void changeStatus() {
```

```
   if (status_.equals("Active")) {
      GeneratePulseTrainEVT
         e = new GeneratePulseTrainEVT(this);
      e.waitDelay(new Time(0.0));
   }

   if (status_.equals("Passive")) {
      this.interrupt("GeneratePulseTrainEVT");
   }
}

public void setGlimpse(Time interval) {
   glimpseInterval_ = interval;
}

public void setRangeScale(Length range) {
   rangeScale_ = range;
}

public void generateSignal() {
   if ( s_ != null) s_.finalize();
   if ( detected ) { return; }
      s_ =
      new PulseTrain( this,
                      owner_.position(),
                      transmitPower_,
                      rangeScale_
                      );
      s_.propagate();

      GeneratePulseTrainEVT
         e = new GeneratePulseTrainEVT(this);
      e.waitDelay(glimpseInterval_);
}


public void receiveSignal(Signal s)
{
   if ( (s instanceof PulseTrain)  &&
        ((PulseTrain)s).creator() == this ) {
      nReturns++;
      Power signalStrength = (((PulseTrain)s).returnedPower());
      double snr = signalStrength.value() *
                   antennaGain_ *
```

```
                    antennaAperature_ *
                    integrationEfficiency_ *
                    nPulses_;
      snr -= noiseStream.BoxMuller(noisedB_, noisedB_/2.0);
      if ( snr > threashold_ ) {
         detected = true;
         out.println(TimeMaster.SimTime() +
                     " detect " + theModel_.i + "\t" +
                     theModel_.trialNo + "\t" +
                     snr + "\t" +
                     (((PulseTrain)s).interactionLocation_.subtract(
                     owner_.position())).length());
         theModel_.doEndSimEVT(this);
      } else {
         out.println(TimeMaster.SimTime() +
                     " nodetect " + theModel_.i + "\t" +
                     theModel_.trialNo + "\t" +
                     snr + "\t" +
                     (((PulseTrain)s).interactionLocation_.subtract(
                     owner_.position())).length());
      }
   }
   s.dispose();
}

public void standBy() {
}

public void activate() {
}

public void deactivate() {
}

public void sendInfo(SensorUser u) {
}

public void receiveCommand( SensorCommand c) {
   doSensorCommand(c);
}


public void reportTo( DataAccumulator stats ) {
   myStats = stats;
```

```
   }

   public void reset() {
      if ( detected ) {
         myStats.getSample(1.0);
      } else {
         myStats.getSample(0.0);
      }
      nReturns = 0;
      detected = false;
   }

}  // class MonopulseRadar


class GeneratePulseTrainEVT extends SimEvent
{
   private boolean localdisposed;

   public GeneratePulseTrainEVT(MonopulseRadar o) {
      super(o);
   }

   public void onRun() {
      ((MonopulseRadar)myOwner).generateSignal();
   }

   public void onInterrupt(){}

}
```

```
//FILE: PulseTrain.java
import scalar.*;
import misc.*;
import simkit.*;
import simkit.javasim.*;
import simkit.awt.*;
import g2d.*;
import simkit.javasim.ammt.*;
import java.io.*;
import java.util.*;


public class PulseTrain
   extends SimEntityImpl
   implements ActiveSensorSignal
{
      private static final double C = 299792458.0;

      private MonopulseRadar creator_;
      private Time           creationTime_, interactionTime_;
       Position       creationLocation_, interactionLocation_;
      private Power          transmittedPower_,
                             reflectedPower_,
                             receivedPower_,
                             returnedPower_;
      private boolean        PT_Disposed;
      private Length         rangeScale_;
      private double         partialSNR;
      private static    PrintStream      out;
      static boolean inited=false;

// Constructor for outgoing signals

   public PulseTrain( MonopulseRadar  owner,
                      Position        startingPoint,
                      Power           power,
                      Length          rangeScale )
   {
      creator_             = owner;
      creationTime_        = TimeMaster.SimTime();
      creationLocation_    = startingPoint;
      transmittedPower_    = power;
      rangeScale_          = rangeScale;
```

```
      interactionTime_     = null;
      reflectedPower_      = null;
      receivedPower_       = null;
      interactionLocation_ = null;
      PT_Disposed          = false;

      if (!inited ) {
      try{
         out =new PrintStream(new FileOutputStream("train.log"));
      } catch ( Exception e) {
         System.err.println(e);
      }
         inited = true;
      }
   }

// constructor for reflected signals

   public PulseTrain( PulseTrain original_signal,
                      double     reflectionRatio,
                      Position   interactionLocation )
   {
      try {
         creator_             = original_signal.creator_;
         creationTime_        = original_signal.creationTime_;
         creationLocation_    = original_signal.creationLocation_;
         transmittedPower_    = original_signal.transmittedPower_;
         rangeScale_          = original_signal.rangeScale_;

         interactionTime_     = TimeMaster.SimTime();
         interactionLocation_ = interactionLocation;

         PT_Disposed          = false;

      // calculate received power at target
         Length dist = new Length((creationLocation_.
                          subtract(interactionLocation_))
                           .length());
         double proploss = dist.value() * dist.value();
         proploss        = proploss* 4.0 * Math.PI;
         receivedPower_ = (Power)(transmittedPower_.divide(proploss));
          // really w/m^2
      // apply the ratio from the interaction
```

```
      out.println(TimeMaster.SimTime() + "\t" +
                  receivedPower_ + " Delivered at range " +
                  dist);

      reflectedPower_ = receivedPower_.multiply(reflectionRatio);
       // w again

      out.println("        " + "\t" +
                  reflectedPower_ + " reflected");

   } catch (Exception e) {
      System.err.println(e);
      e.printStackTrace();
   }
}

public void propagate() {
   Responder       cst;
   Position        cst_offset, cst_pos;
   Time            delay;

   for( Enumeration e = Referee.responders();
        e.hasMoreElements();) {


      cst         = (Responder)(e.nextElement());
      cst_pos     = ((Positionable)cst).position();
      cst_offset  = (Position)(cst_pos.subtract
                          ( creationLocation_ ) );
      double dist = cst_offset.length();


      if ( dist <= rangeScale_.value() ) {
         // calculate arrival delay for this potential contact

         if ( dist > 0) {
            delay = new Time(dist/C);
            PulseArrivalEVT evt = new PulseArrivalEVT(this, cst);
            evt.waitDelay(delay);
         }
      }
   }
   Time dieAt = new Time(rangeScale_.value() / C);
   PulseDeathEVT evt = new PulseDeathEVT(this);
```

```
   evt.waitDelay(dieAt);
}

public void doPulseDeathEVT() {
   this.dispose();
}

public void doPulseArrivalEVT( SignalReceiver destination) {
   destination.receiveSignal(this);
}

public void doPulseReturnEVT( SignalReceiver destination) {
   destination.receiveSignal(this);
   this.dispose();
}

public Sensor creator() {
   return creator_;
}

public Position creationLocation() {
   return creationLocation_;
}

// method for sensor to get the data out of
// the signal

public Power returnedPower() {
   return returnedPower_;
}

public String toString() {
   return super.toString();
}

public void returnToSender() {
   double dist = ((creator_.owner_.position())
                  .subtract(interactionLocation_)).length();
   double proploss = dist * dist;
   proploss        = proploss* 4.0 * Math.PI;
   returnedPower_  = reflectedPower_.divide(proploss);
   double delay = dist/C;
   PulseReturnEVT e = new PulseReturnEVT(this, creator_);
   e.waitDelay(new Time(delay));
```

```
   }

   public void dispose() {
      if (PT_Disposed) return;
      creator_             = null;
      creationTime_        = null;
      creationLocation_    = null;
      transmittedPower_    = null;
      interactionTime_     = null;
      reflectedPower_      = null;
      rangeScale_          = null;
      interactionLocation_ = null;
      PT_Disposed          = true;
   }

   public void finalize() {
      super.finalize();  // let the SimEntityImpl do its cleanup
      dispose();
   }


} // class PulseTrain


class PulseArrivalEVT extends SimEvent
{
   private SignalReceiver destination_;

   public PulseArrivalEVT( PulseTrain     owner,
                           SignalReceiver destination) {
      super(owner);
      destination_ = destination;
   }

   public void onRun() {
      ((PulseTrain)myOwner).doPulseArrivalEVT(destination_);
   }

} // class PulseArrivalEVT

class PulseReturnEVT extends SimEvent
{
   private SignalReceiver destination_;
```

```
   public PulseReturnEVT( PulseTrain     owner,
                          SignalReceiver destination) {
      super(owner);
      destination_ = destination;
   }

   public void onRun() {
      ((PulseTrain)myOwner).doPulseReturnEVT(destination_);
   }

} // class PulseArrivalEVT

class PulseDeathEVT extends SimEvent
{
   public PulseDeathEVT( PulseTrain     owner) {
      super(owner);
   }

   public void onRun() {
      ((PulseTrain)myOwner).doPulseDeathEVT();
   }

} // class PulseArrivalEVT
```

```
// FILE: PulseTrainResponder.java

import scalar.*;
import misc.*;
import simkit.*;
import simkit.javasim.*;
import simkit.awt.*;
import g2d.*;
import simkit.javasim.ammt.*;
import jgl.*;
import java.util.*;



public class PulseTrainResponder
   implements Responder
{
   private static RandomStream rcs_rand =
                  new RandomStream( (int)9);

   private Positionable owner_;
   private double       sigma_rcs_;
   private double alpha, lambda;

   public PulseTrainResponder(SignalReceiver owner)
      throws MoveNotSupportedException
   {
      if ( owner instanceof Positionable ) {
         owner_ = (Positionable)owner;
      }
      else throw new MoveNotSupportedException();
   }

   public void setRCS(double sigma_rcs) {
      sigma_rcs_ = sigma_rcs;
      alpha = 2.0;
      lambda = sigma_rcs/2.0;
   }

   public void receiveSignal(Signal s) {
      if ( s instanceof PulseTrain ) {

         // get a random value for my rcs
         PulseTrain response =
```

```
            new PulseTrain( (PulseTrain)s,
                            rcs_rand.Gamma(alpha, lambda),
                            owner_.position());
         response.returnToSender();
      }

   }
}
```

```
// FILE: SimpleShip.java
import scalar.*;
import misc.*;
import simkit.*;
import simkit.javasim.*;
import simkit.awt.*;
import g2d.*;
import simkit.javasim.ammt.*;
import jgl.*;
import java.util.*;
/**
Target ship player for Model1.
*/

public class SimpleShip
   extends Platform
   implements SensorOwner, SensorUser, Commandable, Responder
{
   private Time nextManeuverTime, legtime;
   private Bearing course1, course2;
   private int curCourseNo;
   private Length leg;
   private Array waypoints;
   private PulseTrainSim theModel;

   public SimpleShip( String name, PulseTrainSim theSim ) {
      super(name);
      nextManeuverTime = null;
      responders_ = new HashSet();
      sensors_    = new HashSet();
      theModel = theSim;
      Referee.registerPlayer(this);
   }


   private void doReportWhen( ManeuverCommand c) {
      Length l = (Length)(c.get("Distance Travelled"));
      Time delay = l.divide(speed());

      WaypointArrivalEVT e = new WaypointArrivalEVT(this);
      e.waitDelay(delay);
   }

   public void doWaypointArrivalEVT() {
```

```
   theModel.doEndSimEVT(this);
}

private void doStandardPattern(ManeuverCommand c) {
   if ( ((String)(c.get("Pattern"))).equals("Barrier")) {
      doBarrierPattern(c);
   }
   else {
      System.err.println("Unknown Maneuver command received by " +
                          this);
   }
}


private void doBarrierPattern( ManeuverCommand c) {
   waypoints = new Array();
   waypoints.add(new Position(position()));
   waypoints.add(((Position)(c.get("End"))));
   setSpeed((Velocity)(c.get("Speed")));
   course2 = ((Position)(waypoints.at(1)))
                .bearingFrom((Position)(waypoints.at(0)));
   course1 = ((Position)(waypoints.at(0)))
                .bearingFrom((Position)(waypoints.at(1)));
   leg = new Length(((Position)(waypoints.at(0))).
              subtract((Position)(waypoints.at(1))).
                length());

   legtime = leg.divide(speed());
   setCourse(new Bearing(course2));
   curCourseNo = 2;

   SimpleShipManeverEVT e = new SimpleShipManeverEVT(this);
   e.waitDelay(legtime);
}

public void doManeuverEVT() {
   updatePosition();

   switch(curCourseNo) {
      case 1:
         // this means the next course should be to point 1
         this.setCourse(new Bearing(course2));
         curCourseNo = 2;
         break;
```

```
         case 2:
            // this means the next course should be to point 2
            this.setCourse(new Bearing(course1));
            curCourseNo = 1;
            break;
         default:
           System.err.println("Oh-Oh, something is wrong");

      }
      SimpleShipManeverEVT e = new SimpleShipManeverEVT(this);
      e.waitDelay(legtime);
   }

   public void receiveSensorStatus(SensorStatus s){}
   public void receiveSensorInfo(SensorInfo i){}

//==========================================================================
// COMMAND HANDLING
//==========================================================================
   public void receiveCommand( Command c ) {
      if ( c instanceof ManeuverCommand ) {
         doManeuverCommand( (ManeuverCommand)c );
      } else
      if ( c instanceof SensorCommand ) {
         doSensorCommand( (SensorCommand)c );
      }

   }
//==========================================================================
// SENSORS
//==========================================================================
   protected HashSet sensors_;

   public void addSensor( Sensor s) {
      sensors_.put(s);
   }

   public void removeSensor( Sensor s) {
      sensors_.remove(s);
   }


//==========================================================================
```

```
// RESPONDERS
//==========================================================================
   protected HashSet responders_;

   public void addResponder( Responder r) {
      responders_.put(r);
   }

   public void removeResponder( Responder r) {
      responders_.remove(r);
   }


//==========================================================================
// SIGNAL RECEPTION
//==========================================================================
   public void receiveSignal( Signal s ) {
      for ( Enumeration e = responders_.elements();
                        e.hasMoreElements();) {
         ((Responder)(e.nextElement())).receiveSignal(s);
      }

      for ( Enumeration e = sensors_.elements();
                        e.hasMoreElements();) {
         ((Sensor)(e.nextElement())).receiveSignal(s);
      }

   }
   public boolean knowsCommand( Command c ) {
   // this has to get more useful, but for now its ok
      if ( ( c instanceof SensorCommand) ||
           ( c instanceof ManeuverCommand ) ) {
         return true;
      }
      return false;
   }

   public void doManeuverCommand( ManeuverCommand c ) {

      String maneuverKind = (String)(c.get("Kind"));

      if (maneuverKind.equals("Course/Speed Change")) {

         this.setCourse(((Bearing)(c.get("Course"))));
```

```
         this.setSpeed(((Velocity)(c.get("Speed"))));
      }
      if (maneuverKind.equals("Standard Pattern")) {
          doStandardPattern(c);
      }
      if (maneuverKind.equals("Report When")) {
          doReportWhen(c);
      }
   }

   protected void doSensorCommand( SensorCommand c ) {

      String sensorType = (String)(c.get("SensorType"));

      if (sensorType.equals("All")) {
         for ( Enumeration e = sensors_.elements();
               e.hasMoreElements(); ) {
            ((Sensor)(e.nextElement())).receiveCommand(c);
         }
      } else {
         System.err.println(
               "Don't know how to handle sensor commands for " +
               "sensors of kind " + sensorType);
      }
   }

}  // class SimpleShip




class SimpleShipManeverEVT extends SimEvent
{
   public SimpleShipManeverEVT(SimEntityImpl owner) {
      super(owner);
   }

   public void onRun() {
      ((SimpleShip)(myOwner)).doManeuverEVT();
   }

   public void onInterrupt() {}
}
```

```
class WaypointArrivalEVT extends SimEvent
{
   public WaypointArrivalEVT(SimEntityImpl owner) {
      super(owner);
   }

   public void onRun() {
      ((SimpleShip)(myOwner)).doWaypointArrivalEVT();
   }

   public void onInterrupt() {}
}
```

```
// FILE PulseTrainSim.java
import scalar.*;
import misc.*;
import simkit.*;
import simkit.javasim.*;
import simkit.awt.*;
import g2d.*;
import simkit.javasim.ammt.*;
import java.io.*;
import java.util.*;

/**
Simple Barrier Search simulation
*/

public class PulseTrainSim extends SimEntityImpl
{
   static int nTrials, trialNo, i;
   static Date startTime, endTime;
   static Histogram batchStats =
          new Histogram("Batch Means", true);
   static PrintStream       out;
   static Format mf = new Format("%12.6f");



   public static void main( String args[])
      throws Exception
   {
      if ( args.length   != 6 ) {
         System.out.println("Wrong number of arguments");
         System.out.println(
            "  usage: java CCBarrier <Number of Batches> "+
            " <Number of Trials per batch>" +
            " <Searcher Speed> <Target Speed>" +
            " <Glimpse Interval> <seed>");
         System.exit(1);
      }
      TimeMaster.reset();
      ;
      out = new PrintStream(
         new FileOutputStream("Monopulse_main_Results.dat"));

      int batches = Integer.parseInt(args[0]);
```

```
    nTrials = Integer.parseInt( args[1] );
    double ss = (Double.valueOf(args[2])).doubleValue();
    double ts = (Double.valueOf(args[3])).doubleValue();
    Time glimpse = new Time((Double.valueOf(args[4])).doubleValue());

    long seed = Long.parseLong(args[5]);
    startTime = new Date();
    trialNo = 0;
    PulseTrainSim theModel = new PulseTrainSim( ts, ss);

    theModel.targetPositRand.SetSeed(seed);
    theModel.theSensor.setGlimpse(glimpse);
    for ( i = 0; i < batches; i++){
    trialNo = 0;
    while( trialNo < nTrials ) {
       theModel.runSim();
       theModel.theSensor.reset();
       trialNo++;
    }
    System.out.println("End of batch");
    endTime = new Date();

    System.out.println("Number of Trials:           " +
                        theModel.searchSuccessStats.count());
    System.out.println("Avg Detection Rate:         " +
                        theModel.searchSuccessStats.mean());
    out.println(i + "\t" + mf.form(theModel.searchSuccessStats.mean()) +
                "\t" + mf.form(theModel.searchSuccessStats.variance()));


    System.out.println("Cumulative Run Time = " +
                      ((endTime.getTime() - startTime.getTime())/1000.0) +
                      " sec");
    System.out.println("Next Seed = " +theModel.targetPositRand.seed());

    batchStats.getSample(theModel.searchSuccessStats.mean());
    theModel.searchSuccessStats.reset();
    }

    batchStats.makeHistogramPlotDisplay(/*bin width*/0.1);
    batchStats.makeMovingAvePlotDisplay();
    batchStats.makeReportDisplay();
}
```

```
SimpleShip        searcher, target;
SimpleEnvironment theEnvironment;
Command           c;
double            d, dd,searchSpeed_,targetSpeed_;
RandomStream      targetPositRand;
DataAccumulator   searchSuccessStats;
MonopulseRadar    theSensor;
ClockFrame        clock;
Position startSearchAt;


public PulseTrainSim(double targetSpeed, double searchSpeed) {
   searcher             = new SimpleShip("USS Searcher", this);
   target               = new SimpleShip("USS Target", this);
   theEnvironment       = new SimpleEnvironment(/*170.568e-15,*/ 1.0);
   searchSuccessStats   = new DataAccumulator(
                              "Search Success", false);
   targetPositRand      = new RandomStream();
   targetSpeed_=targetSpeed;
   searchSpeed_=searchSpeed;
   d = 0;
   double dd;

   theSensor = new MonopulseRadar( "Monopulse Radar MK I",
                                   searcher,
                                   false,
                                   Length.fromNMValue(10.0),
                                   new Power(10.),
                                   1.0,  // antenna gain
                                   .96,   // integration efficiency
                                   1.37,  // antenna aperature
                                   -6.0,  // detection threshold
                                   18,
                                   6.0,
                                   this );

   theSensor.reportTo( searchSuccessStats);

   searcher.addSensor(theSensor);
   PulseTrainResponder r =  new PulseTrainResponder(target);
   r.setRCS(10000.0);
```

```
   target.addResponder( r);
   try{
      out = new PrintStream(
                new FileOutputStream("Monopulse.log"));
   } catch ( Exception e) {
      System.err.println(e);
   }

   searcher.setPosition( new Position(
                                 Length.fromNMValue(10.0),
                                 new Length(0.0),
                                 new Length(0.0) ) );
   c = new ManeuverCommand();
   c.put("Kind", "Standard Pattern");
   c.put("Pattern", "Barrier");
   c.put("End", new Position(   Length.fromNMValue(70.0),
                                      new Length(0.0),
                                      new Length(0.0)));
   c.put("Speed", Velocity.fromKnotValue( searchSpeed_));
   searcher.receiveCommand( c );

   c = new SensorCommand();
   c.put("Change", "Active");
   c.put("SensorType", "All");
   searcher.receiveCommand( c );
}


public void runSim() {

   dd = targetPositRand.Uniform(0,80);

   target.setPosition( new Position( Length.fromNMValue(dd),
                                     Length.fromNMValue(10.0),
                                     new Length(0.0) ));


   c = new ManeuverCommand();
   c.put("Kind", "Course/Speed Change");
   c.put("Course", new Bearing( 180.0, 0.0, 0.0));
   c.put("Speed", Velocity.fromKnotValue( targetSpeed_));
   target.receiveCommand( c );

   c = new ManeuverCommand();
```

```
      c.put("Kind", "Report When");
      c.put("Distance Travelled", Length.fromNMValue(20.0));
      target.receiveCommand(c);

      TimeMaster.startSimulation();
   }


   public void doEndSimEVT(Object caller) {
      target.interruptAll();
      TimeMaster.stop();
   }
}

//=====================================================
class EndSimEVT extends SimEvent
{
   public EndSimEVT(SimEntityImpl owner) {
      super(owner);
   }

   public void onRun() {
      ((PulseTrainSim)(myOwner)).doEndSimEVT(this);
   }

   public void onInterrupt() {}
}
//=====================================================
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060–6218

2. Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943–5101

3. Chairman, Code OR 1
   Department of Operations Research
   Naval Postgraduate School
   Monterey, CA 93943–5121

4. Professor Arnold Buss, Code OR/Bu 4
   Department of Operations Research
   Naval Postgraduate School
   Monterey, CA 93943–5121

5. Professor James Eagle, Code OR/Er 4
   Department of Operations Research
   Naval Postgraduate School
   Monterey, CA 93943–5121

6. Dr. William Hardenburg, Code 5740 4
   Naval Research Laboratory
   4555 Overlook Ave., SW
   Washington, DC 22193