AD-A196 610

(4)

DTIC FILE COPY

ISI Research Report
ISI/RR-88-205
March 1988

David Mizell
Yu-Wen Tung
Susan Coatney
Scott Carter
Rivi Sherman
Walid Najjar

University of Southern California

On the Design of ISI's Initial Prototype SDI Architecture Simulator

88 6 15 002

INFORMATION SCIENCES INSTITUTE ISI
213/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

ADA196610

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT This document is approved for public release, distribution is unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ISI/RR-88-205	5. MONITORING ORGANIZATION REPORT NUMBER(S) --------------
6a. NAME OF PERFORMING ORGANIZATION USC/Information Sciences Institute — 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION --------------
6c. ADDRESS *(City, State, and ZIP Code)* 4676 Admiralty Way Marina del Rey, CA 90292	7b. ADDRESS *(City, State, and ZIP Code)* --------------
8a. NAME OF FUNDING / SPONSORING ORGANIZATION SDIO/S — 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-86-K-0311
8c. ADDRESS *(City, State, and ZIP Code)* (continued on other side)	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
--------------	--------------	--------------	--------------

11. TITLE *(Include Security Classification)*

On the Design of ISI's Initial Prototype SDI Architecture Simulator [Unclassified]

12. PERSONAL AUTHOR(S) Mizell, David; Tung, Yu-Wen; Coatney, Susan; Carter, Scott; Sherman, Rivi; Najjar, Walid

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Research Report	FROM ________ TO ________	1988, March	58

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
09	02	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
discrete-event simulation; modular software design; object-oriented simulation; strategic defense architecture simulation

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This report describes the software design of the first prototype strategic defense architecture simulation system developed under ISI's "SDI Communications Research" contract. While the long-term goal of the simulation task is to investigate techniques for speeding up simulators of this type via parallelism, the system described here is a sequential program in which it must be possible to: incorporate nontrivial, executable representations of the battle management computations taking place on any defense platform within the simulation; and change the candidate defense architecture being modeled independently of the simulation software that models the external physical environment.

The report documents our approach to designing a simulator that meets those requirements. It includes both a description of the software design and some historical material about our design process: what design decisions we made, why we made them, and what alternatives were considered at the time.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] [✓ SAME AS RPT.] ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Sheila Coyazo Victor Brown	22b. TELEPHONE *(Include Area Code)* 213-822-1511	22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

8c. (continued)

Strategic Defense Initiative Organization/Systems Office
SDIO/S
Office of the Secretary of Defense
The Pentagon
Washington, DC 20301-7100

ISI Research Report
ISI/RR-88-205
March 1988

David Mizell
Yu-Wen Tung
Susan Coatney
Scott Carter
Rivi Sherman
Walid Najjar

University of Southern California

On the Design of ISI's Initial Prototype SDI Architecture Simulator

INFORMATION SCIENCES INSTITUTE ISI *213/822-1511*
4676 Admiralty Way/Marina del Rey/California 90292-6695

This research is supported by the Strategic Defense Initiative Organization under Contract No. N00014-86-K-0311. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of SDIO, the U.S. Government, or any person or agency connected with them.

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION

1.1 NATURE AND GOALS OF THE SIMULATION PROJECT

The purpose of the simulation task within ISI's "SDI Communications Research" contract is to address the basic research issues pertaining to the design and development of a simulation system capable of executing simulations of candidate strategic defense architectures at varying degrees of fidelity, for the purpose of comparatively evaluating their performance in the full systems context. The premise that underlies the project is that SDIO needs a simulation system that enables it to compare and evaluate different candidate defense system architectures at a level of abstraction that does not incorporate completely detailed models of every system component. The system should provide, however, a nontrivial model of the battle management software and include models of all major system components, in an "end-to-end" simulation, so that the manner in which the battle management software makes all of the parts of the defense system work together may be evaluated.

This is a basic computer science research project, and as such is not intended to produce a complete, refined simulation system capable of meeting SDIO's needs for end-to-end defense architecture simulation. The physical and behavioral characteristics of defense platforms and components are not known to us, even at a fairly high level of abstraction, and would have to be supplied by contractors with more detailed knowledge of candidate system and component designs. Our efforts are focused upon the following three computer science research issues:

1. We are investigating how to structure the design of a modular software system for SDI architecture simulation that meets these two design requirements that follow from the original premise:

   a. Because the intent of the simulator is to compare many designs, not just demonstrate a single one, it must be possible to change the candidate defense architecture being modeled independently of the simulation software that models the external physical environment.

   b. Because the main focus of the simulator is on systems integration issues and how the battle management system makes the many components work together, it must be possible to incorporate nontrivial, executable representations of the battle management computations taking place on any defense platform within the simulation

   Furthermore, it is desirable for the software design to allow changing the amount of detail in the simulation models of system components or the external environment without restructuring the simulator itself.

2. We conceive this simulation system as one that will be "owned" by SDIO, in the sense that it will not be designed around a particular candidate defense architecture, but instead will take a specification of a candidate architecture as input. Ideally, the user interface would be "friendly" enough to allow SDIO personnel to enter abstract architecture specifications themselves, and use the system as a personal tool for examining and comparing system design

approaches at a high level. Accordingly, we are trying to design a simple, flexible user interface that minimizes the effort necessary to specify the physical configuration and components of a defense architecture and to characterize the high-level behavior of the battle management system for the sake of simulation.

3. Given that even high-level simulations of systems this large and complex are extremely demanding in computation cycles, it is vital to address methods of distributing this sort of simulation across multiple processors for the sake of faster execution.

The first phase of this project concentrated on the first issue and developed a sequential SDI architecture simulator with what we believe to be an appropriate modular structure. This system consists of approximately 6,000 lines of C++ [1] code and executes on ISI's VAX 8650. The purpose of this report is to describe this first prototype simulator and thereby illustrate how we attacked the modularity issue.

1.2 OVERVIEW OF THE SOFTWARE DESIGN

1.2.1 MAIN DESIGN COMPONENTS

Basically, our approach has been to aim for a clean separation and a simple, carefully specified software interface between the software modules that model the components of a strategic defense architecture and those that model the external environment, with the goal of making it easy to replace either type of module. In our design discussions, we referred to this separation as that between the "simulator" and the "simulatee." We used these terms to distinguish between those parts of the simulation system that would be invariant across candidate strategic defense architectures and those that modeled a particular architecture.

Here, we were loosely using the term "simulator" to collectively refer to both the software that implements the basic discrete-event simulation mechanisms, such as the event queue and the scheduler, and the software that models those parts of the simulated strategic defense environment that do not change for different defense architectures: physical laws, geography, the characteristics of the threat, etc. In contrast, the "simulatee" encompassed just those aspects of the simulated system that the user can change, in order to represent a different candidate defense architecture: the characteristics of the defense satellites and ground stations, including the equipment assigned to each, the orbital or geographical deployment of each, and the nature of the battle management computation/communication capabilities of each.

An obviously important aspect of the software design is the interface between the "simulator" and the "simulatee." This interface consists of the simulation models of exactly those components of a defense platform that could be expected to be under direct control of the platform's battle management system, e.g., communication capabilities, weapons, sensors or any sort of propulsion devices by which the platform's velocity or attitude could be changed. We refer to these components as *technology modules*, following [2]. They are the sensors and effectors on each platform that comprise the means by which a platform can affect or be affected by the external environment. Each technology module in our simulator consists of a set of data structures and a set of functions that can be called by a platform's battle manager.

1.2.2 MAIN COMPONENTS OF THE SIMULATOR

EVENT QUEUE

As in a conventional discrete event simulator, the simulation consists of the successive evaluation of events, in the order in which those events occur. The events model the interactions between objects in the physical system; e.g., a communication message being sent from one platform to another, the launching of a weapon, or the appearance of a target within the field of view of a sensor.

The events are stored in a central event queue, implemented as a linear doubly-linked list ordered by the timestamps of the events. During the initialization phase, a set of events are placed in the event queue to model the launch sequences for the missiles.

The scheduler removes the earliest occurring event from the event queue and calls the appropriate routines that model the occurrence of this event. This may involve the interaction of a battle management system with the technology modules, which may in turn cause new events to be posted to the queue. This process is repeated until no events remain in the queue.

MISSILES

The simulator inputs a file containing a simple description of the threat scenario, consisting of a specification of the launch sequence of the missiles, where a missile is an attacking enemy ICBM aimed at some ground target. The current implementation uses a greatly simplified model of the behavior of a missile; the trajectory of a missile is simplified to 45 degrees up in boost phase, constant height above ground in the orbital phase, and 45 degrees down in the terminal phase. A missile is detectable only in the boost phase, and the success of a defense platform's shot at a missile is determined by a simple probability function that is based on the distance of the missile from the platform.

PHYSICAL MOVEMENT

The physical movement of satellites is also simplified. Currently, two types of satellite orbits are provided. In the geostationary orbit, the satellite stays over the same point on the Earth's surface. The Walker orbits are a class of orbit where the orbital period is an integral divisor of the Earth's period of rotation; they are designed to achieve a maximum density of satellites over a given point on the Earth's surface.

1.2.3 MAIN COMPONENTS OF THE SIMULATEE

PLATFORMS

C++, the programming language used to implement this simulation system, contains a construct called a "class" used to define abstract data types. We used the class construct to define a data structure that formed the basis of the "simulatee" portion of the system. We called this data structure the *platform*. Instances of the platform class are used to model each defense satellite, ground station, or any other defense component capable of individual actions that could affect the rest of the system. Each platform instance contains a unique identifier, a set of *attributes* and a set of *capabilities*. Attributes are properties of a specific platform, such as its orbital position and velocity, number of weapons expended, etc. A platform's capabilities are the set of technology modules that the user/architecture specifier has declared to be components of the platform: which types of sensors, weapons or communication systems are attached to this type of platform, and the battle management software it executes.

DEPLOYMENTS

The user inputs a specification in tabular format of the *deployment* of the defense system's platforms: initial position and (for satellites) the type of orbit. It would be worthwhile for a more realistic simulator of this type to model these assets being deployed over time, but our prototype has them all in place at the beginning of simulation.

BATTLE MANAGER ABSTRACTIONS

Our approach to the modelling of battle management computations probably distinguishes our simulator from most discrete event simulators. The simulator must take into account the fact that any defense platform can include a battle management computing system as a component, and take care of executing the relevant portion of the platform's battle management software each time an event occurs to which the platform is supposed to respond. As a result, our event handling routine somewhat resembles the scheduler of an operating system, since it must pass control to and from each simulated battle manager whenever the battle manager executes, and resume control afterwards.

Our simulator does not model battle management programs as being in continuous execution; this is not possible within the context of a discrete event simulator. Furthermore, we felt that it would be a natural and useful abstraction for the battle management computing system on a platform to be represented as being inactive until an event occurs that is of concern to it. At the simulated time at which the event occurs, the simulator activates the program that represents the platform's battle manager and it executes a code segment designed to be the response of that platform to that event. By calling a "delay" function, the simulated battle manager is also able to schedule itself for reactivation after a specified time interval independently of any other event occurring in the simulation. We use the term "BMA" to refer to these high-level, stimulus/response-structured specifications of the battle management software aboard defense platforms. It can be thought of as an abbreviation of "battle manager abstraction."

In our prototype, the BMA specifications provided by the simulation user are expected to contain C or C++ code segments. The user can declare variables within any of these segments. Our simulator distinguishes a subset of these variables, which we call *persistent variables*, or "pvars." These variables are saved and restored by the simulator between activations of a BMA, so that the writer of a BMA can safely assume that whenever a BMA is activated, each pvar contains the same value left there in the previous activation. It might be worth the memory consumption to treat all BMA variables this way, in order to simplify the user's programming model. Further experience with our prototype may clarify this issue.

We have recently implemented a first prototype high-level language preprocessor that accepts as input a battle management software specification based on this event-driven, "stimulus-response" abstraction of battle management computations and outputs a C++ program that can be incorporated into a simulation. This high-level battle management specification language and its preprocessor will be the subject of a future technical report. It is a key aspect of our approach to the second issue, that of providing a user interface that minimizes the effort necessary to specify the defense architecture to be simulated.

1.2.4 TECHNOLOGY MODULES

As part of the input to the simulation system, each platform is specified in a tabular format which denotes which technology module types are assigned to which platforms. Each type of technology module is expected to have associated with it a unique set of functions that the BMA of a platform "equipped" with that technology module can call and, usually, a set of events to which the simulator can activate the BMA to respond.

We implemented three technology modules in support of the simplified model of a defense architecture that we used to debug our prototype simulation system: a weapon, a communication facility, and a sensor.

The weapon technology module is a simple model of a laser. There was no attempt at realism in the model beyond making the probability of killing a target a linear function of its distance and requiring a delay period between successive firings of the weapon.

The communication facility allows the sending of messages between the BMAs of different platforms. Messages can be sent to specific recipients or broadcast to all other platforms. The model assumes an unrealistic ability to directly communicate between any two platforms without any storing-and-forwarding ever being necessary.

The sensor technology module we implemented is a quite abstract one which we called the "staring" sensor. It is represented to be continously looking in a particular direction with a specified viewing angle. Whenever a target enters or leaves its field of view, an activation event is posted for its BMA.

Some may find this model of a sensor too abstract even for an architectural-level simulation, because it ignores the time-dependent scanning behavior of a real sensor and assumes an infallible discrimination capability, since it sees all of the targets, and only the targets, within its field of view. Besides the trivial reply that the technology modules are designed to be replaced anyway, our counterargument is that the abstraction is a faithful one, in the following sense: In any conceivable battle manager, there will be a high-level decision-making portion of the system that is presented with a list of targets that its sensor systems have detected, so that it can decide what to do about them. This presentation will be very similar to how our sensor abstraction hands the BMA a list of targets. In other words, our sensor abstraction incorporates the scanning control, tracking and discrimination that would be part of the battle manager in a more detailed simulation model, but its interface to the BMA is quite simular to the interface that the scanning, tracking and discrimination part would have to the decision-making part of the battle management software.

This technology module abstraction discussion actually raises a quite interesting design issue with respect to the replacement of technology modules with more realistic ones. Whenever this is done, the interface between the BMA and the technology module is likely to have to change, to become more detailed. The issue of how, in the course of changing the level of detail in a technology module, to systematically adjust this sliding boundary (with its associated hierarchy of increasingly detailed interfaces) between what is represented as part of the BMA and what is represented as part of the technology module seems worthy of investigation in its own right.

1.2.5 INPUT/OUTPUT

Presently, the tables that specify platform configurations and components are manually created, but we are prototyping an interactive user interface for this purpose. The battle management algorithms are also manually inserted into the simulator, but this will be changed when we interface the high-level language preprocessor to the simulation system. As the simulator executes, it writes into a file a list of events that have occured in the simulation in order of the simulation time at which they occured. We perform postprocessing on this event history file after completion of the simulation run in order to maximize user flexibility in deriving information from the event history. In this postprocessing step, the event history data is inserted into a database management system. The user can then query this database in many different ways. We have also successfully prototyped a graphics software package which extracts information from the event history database and can represent position and motion of platforms and targets over a specified interval of simulation time as seen from a specified point of view. This graphics package is built upon the orbital display software developed within the graphics task of the same SDI Communications Research contract [3].

1.3 PLANS FOR FURTHER WORK ON THE SIMULATOR

The ISI simulation effort is a basic computer science research project, and as such is not intended to produce a complete system suitable for detailed SDI architecture studies. That would require a larger-scale development effort that is more suited to be undertaken by defense industry software development organizations. Our goal as computer scientists was to address the most difficult design problems inherent to this kind of simulation software system in such a way that our solutions to them were reflected in the overall design of this prototype. The measure of our success will be the degree to which more detailed models of defense architectures, system components, and the physical environment can be added to the simulator without necessitating changes to the overall structure of the simulation software.

We are currently working on a partial test of the hypothesis that our software design will be robust under the addition of more detailed models, by specifying a few, more detailed defense architectures, accompanied by some more detailed technology modules. Additionally, we intend to implement a prototype of our design for an interactive "front end" that supports the specification of the configuration and components of a defense architecture. This will incorporate the high-level battle management specification language preprocessor mentioned in the previous section. We also plan to prototype new simulation data output collection and reduction facilities.

We intend for this phase of the research to culminate in a sequential simulation prototype that is a "finished product" in the sense that it should be feasible for a team of software professionals to input their own architectural specifications to our simulator and revise or replace our technology modules or environmental models without hands-on assistance from us. In the future, we plan to focus our attention on the remaining and most challenging research issue for simulators of this kind: how to improve their execution performance via parallel execution.

1.5 ORGANIZATION OF THIS REPORT

The rest of this research report contains what we call our "Design Notes." This text was produced and updated throughout the design and implementation of the sequential simulator prototype. It is of dual intent: first, to provide a high-level description of the software design and second, to provide a record of the design decisions that we made along the way, with information about why we made them, what alternatives were considered, etc. The source code of the main program of the simulation system and of the simple battle management programs we used for debugging are included as an appendix, along with a copy of the documentation maintained along with them in our source files. They illustrate, in particular, the manner in which control is passed between the simulator and the battle management programs provided by the user for execution within the simulation system.

1.6 ACKNOWLEDGMENTS

Several of the simulation interface design approaches used in this prototype were suggested by Danny Cohen, in earlier drafts of [2].

CHAPTER 2. EVENTS AND THE EVENT QUEUE

As in most discrete event simulators, each event is is represented in our simulation system as a data structure that contains variables indicating the name of a function to execute, the time at which to execute this function, and the arguments to be passed to this function.

Such events are executed chronologically to simulate the real world activities, and the simulation clock is advanced to agree with the timestamp of the event being simulated. The collection of events to be executed is called event queue. Events are removed from the event queue according to the event time.

This chapter describes and discusses the design of such events and the operations of the event queue, in the form of question-answering. The questions are:

- How is the event queue used at the top level in the main program?
- What are considered to be events?
- What are the data structures of event and event queue?
- What if two events have the same time stamp?
- How are all these events and functions interrelated?
- How does control switch between a BMA and the simulator?
- How are events initialized?

2.1 How is the event queue used at the top level in the main program?

In the simulator main program, there is a event scheduler that is mainly a "get-evaluate-put" cycle of events. At each cycle, one event is popped from the event queue and is executed by some function according to the event type. During the execution, the states of simulated objects may be changed, and more events may be created. The loop continues until the event queue is empty (at which time the simulation is done).

The main loop in the main simulator program looks like this:

```
while (current_event get_next_event() )  // while queue is non-empty
{ switch (current_event->get_event_type())
        { case (SHOOT): // ... execute_shooting()
          case (BMA_SENSE): // ... call BMA
         // .....
        }
}
```

2.2 What are considered to be events?

In our simulator, events include:

1. BMA related activities:
   (a). interaction between a BMA and the world: BMA_DELAY, BMA_INIT;
   BMA_DELAY causes the BMA instance to be (re)activated after delay specified in simulation time units. **BMA_INIT** causes the BMA instance to execute a specific code branch which contains all user-level initialization code for that BMA.
   (b). interaction between a platform/BMA and its devices: BMA_SENSE, BMA_COMM;
   BMA_SENSE occurs when a sensor detects a change in one or more of its targets

state. **BMA_COMM** occurs when a [inter-platform] communication message arrives at a platform.

2. attacking-missile-related activities:
   (a). interaction between a missile and a sensor: LAUNCH, PDESTROY (also see 3(b) below);
   LAUNCH causes a missile (ICBM) to be launched from its carrier. **PDESTROY** indicates that a missile has probably been destroyed by a weapon.
   (b). interaction between a missile and the defense system: PNUKE;
   PNUKE occurs at the time that a missile is expected to reach its target. If the missile has not been destroyed by the time of the event, a nuclear explosion occurs.
3. weapon-related activities:
   (a). interaction between a weapon and the world: SHOOT;
   (b). interaction between a weapon and a missile: PDESTROY;
4. sensor-related activities:

(a). interaction between a sensor and a target: SENSE_APPEAR, SENSE_DISAPPEAR, SENSE_BLOWNUP;
(b). interaction between a sensor and its target table: SAMPLE;

5. physical-world-related information:
(a) interaction between the world and the simulator: PRINT_POS;
PRINT_POS occurs at a user-specified frequency over a user-specified interval of simulation time, and causes position data for missiles and platforms to be saved for later analysis and display.

2.3 What are the data structures of an event and the event queue?

The basic data structure of an event has the following fields:
(1) time stamp: a field that contains the time the event will occur;
(2) type: the type of the event;
(3) description: a type-dependent description of the event content.

In the case of a BMA-type event, the field of description contains the information of the BMA instance ID and a label where control should go to (in the BMA code) when this event is executed.

The event queue is structured as a doubly linked list of all events to be executed. These events are ordered by their time stamp.

There are two major event queue operations: **insert**, and **get_next_event**. The **insert** function inserts the current event in the queue at the point where all events after it have later time, and all events before it have earlier or equal time. The **get_next_event** function pops the first event from the queue and updates the head_event_queue variable.

2.4 What if two events have the same time stamp?

If two or more events share the same time stamp, the function **insert** orders these events by means of their priorities (or type numbers) -- a lower number means a higher priority, and will thus be inserted earlier in the event queue.

This is the priority of events starting from the highest one:

1: SHOOT,
2: PDESTROY,

3: BMA_SENSE,
4: SAMPLE,
5: SENSE_APPEAR,
6: SENSE_DISAPPEAR,
7: SENSE_BLOWNUP,
8: LAUNCH,
9: PNUKE,
10: BMA_DELAY,
11: BMA_COMM,
12: BMA_INIT,
13: PRINT_POS.

This order is assigned arbitrarily except that BMA_SENSE and SAMPLE events must have higher priority than SENSE events. This is because of the sampling logic:

> SAMPLE event represents a collection of one or more SENSE events occurred a little earlier, and the BMA must be interrupted by the SAMPLE event via BMA_SENSE event in order to respond to the information those earlier SENSE events provide. Since each sampling period must be done completely before another can start (the logic is explained in detail in Chapter 5), BMA_SENSE and SAMPLE must be executed before SENSE events which belong to the next sampling period.

2.5 How are all these events and functions interrelated?

Events and those functions called to execute the events are sometimes related to each other.

The event PRINT_POS works quite independently: it calls the function **print_objects** and re-schedules itself to recur after a user-specified time period.

BMA events events activate corresponding BMA sections, and are posted by functions such as **sample**. They have otherwise little interaction with other events and functions.

All other events and corresponding functions are closely related; their interrelationship is described by the diagram in Figure 1. This diagram is divided into three columns: missile, sensor and weapon. In each column, the right-hand-side has event names (in capital letters) and the left-hand-side has function names (in lower-case letters). A line between an event and a function indicates their association. An arrowhead indicates the calling sequence of a function and an event; e.g., "A --> B" means A calls B, where one of A and B must be a function and the other can be either a function or an event. Words marked near a connection line is the condition of the calling asssociation between the function and the event (e.g., "A --C--> B" means A calls B if C holds), or the mapping of the function calls (e.g., "1 to many" means a one-to-many mapping, or one function may create multiple events).

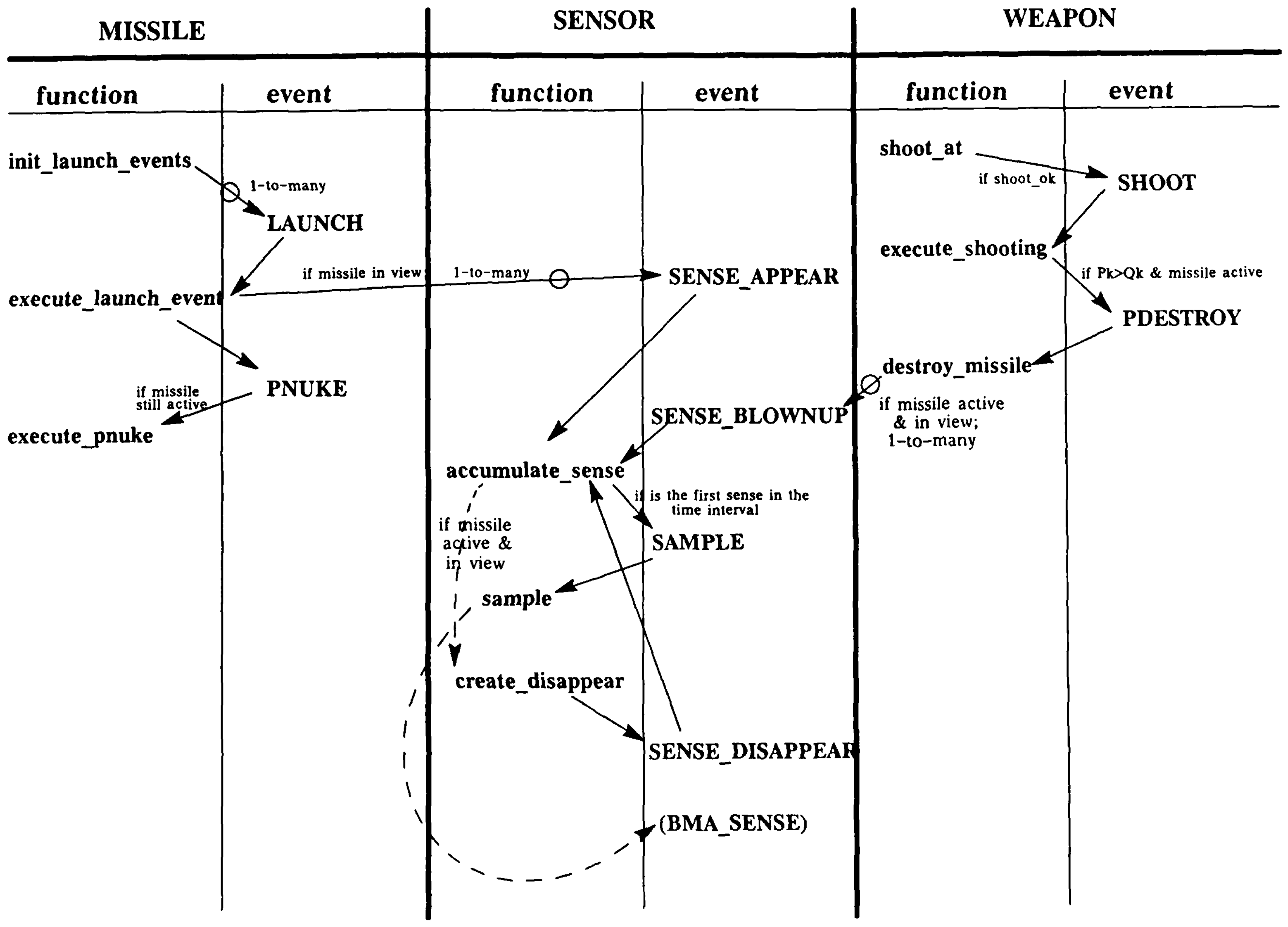


Figure 1. Events

2.6 How does control switch between a BMA and the simulator?

In the case of executing a BMA event, part of the user-defined BMA program would be used as an external function to the simulator main program. Since the BMA program is not pre-defined, the major issue here is how to switch control from the simulator to the desired location of the BMA and switch control back when it is done.

The strategy used is:

- a BMA event is set up with a BMA branching address consisting a BMA instance ID and a BMA label. This BMA branching address is put into the event structure by the function "delay" in the BMA program. When this BMA event is executed,
- the BMA ID is retrieved by function **get_platform_id**;
- the BMA label is retrieved by function **get_label**; and
- a set of persistent variables for that platform instance is retrieved by a function **get_pvar_ptr**; this set of persistent variables may be updated by a **put_pvar_ptr** function; then
- the BMA function is identified by a pointer to this function found by **get_bma**; and the ID, label and persistent variables are arguments passed to this BMA function.

Therefore, the following statement in the simulator program will pass control to the appropriate BMA function at the specific location: (*bma_fun)(label, pid, ptr_persistent).

The BMA function names should be provided by the user, but they are hard-coded as "bma0," "bma1," and "bma2" in the current implementation. Labels may be assigned freely by the BMA programmer, except that "label 0" is reserved for an initialization routine (see Chapter 8).

A case statement in this BMA program switches the control to the corresponding piece of code:

```
switch(label){
        case LABEL0: // initialization

        case LABEL1:  // do something (sense, fire a weapon, etc)
                      // which may create zero or more events;
        // ......
         }
```

and the case statement also ensures that control will go back to the main loop in the simulator program automatically.

2.7 How are events initialized?

In the current implementation, the function **init_event_queue** initializes the event queue to call all instances of BMAs at time 0 with label 0. In other words, it is assumed that all BMA instances are activated simultaneously at time 0, and all BMA programs will initialize their capabilities (e.g., communication channels, sensors).

This initialization scheme would be more complicated but more realistic if each BMA instance is activated in a certain deployment sequence. This deployment scheme or other alternatives may be implemented in later versions of the simulator.

CHAPTER 3. BATTLE MANAGER ABSTRACTIONS

This chapter describes the battle manager abstractions, or BMAs, which are the part of the "simulatee" that models the computations of the defense architecture's battle management/C3 system.

From a simulator programmer's point of view, BMAs can be viewed as external functions, to be provided as input by the defense system architect.

Each platform has one BMA which may or may not be the same as another BMA. Each BMA is a function called with the following arguments:
1. label (entry point of the case statement),
2. platform_instance_id,
3. persistent_variables.

3.1 Label 0 is assigned as the initialization label

In the main "case" loop, there must be at least two case statements in order for the platform to be useful. One of these two labels must be 0 (or its equivalent, LABEL0), and the corresponding case statement is meant to initialize all capabilities in this platform. The reason that label 0 is reserved for initialization purpose is because all BMAs are called at simulation time 0, with label 0, when the simulator event queue is being initialized. Once the BMA high-level language preprocessor is incorporated, the initialization step will be completely hidden from the user.

3.2 Other labels represent events relevant to the BMA

The sensor, weapons, communication, and propulsion capabilities associated with a platform constitute its links to the simulated outside world. It is assumed that each such technology module has one or more events related to it that are of significance to the BMA. When one of these events occurs, the simulator *activates* the BMA by passing control to it at the label associated with that event.

3.3 A BMA example

A (simplistic) example BMA program suggested by David Mizell was implemented in our simplified debugging model. In this example, there are three kinds of platforms each uses a BMA: HQ, GEO and CV, where HQ stands for *headquarters platform*, GEO stands for *geosync sensor platform*, and CV stands for *carrier vehicle platform*. This example is described below in the form of a set of production rules that roughly correspond to the syntax of the high-level language for specifying BMAs that will soon be incorporated into the simulator's user interface. Each left-hand-side of a rule represents an event that causes the simulator to activate the BMA so that it can respond to the event, and the right-hand-side is the code that the BMA executes in response.

(1). HQ BMA:

```
receive target data from some GEO -> broadcast "weapons release"
                                         message
```

(2). GEO BMA:

```
PEACE: targets appear -> send target data to HQ
       receive "weapons release" from HQ ->
```

```
                        { broadcast target  data to CVs;
                        state := BATTLE
                        }

BATTLE: target appear -> broadcast target data to CVs
        target disappear -> broadcast target data to CVs
```

(3) CV Weapon Platform BMA:

```
PEACE: receive "weapons release" from HQ -> state := BATTLE

BATTLE: receive GEO sensor data ->
                        { update target list;
                          while (shootable targets left and shots left) do
                               { pick target;
                                 shoot;
                                 update count of number of shots left
                               }
                        }
```

CHAPTER 4. MISSILES

The missile package defines the behavior of a missile. A missile is defined as an attacking enemy ICBM aimed at some ground target.

In this chapter, we describe the background of our design by answering the following questions:

- What are boost, orbital and terminal phases?
- How do we simplify the missile behavior?
- How do we determine whether or not a missile is in the view of a sensor?
- What are the interrelationships between missile events and functions?

4.1 What are boost, orbital and terminal phases?

There are generally considered to be four phases in the launch-to-impact sequence for ICBMs [4,5]:

(1). *boost phase*: the initial phase which is characterized by transit through the atmosphere. The hot rocket exhaust make the missile easy to be detected by sensors. This phase lasts for a few minutes;

(2). *post-boost phase*: the rocket ejects the nose cones and protective shrouds. It then deploys the bus which houses the warheads and decoys. During this phase, some five minutes, the missile is still in one piece and is still an attractive target, but less detectable;

(3). *mid-course phase*: thousands of warheads and decoys are deployed. It takes some twenty minutes;

(4). *terminal* (or *reentry*) *phase*: the missile begin to enter the atmosphere on the way to the target. Warheads and decoys can be easily detected, but time is very short (a few tens of seconds).

In our simplified debugging model, we do not distinguish between (1) and (2) -- both are called "boost phase."

4.2 How do we simplify the missile behavior?

The behavior of a missile is simplified in our simulator in many ways. Some basic assumptions, among others, are listed here:

1. the trajectory of a missile is simplified to: 45 degrees up in boost phase; constant altitude in orbital phase; and 45 degrees down in the terminal phase; one constant velocity in boost and terminal phase and a different constant velocity in terminal phase;
2. a missile is detectable only when it is in the boost phase;
3. the success of a defense platform's shot at a missile is determined by a simple probability function (see Chapter 5).

4.3 How do we determine whether or not a missile is in the view of a sensor?

The logic is simplified. First of all, in our simplified debugging model, a missile is visible only in its boost phase. Next, if a missile is seen at its launch point. it is seen throughout its boost phase.

There are two functions to determine whether or not, and when, a missile has been detected by a given sensor. These two functions are **inview** and **outview**.

The function **inview** checks if the launch point of the weapon is in the range of the sensor. First, the function checks if the sensor's viewing cone covers the whole hemisphere; if so and if the launch point is in that hemisphere, or if not but the launch point is in the cone projection area, then the missile will be in view starting a little while after the launch time.

The function **outview** estimates the time a missile goes out of the view of the sensor by adding the boost phase time to the "in the view" time. The missile will be in the view until the end of its boost phase or until until it is destroyed by a weapon.

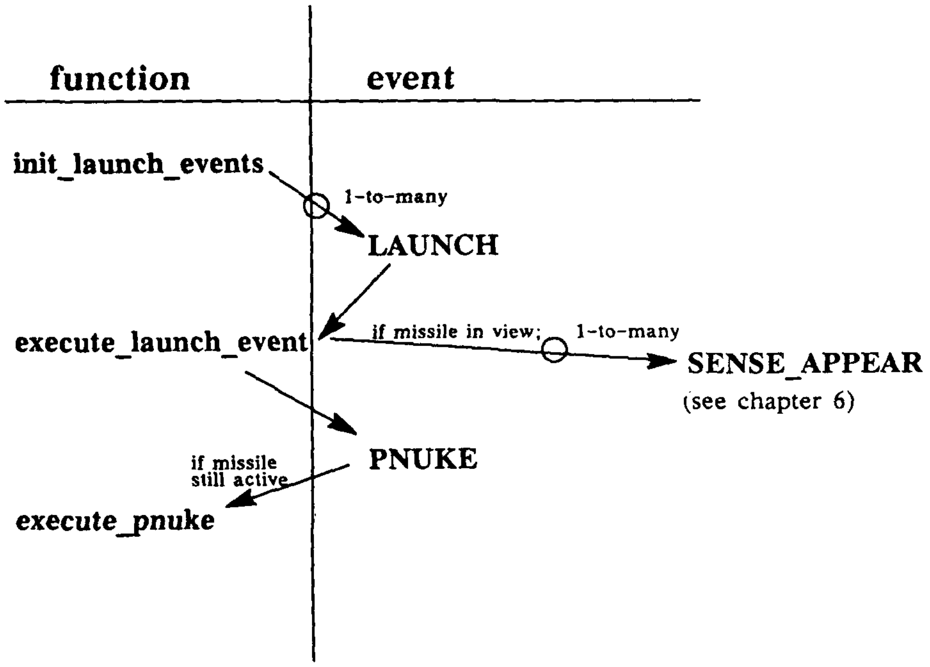
4.4 What are the interrelationships between missile events and functions?

There are two events and three functions defined and interrelated in this package. The two events are LAUNCH and PNUKE; the three functions are **init_launch_events**, **execute_launch_event**, and **execute_pnuke**.

The function **init_launch_events** initializes all LAUNCH events according to predefined times in a threat definition file. Each LAUNCH event is then executed by the function **execute_launch_event**. This function uses the function **inview** to determine if any sensor can see the launch and posts SENSE_APPEAR events for all such sensors. This function also posts a PNUKE event for a possible hit on the missile's target at the prediceted time of impact.

A PNUKE event is executed by the function **execute_pnuke** which returns true if the missile is still active at the time it is supposed to reach its destination.

The interrelationship is described in the diagram below:



CHAPTER 5. WEAPONS

A weapon is defined in chapter 1 as a capability of a platform. Weapons are used by defense platforms to shoot at targets (attacking missiles). The class of weapons might include various kinds of lasers, particle beam weapons, kinetic energy weapons, etc. We only implemented a simple model of lasers in our first prototype.

This chapter addresses the following detailed issues of weapon package design:

- How do we simulate the shooting mechanism?
- How do we determine if a shot is effective? (What are Pk and Qk ?)
- Can a sensor see a weapon?
- Does a weapon have a special trajectory?
- How are events and functions interrelated?

5.1 How do we simulate the shooting mechanism?

Firing a weapon can be done at any time as long as the following constraints are met:

(1). availability: that is, there is one or more weapons left, or in the case of a laser, there is enough energy to fire another laser shot;
(2). operational delay: A finite time is required to load or aim a weapon, or, in the case of a laser, to recharge the laser device.

The function **shoot_ok** checks these constraints and returns true only when the operative contraints are met, in which case the function **shoot_at** posts a **SHOOT** event scheduled to occur after a delay (currently, a constant delay regardless of the distance between the platform and the target).

In our simplified debugging model, we assume any available weapon can be fired without malfunctioning. However, a "sure fire" weapon is never a "sure kill" one. The uncertainty of the kill depends on the distance between the weapon and the target, the killing-energy carried by the weapon, and also errors and fuzziness of the target information, errors from the estimation of the position and time of the target, and errors from the environment (deflection due to the weather or the surface material of the attacking missile, and so on).

5.2 How do we determine if a shot is effective? (What are Pk and Qk ?)

We determine whether a shot is successful by means of two probabilities: **Pk** and **Qk**. **Pk** is the probability of the shot being successful.. In the current implementation **Pk** is only a function of the distance between the weapon and the target. A more detailed implementation might include effects of target size, IR brightness and cross-section of the target, angle of intercept, accuracy of tracking, etc. **Qk** is a random on the interval (0,1]. By doing this, we are able to say a shot is an effective kill if and only if Pk > Qk.

The evolution of this idea is described below. Originally, we had only Pk, the probability of a successful shot. Each time the weapon was fired, the shot was considered to be successful if the probability of a successful shot was greater than some constant (epsilon):

```
if Pk() > epsilon, shot was successful
```

The following objection was raised: suppose Pk() returned the value 0.9 on 10 different occasions. One would then expect that 9 of those 10 shots would be successful. However, if

epsilon was set to 0.8, then all of those shots would be successful. The solution proposed was to roll a random number in the interval (0...1], and compare this number (returned by Qk()) against the number returned by Pk():

if Pk() > Qk(), shot was successful

The function execute_shooting does all that, and if Pk() > Qk() it posts event PDESTROY. The execution of event PDESTROY is then carried out by function destroy_missile which will kill the target.

5.3 Can a sensor see a weapon?

Our existing simple sensor models do not see weapons, and our existing weapon models do not communicate, so the result of firing a weapon is not detectable. However, a BMA can execute a time-out algorithm to determine whether a particular target has disappeared when it should.

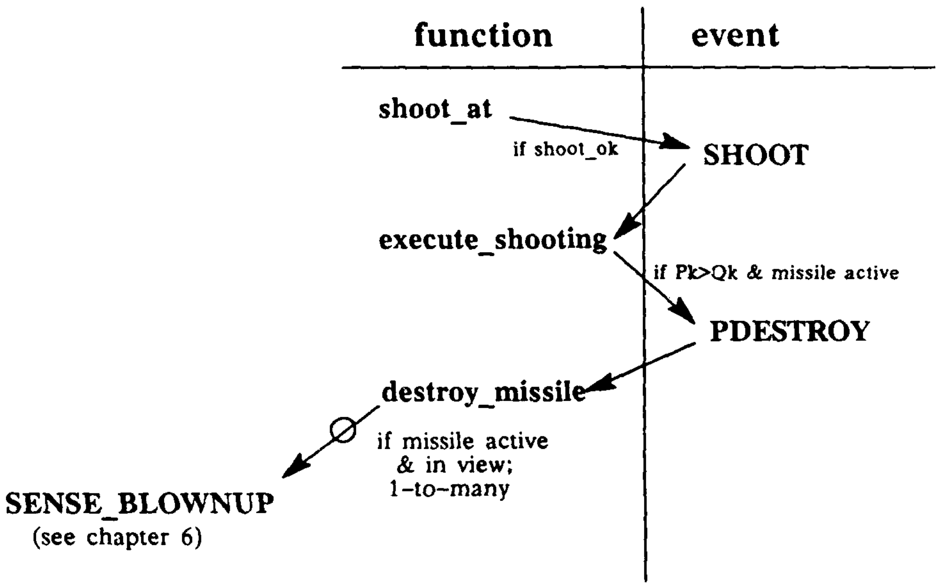
If the shot was a successful kill, the function **destroy_missile** marks the missile as "destroyed" and checks which if the destruction event occurred in the field of view of any sensor. A SENSE_BLOWNUP event is posted for each of those sensors. It is possible for multiple **PDESTROY** events to be posted for the same missile; however, only the [temporally] first **PDESTROY** has any effect.

5.4 Does a weapon have a special trajectory?

No. Unlike the missile, the trajectory of a weapon is not specified at all, only the location of the target of the weapon is specified.

5.5 How are events and functions interrelated?

The following diagram shows the interrelationship between two events (**SHOOT** and **PDESTROY**) and three functions (**shoot_at**, **execute_shooting**, and **destroy_missile**). They are as explained in previous paragraphs.



CHAPTER 6. SENSORS AND TARGETS

A sensor is a capability of a platform. A sensor may send a pulse of energy in a certain direction and get a return (target information) from it, or perhaps just passively receive energy (e.g., IR radiation from a boost plume) from a certain direction. The field of view of a sensor can be either time-variant (a "flashlight" sensor) or time-invariant (a "staring" sensor). Presently, only "staring" sensors are implemented.

A target corresponds to an intruding object (such as ICBM) detected by a sensor. The target data reflects only what the sensor sees, not necessarily the "truth." In other words, target information may include some errors or fuzziness.

In this chapter, we will answer these questions:

- Why do we assume staring sensors?
- What are SENSE events?
- Why do we need the SAMPLE event?
- How do SENSE, SAMPLE events and their related functions interact?
- What is needed to prevent SAMPLE from failing?
- How do we select the time interval D?
- Why do we care about the longitude, but not the latitude of a staring sensor?
- What is the angle of a sensor?
- What kind of target information does a sensor provide?

6.1 Why do we assume staring sensors?

We had considered two types of sensors:

(1). *flashlight sensors* (this term was coined by Scott Carter and originated from Dick Lipton's "mosquitos and flashlights" phrase):

> The sensor takes a "snapshot" of the targets in a particular direction at a particular time. To use such sensors, the user of this simulator has to include a non-trivial tracking algorithm in his BMA.

(2). *staring sensors* (term coined by Scott Carter):

> This type of sensor is thought of as continuously sensing in a certain direction and continuously maintaining a track list of all targets within its field of view.

In the real world, a sensor may not be able to cover a large enough space and it may have to scan from one end to another in order to catch the target (i.e., a flashlight sensor). We designed the staring sensor based on our general design philosophy -- the philosophy of not requiring the user to provide all "ten million lines" of BMA before he can use the simulator. The staring sensor abstraction contains an abstraction of what, in a real system or a more detailed simulation, would be part of the battle management system: the tracking algorithm that builds and maintains the track file.

6.2 What are SENSE events?

A staring sensor is able to inform the BMA whenever a new target enters, leaves, or is destroyed in its field of view. Each such instance can be simulated by a SENSE event.

There are three types of **SENSE** events: **SENSE_APPEAR**, **SENSE_DISAPPEAR**, **SENSE_BLOWNUP**. These events indicate the appearance, disappearance and destruction of a missile in the field of view of the sensor, respectively.

There is one closely related type of event, **SAMPLE**, in this sensor package.

6.3 Why do we need the SAMPLE event?

Ideally, a **SENSE** event can be realized by causing an activation of one branch of a particular BMA instance. However, since the BMA instance would be activated by for each **SENSE** event, it would react as if it never detected more than one sensing event at the same time -- but in the real world, a BMA may behave differently when multiple targets present and when only a single target presents. In other words, the above one-**SENSE**/one-activation scheme does not simulate very well the case of simultaneous, or nearly simultaneous, appearance/disappearance of multiple targets in the range of the sensor.

To solve this difficulty, we decided that the BMA activation should occur for all **SENSE** events that happen simultaneously or nearly simultaneously. The logic was changed so that all **SENSE** events occurred in a certain time interval are accumulated, and the BMA instance is only activated at the end of each such time interval.

Thus, a **SAMPLE** event is needed to implement both the accumulation of **SENSE** events and the activation of the BMA instance at the end of a time interval.

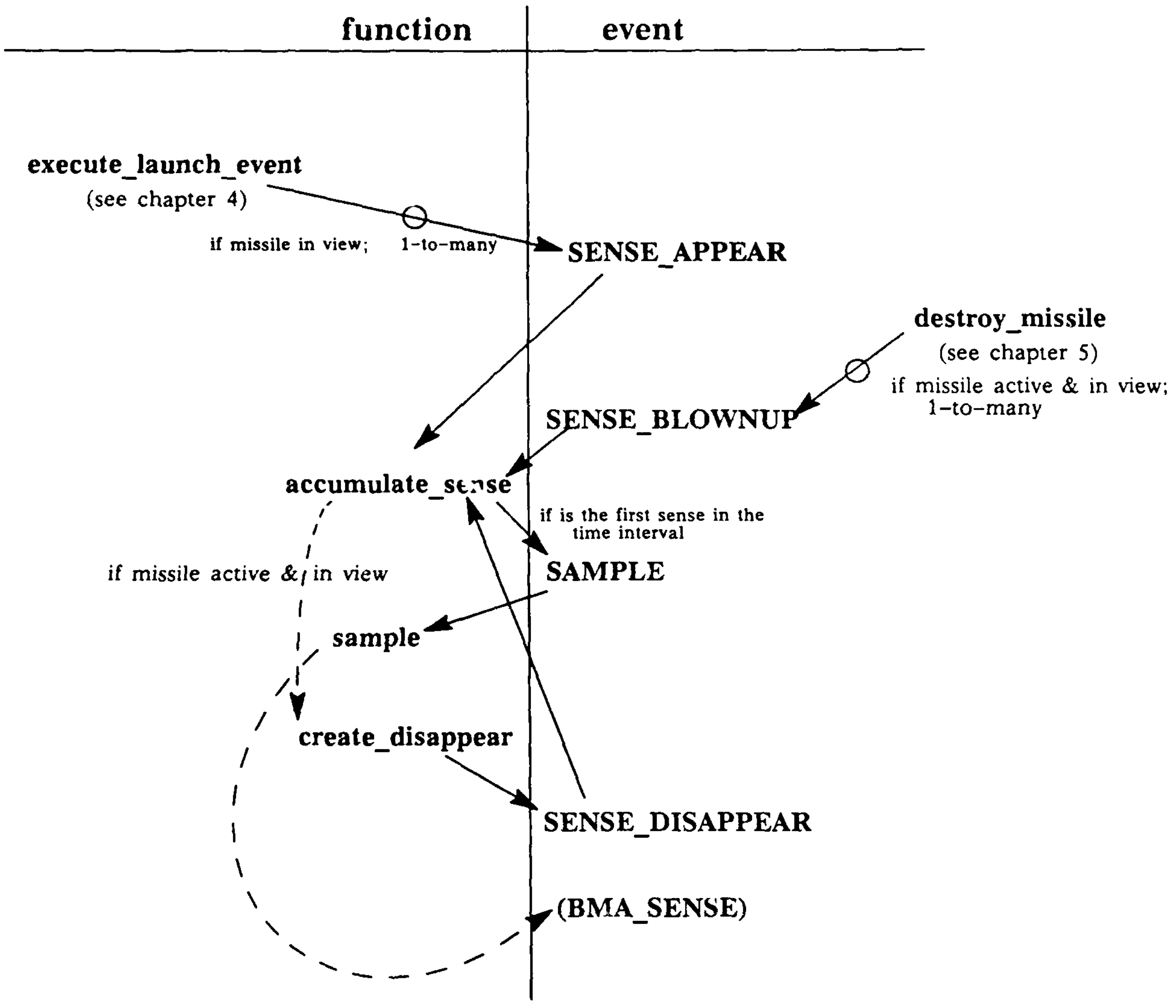
The logic involved is that all sensing events are accumulated along D time units by the function **accumulate_sense**, and a **SAMPLE** event is posted if and only if some target(s) appeared, disappeared, or were destroyed at time t, (n-1)D <= t <= nD. At the end of that time period, nD, the **SAMPLE** event is executed by the function **sample**, and a **BMA_SENSE** event is posted which will activate the BMA instance.

The BMA in turn calls the function **get_target** which returns a list of targets which have appeared, disappeared, or been destroyed in the time interval [(n-1)D,nD). It would then be the responsibility of the BMA to decide what to do with these targets.

It is clear that exactly one SAMPLE event will be posted for each time interval as long as the number of the SENSE events is no less than one. In the case that there is no SENSE event in the time interval, there will be no SAMPLE event.

6.4 How do SENSE, SAMPLE events and their related functions interact?

The details of the interrelationships between the events **SENSE** and **SAMPLE**, and the functions **accumulate_sense**, **sample**, and **create_disappear** are shown below, where **execute_launch_event** and **destroy_missile** are functions found in missile package:


function
event
execute_launch_event
(see chapter 4)
if missile in view; 1-to-many
SENSE_APPEAR
destroy_missile
(see chapter 5)
if missile active & in view; 1-to-many
SENSE_BLOWNUP
accumulate_sense
if is the first sense in the time interval
if missile active & in view
SAMPLE
sample
create_disappear
SENSE_DISAPPEAR
(BMA_SENSE)

The function **accumulate_sense** has the following logic:
(1.) if the calling SENSE event is the first one in that time interval, post a SAMPLE event;
(2.) for the SENSE_APPEAR event (the first SENSE or not), figure the time the target will go out of view by calling function create_disappear.
(3.) for all types of SENSE event, append the sensed target to the target list.

The function sample has the following logic: for each target on the target list, if it is still alive, post a BMA_SENSE event to activate the BMA.

When the BMA is activated, it calls the function **get_target** and gets target information from this function, and it may do something about those targets -- such as firing some weapons, sending its information to other platforms, etc.

6.5 What is needed to prevent SAMPLE from failing?

Current implementation of the above one-SAMPLE/one-interrupt scheme is not robust, however. This is because the logic relies on the use of function **get_target** in the BMA whenever a BMA_SENSE is called. If the user does not provide the call to function **get_target**, the whole logic fails.

Recall the first step of function **accumulate_sense** mentioned above, the condition of "first SENSE event" in the time interval is determined by the emptiness of the current **target_table** for that sensing platform. Since the empty **target_table** entry is the result of executing the **get_target** function, a SAMPLE event may never be created and the BMA will never be interrupted again if the **get_target** function is missing from the BMA code.

The user should be warned with this potential problem and should always remember to use the **get_target** function whenever there is a BMA_SENSE event. To solve the problem forever, we should revise the code such that the condition of the first SENSE event in that time interval is detected by an independent boolean variable, but not by the emptiness of the **target_table**.

6.6 How do we select the time interval D?

The length of each sampling time interval, D, should really be determined by the platform capability. This D represents the time interval between updates of a BMA of new information from its sensor.

Therefore, if D is selected small, the BMA has relatively little amount of time to respond to the possible targets. To the extreme of this, the BMA simply has no time to respond, or it is forced to postpone its decision-making for a certain amount of cycles -- but this is a duplicate of the sampling scheme and is, of course, not desirable.

On the other hand, if a large D is selected, the BMA may lose response time, and possibly lose some accuracy in its target tracks. So it is important to select a realistic time interval D. In practice, this D should be provided by the user, but for now it is arbitrarily set at one second.

6.7 Why do we care about the longitude, but not the latitude of a staring sensor?

In the class **staring_sensor** a sensor is described by its longitude, view angle, and the ID of the platform that owns the sensor, but not by its latitude. This is because in our simplified model, all sensors are assumed to be mounted on geostationary platforms which must be above the equator all the time, so the latitude of any sensor is always known (latitude = 0). This assumption was made only to simplify the initial implementation.

6.8 What is the angle of a sensor?

The "angle" of a sensor means how wide a field a sensor can see, that is, the angle of the sensor's viewing cone. A sensor is assumed to "stare" at the earth in the normal direction, so the "angle" defines a unique viewing cone. It can be used to determine whether or not a missile can be seen by the sensor at a given time (refer to Section 3.3).

6.9 What kind of target information does a sensor provide?

Each platform keeps a copy of its **target_list** which describes what this platform sees (via its sensor).

Each target list may have a list of missile information, each of which include the missile ID, a trajectory, target state and the time at which the target entered its current state (e.g. visible).

A target is always in one of its three states: **INVISTOVIS**, **VISTOINVIS** and **VISTOBLOWN**. **INVISTOVIS** means thast the target is currently in a visible state, having previously been invisible (from being out of the sensor's field of view, beneath cloud cover, etc.). **VISTOINVIS** means that the target is not currently visible to the sensor, either from exiting the sensor's field of view or from the missile ending its boost phase (i.e., the missile is treated as effectively invisible once it has deployed reentry vehicles and decoys). **VISTOBLOWN** means that the missile has been destroyed (as a result of a weapon firing) in the field of view of the sensor (it is assumed that the sensor can differentiate between a destructuion event and the normal behavior of the missile).

In our simplified debugging model, the target data provided by a sensor model is identical to "simulation truth," that is, the actual trajectory parameters of missiles used by the simulator modules that implement the physical motion of missiles. The simulation software is structured, however, so that a function could be included which introduces some "noise," or discrepancies between the "true" target data and that returned by a sensor model.

CHAPTER 7: COMMUNICATION

Our simulator treats communication as message passing between platforms.

In this chapter, we will try to answer the following questions:

- Who can talk and who can listen?
- What are the space and time constraints for communication?
- How does a BMA use the communication mechanism?
- What does the simulator need to do?
- What does the post office do?
- What is the timing calculation in the post office model?
- What is the difference between letters and messages?
- In what order are messages received?
- Can the receiver sort the incoming messages?

7.1 Who can talk and who can listen?

When we modeled the communication behavior of the platforms, we made the following assumptions on their ability to talk or listen:

1. Those platforms that have the communication capability may both talk and listen to each other.
2. A communication platform can send a message to one, a set of, or all other platforms at a time.
3. A communication platform can receive any number of messages at any time.
4. A communication platform can send and receive messages via several different channels.

7.2 What are the space and time constraints for communication?

As far as space is concerned, we assume that:

1. there is no limit to the distance a message can travel;
2. there is no limit to the length of a message; and
3 a platform has unlimited buffering space for incoming messages.

However, a longer travel distance and a lengthier message do imply a lower probability of successful transmission. This is implemented by a probability function Pc (suggested by Danny Cohen): $Pc = 0.997 * \exp(-L/L0) * \exp(-R/R0)$ where L is the length of the message, R is the communication distance, and L0 and R0 are constants set to 5,000 bits and 20,000 km respectively. For simplicity, the current implementation sets L to 0.

As to the time constraints, we assume that:

1. the send operation takes time ST;
2. message traveling takes time DT;
3. nothing else takes any significant time.

7.3 How does a BMA use the communication mechanism?

In our model, a BMA can perform the following communication operations:

1. SEND message M to platform(s) Y via channel(s) CH, and (optionally) activate the sender when the message is out.
2. BROADCAST message M to all platforms via channel CH, and (optionally) activate the sender when the message is out.
3. SET the activation flag for this platform and this channel, and go to this label (in BMA code, usually followed by a GET) when the activation event occurs.
4. GET the first message available for this platform and this channel.

A platform has to signal "GET" in order to get a message from its own message buffer. If the receiver has requested to be activated upon the receipt of a message, an incoming message will cause the activation of the receiver when it is the time to signal a "GET."

The decision that the sender can also be activated was made because we wanted both send and receive functions in a BMA program to return immediately. Since a send function takes some time to complete the sending operation, the sender may want to be activated when the send operation is completed.

7.4 What does the simulator need to do?

Inter-platform communication is modeled by a set of data structures and functions that are needed in addition to the above send and receive functions. These data structures and functions are mainly used to

1. store the message M sent from source X;
2. calculate sending operation time ST;
3. calculate message traveling time DT;
4. compute the probability of successful transmission Pc, and if this Pc is high enough, route the message to its destination and interrupt the receiver when so requested; and
5. activate the sender when so requested.

We call this set of data structures and functions the "post office," because its functions can be viewed as postal service functions.

7.5 What does the post office do?

The post office is designed as a 2-D array, in which each element is a mailbox, one for each channel per platform. An outgoing message goes directly to the post office, and after some processing (simulated by some delay time) it will be delivered to the destination mailbox. When there is an incoming message, the destination platform will be activated by the mail-man, if the activation flag on that mailbox is set. Whether the activation flag is set or not, the receiver must go to his mailbox in order to get the mail.

7.6 What is the timing calculation in the post office model?

If the sender send a message at time T, it is assumed that the message is out at T+ST, where ST is the sending operation time delay. The sender gets activated at T+ST if the flag is set.

For simplicity, it is assumed that this ST is independent of the number of destination platforms. In other words, we assume that the time needed to send a message is fixed whether the message is going to one or to many other platforms.

The time ST is calculated by **comm_msg_queue::send_op_time**. In the current implementation, ST is set to zero. But we could also make it larger than zero, in order to model different communication devices which take significant time to encode data or to set up transmission links.

The message leaves the sender at time T+ST. In other words, it gets to the post office at T+ST, since the post office covers anything between a sender and a receiver. This message will be delivered to the destination mailbox at T+ST+DT, where DT is the message traveling time from sender to receiver.

In the current implementation, DT is calculated as the distance of sender from receiver times a constant. It is done by the function **delivery_time** which is called by **comm_msg_queue::deliver**.

7.7 What is the difference between letters and messages?

Other than "message" itself, there is a related data structure "letter" containing a pointer to a message.

The reason for having letters is to allow a message to be in multiple lists simultaneously. For instance, if message "A" is broadcast to all the platforms, a letter will be delivered to each platform's mailbox; but all of these letters will be pointing to the same message "A."

A letter is an element of the linked list associated with a mailbox in the post office; it can be accessed only by routines internal to the communication package. It is only a device to allow the same message to reside in several mailboxes simultaneously.

When a receiving platform gets a message from the mailbox, a copy will be made of message "A," and this copy will be given to the platform. The BMA always receives a private copy of the message. The BMA can do whatever it wants to with this copy; e.g., it can send this message to other platforms, or delete it after reading its contents.

This is why a reference count is kept of the number of letters which are pointing at a message at a given time. When the reference count reaches 0, the message is destroyed. The trick of keeping a reference count of the number of letters currently pointing to the same message is internal to the communication package, and is completely transparent to the BMA.

7.8 In what order are messages received?

A message is received (and possibly processed) by the receiver only when the receiver explicitly posts a get function. In this function, the receiver can only specify the channel (i.e., the mailbox) he wants to get a letter from, he cannot specify the sender or anything else.

In general, the receiving order of messages is precisely the arriving order of them, which has nothing to do with their sending order. So one message comes from a nearby platform may be received before another message sent earlier from a distant platform. However, if two messages come from the same distance away on the same channel, the message sent earlier will also be received earlier.

Letters in a mailbox are organized in their arrival time order, so the receiver simply pops the top letter in that mailbox each time.

7.9 Can the receiver sort the incoming messages?

Messages may have different "types" that can be known by the BMA.

The base class **comm_message** contains those fields that are common to all messages, e.g. time message arrived at destination, reference count, sender's ID.

A class (e.g., M1_message) is then derived from the base class for each message type; this class contains the type-dependent fields for that message type (e.g., textptr, val, and cycle for an M1_message).

The BMA programmer is required to construct a new message by declaring a variable of that message type and passing the appropriate arguments to the constructor. The communication package routines **comm_msg_queue::send** and **comm_msg_queue::broadcast** take a pointer to the base class comm_message as a parameter. The BMA can pass a pointer to a derived class, say M1_message, to these routines. Thus the communication package routines will be oblivious to the type of the message.

The only drawback to this scheme is that the BMA will have to do an explicit conversion (*cast*) of the pointer returned by the function **comm_msg_queue::get**. That is, since **get** returns a pointer to the base class comm_message, this pointer must be explicitly *cast* to a specific derived class in order to access the fields of that derived class. But, in comparison to the overall advantages of using this scheme, this small inconvenience seems to be worthwhile.

A new message type can be declared by deriving a new class from the base class comm_message.

CHAPTER 8. PHYSICAL MOVEMENT

This chapter describe software routines that "moves" objects in the simulation. The software is designed to be a simple approximation of real physical laws of motion.

8.1 Definitions

Objects assume different trajectories depending upon their types. There are three types defined in our prototype: GEOSYNC, WALKER, and MISSILE.

GEOSYNC is a synonym for a geostationary or geosynchronous orbit, one where the satellite is always over the same point on the Earth's surface.

WALKER orbits are a class of orbit where the orbital period is an integral divisor of the Earth's period of rotation (sidereal rotation is with respect to fixed stars rather than with respect to the Sun). Walker orbits have the property that the satellite passes over the same path on the Earth's surface every N orbits (it turns out that N is usually on the order of 16). Walker orbits are usually populated by M satellites, where M is a multiple of N (usually ~ N**2). The result is to give coverage of a region of the Earth's surface that is time-invariant.

WALKER orbits are designed to achieve a maximum (and constant in time) density of satellites over a given point on the Earth's surface. In a related function **traj::create_walker_orbit**, arguments convlat and convlong are the latitude and longitude of the convergence point (this is the one point through which all the satellites pass), argument nplat is the total number of platforms in the set of walker orbits, and i is the ordinal of a given platform [i = 0 .. nplat -1].

WALKER orbits consist of a number of "rings." Each ring is a circle whose origin is the center of the earth and each ring contains a point directly above the convergence point described above. The circumference of the circle is the orbital path the satellite follows. Think of it as equivalent to slices of an apple. Within a given ring the satellites follow each other at constant distance. Each ring is separate from its neighboring ring by a constant angle (if the convergence point were the North Pole, the angle separating the rings would be a longitude difference equal to 360 degrees divided by the number of rings).

To maintain constant time-average coverage, the number of rings times the period must equal the sidereal period of rotation of the earth (~23h56m). The values of n for number of rings that give reasonable solutions for low altitude orbits are in the vicinity of 16. At any rate, the value of n specifies the altitude and period of the orbit.

CHAPTER 9. INPUT, OUTPUT AND GRAPHICS

In this chapter, the input files needed by the simulator, the output files generated by the simulator and the software needed to support graphics animation of the simulation are described.

9.1 Input to the simulator

In order to run the simulation program, some input data files are needed: a capability file, a platform data file, and a threat file.

The capability file initializes the capability table. The information is read in via the function **init** -- a member function of capability_array, with the following format:

platform type
number of sensors, sensor type ...
number of communication channels, channel_number ...
number of weapons, weapon type ...

The names of BMAs will soon be included in the above format. However, at present time they are hard-coded in the function **capability_array::init**, and their names are: bma0, bma1 and bma2.

The platform data file specifies name, type and trajectory information for each platform. It is read in via the function **init_from_file** -- a member function of platform_list. Each field is identified by the platform constructor function and in turn the functions **create_traj**, **create_walker_orbit** or **create_geosync_orbit**.

In this platform data file, each line specifies one platform and has the following format:

platform name,
platform type,
trajectory type,

and the rest fields are options: if the trajectory type is geosync, then there is one more argument for the longitude of the platform; if the trajectory type is walker, then there are five more arguments: the total number of walker platforms, the ordinal of this walker in the set, the number of rings in the set, and the latitude and longitude of the convergence point of these rings.

The threat file specifies the launch time, place and destination of each attacking missile. This information is read in via the function **init_from_file** -- a member function of missile_list, with the following format:

latitude of launch point,
longitude of launch point,
latitude of target point,
longitude of target point,
launch time.

9.2 Output files

Several output files are generated by the simulator. These include an event history file, errors, communication messages log file, target table, and position file.

Two other output files, namely "db.pos" and "db.evnt," are generated by the simulator; these files can be loaded into a database, and the information contained in them can be accessed through SQL queries on the database.

The file "db.evnt" contains a listing of the events executed by the simulator, ordered by the time stamp of the event. The file "db.pos" contains a list of platform and missile positions. Each line in the file represents the position of an object at some instant in time; the positions are generated in a time-driven manner by the simulator (see the description of the event PRINT_POS). In both files, a record corresponds to a line; the fields within a record are separated by a special character (e.g., '|') to enable the files to be easily loaded into a database.

9.3 Graphics display package

Currently, **SunUNIFY**, a relational database management system, is used to store some of the data produced by the simulator. **Simplify**, a graphic, mouse oriented interface to **SunUNIFY**, is used to declare the entities, browse the database, execute **SQL** queries, and generate simple reports. **Simplify** allows these queries to be saved in files for later reuse.

Currently, two types of entities have been declared; new entities can be easily added by invoking the **Schema** option of **Simplify**. The first type of entity is EVENT; the fields in this entity are the event id, the event type, and the time_stamp of the event; the primary key is the event id. The second entity is POSITION, with fields object_type, object id, time, x, y, z (3-d position coordinates), and plot (whether to plot this point); the primary key is a composite field consisting of the object_type, object id, and the time.

Storing the data in the database allows very sophisticated post-processing of the simulator output; the full power of **SQL** can be used to generate statistics and reports. It can also be used to construct a file of positions which can serve as input to a graphics tool to display the positions (trajectories) of missiles and platforms on the screen. Each record in this file represents a point to be plotted. The format of each point/line-segment is (Z,X,Y,PLOT), where (Z,X,Y) is a position mark and PLOT is an on/off bit indicating whether there should be a line segment from the previous position mark to the current position or not.

The graphics tool we use to generate this information is the software developed by Richard Bisbey's group (see [6]); it has been tailored for our simulator by Susan Coatney. This graphics tool takes files of points to be plotted as input; files have been provided representing a grid of the earth, and a map of the continents; a file containing the positions of the platforms and missiles can also be created. The graphics package then generates a color display of the earth, the platforms, and the missiles; the display can then be repeatedly viewed from any requested angle. The graphic display has been very useful for debugging the simulator code which deals with trajectories.

9.4 Detailed directions for generating graphics

The following is a sample of the sequence of commands a user would input to execute the simulator and generate a graphical depiction of the results. The command to be executed is preceded by the characters >>; a description of the command begins on the next line.

1. >> **newsim**
Execute the simulator, specifying the times at which the simulator should begin and stop generating positions for platforms and missiles. The two files "db.pos" and "db.evnt" will be automatically generated.

2. >> **cd /tmp/mytut**
You must be logged into a Sun workstation at this point. Change directories to /tmp/mytut; this is where the database is located.

3. >> **simplify**
Execute **simplify**. Note that **simplify** must be executed from within **suntools**, and /usr/unify/bin must be in your path. From the **simplify** window, choose the button marked SQL.

4. >> **delete position**
Delete all the old records from the entity POSITION.

5. >> **start load_pos**
Execute the query to load records from the file "db.pos" into the entity POSITION. The file "load_pos" contains the query to load the file; the query specifies an absolute path for the file; this path must be updated if the file is moved (the current path is /u3/mizell/fysicium/src/main/db.pos).

6. >> **start create_pos_file**
Execute the query to access the records in POSITION and create a file which is in the format required by the graphics tool. The query is contained in the file "create_pos_file." It selects only those records with 0 <= *time* <= 30. It can easily be modified by editing "create_pos_file." The file "/tmp/mytut/posfile" will be created as a result of this query.

7. >> **cd /u3/mizell/fysicium/graphics**
Change directories to where the graphics tool is located.

8. >> **sdi_graphics grid.nd earth3d.txt /tmp/mytut/posfile**
Execute the graphics tool, passing the files to be plotted as arguments. The file "grid.nd" contains a grid of the earth; "earth3d.txt" contains a map of the continents. The graphics tool will request some input; once the requested input has been given, a color graphic display of the earth, the platforms, and the satellites will appear in a separate window. The display can then be viewed repeatedly from different points of view. If you wish to change the time frame, exit this program, modify "create_pos_file" appropriately, execute it to create a new posfile, then execute "sdi_graphics" again.

REFERENCES:

[1] Stroustrop, Bjarne, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.

[2] Cohen, Danny, "Simulation Interfaces," informal memorandum, USC/ISI, September 1987.

[3] Bisbey, Richard, "A Network Graphics System for Command and Control," *Proceedings of the Symposium on Interoperability of Automated Systems*, The Hague, Netherlands, 1982.

[4] Rose, Frank, "The Strategic Defense Initiative," *IEEE Potentials*, May 1987, pp. 15-17.

[5] Zorpette, Glenn, "Monitoring the Tests," *IEEE Spectrum*, vol. 23 no. 7, July 1986, pp. 57-66.

[6] Bisbey, Richard, Hollingworth, Dennis and Britt, Benjamin, "Graphics Language (Version 2.2)," USC/ISI technical manual ISI/TM 80-18.1, May 1984.

APPENDIX A.

SIM.DOC

On–line documentation of the main simulation routine.

This file (simulator.doc) is a document for simulator.c, version 1.13;
written by Yu-Wen Tung, 12/1/87;

PURPOSE:

File simulator.c is the main program of the SDI simulation software.

INPUT FILES:

This program requires three input files furnished by the user: capability file, platform data file and threat missile data file.

The first file, capability file, initializes the capability table. Each row in the table specifies types and numbers of the capability of a particular platform, and has the following fields in the indicated order separated by tabs or blanks, in four consecutive lines as shown below:

1. platform_type
2. number of sensors, sensor type ...
3. number of communication channels, channel_number ..
4. number of weapons, weapon type ...

where dots indicate multiple (number, type) pairs are optional.

The second file contains a description of the defensive platforms. Each platform is described by one line containing the following fields in the indicated order, separated by tabs or blanks:

1. platform_name: a unique string which identifies this platform,
2. platform_type: an integer in the range (0,1,2), corresponding to the platform types P0, P1, P2,
3. trajectory_type: an integer in the range (0,1,2,3), corresponding to the trajectory types UNDEFINED, WALKER, GEOSYNC, MISSILE (as defined by enum in traj.h),
4. trajectory_parameters: number of fields and their meanings depend on the trajectory type.

The third file contains a description of all attacking missiles. Each missile is described by one line containing the following:

1. latitude of launch point,
2. longitude of launch point,
3. latitude of target point,
4. longitude of target point,
5. launch time.

OUTPUT FILES:

The program logs the following information in several output files as well as on the screen. It first asks the user what file name he wants to use -- say "foo," and then generates output files accordingly:

1. foo.evnt: events poped from the event queue;
2. foo.msgs: communication messages;
3. foo.pos: platform positions;
4. foo.err: error messages;
5. foo.in: copy of the input platform data file;
6. foo.trg: target table;
7. db.evnt: event file in database format;
8. db.pos: position file in database format.

OPERATION PRINCIPLE:

The main data structure of this program is the event queue, and the main operation on the event queue is the get-eval-put cycle. The initial events are placed one for each platform created. The program gets one event at a time according to the time stamp of the events, evaluates it according to its type, then consumes it and puts zero or more event in the event queue according to the event content. The program terminates when the queue is empty.

The program does the following:

1. initial platform capabilities and trajectories;
2. initialize event queue, communication server and output files;
3. main simulation cycle:

```
while event-queue not empty:
{pop one event from the event queue;
print out the event to .evnt file;
evaluate the event according to its type
(BMA/SENSE/WPN/COMM);
remove the event}
```

RELATED FUNCTIONS:

1. capability_table.init():
   reads from capability file and initialize capability table. The variable capability_table is a global variable of type capability_array, see file objects.c.

2. table_of_instance.init_from_file:
   reads from platform data file and creates a table of platform instances. The variable table_of_instance is a global variable of type platform_list, see file objects.c.

3. table_of_instances.size():
   returns the number of platform instances. It is a member function of class platform_list, see objects.h.

4. missile_table.init_from_file():
   reads from threat missile data file and creates trajectories of missiles. It is a member function of missile_list, see file missile.c.

5. *init_event_queue()*:
   initializes event queue. It cycles through the list of platforms (table_of_instances) and posts an event for each platform's BMA with time_stamp 0. It is an external function for class event, see file event.c.

6. *post_office.init()*:
   initializes communication server. The variable post_office is a global variable of type comm_msg_queue, and the function init() is a member function of class comm_msg_queue, see file comm.c.

7. *get_next_event()*:
   simply pops the first event from the event queue which is organized by time stamps of the events when they are inserted. The popped event is called current_event and is then evaluated according to its type.

EOF

APPENDIX B.

SIM.C

Source code of the main routine.

```
/* @(#)simulator.c        1.13      11/24/87
   - simulator.c is the main program of the SDI simulation software.
   - This program requires 2 input files: one which contains a description
     of the defensive platforms, & one for the missiles.
   - The file buffers for messages, errors, events, positions and targets
     are initialized, & events are written to the output file.
   - The only event type currently used is BMA; SENSE and WPN type
     events have not been implemented yet. (OLD)
   - Implemented events: BMA, SENSE, LAUNCH, SAMPLE (interval sensing), SHOOT,
         DESTROY (a missile), PNUKE (possible ground hit), NUKE (ground hit).
   - See simulator.doc for implementation details.

   written May, 29 by Rivi.
   update 6/17/87 by slc: removed COMM type event
   update 6/23/87 by slc: changed all char[] to string
   update 6/24/87 by slc: changed post office init routine
   update 6/25/87 by slc: changed include files to use z_header.h files
   update 6/30/87 by Walid: added output
   update 7/13/87 by slc: added comments
   update 8/6/87 by susan and rivi : changes because of event class
   update 8/6/87 by walid for SHOOT, DESTROY, PNUKE and NUKE events
   update 8/11/87 by rivi and susan : change sense events , destroy to pdestroy
            delete nuke event
   update 9/1/87 by susan: added capability table
   update 9/2/87 by susan: added call to initialize freestore package
   update 9/3/87 by susan: added BMA_SENSE,BMA_COMM, BMA_DELAY, BMA_INIT events
   update 9/18/87 by susan: added event print for database
   update 10/12/87 by susan: added event PRINT_POS to print positions
         of platforms and missiles to a file for the graphics package.
         Added request for user to input time frame in which to print
         positions.
*/

#include <stream.h>
#include "utilities.h"
#include "comm_header.h"
#include "event_header.h"
#include "sensor_header.h"
#include "missile_header.h"
#include "target_header.h"
#include "weapon_header.h"
#include "freestore.h"

// functions used only by simulator.c
ostream& operator>>(ostream& from, stime& var);
static void write_logdata(ostream& to,string input_file_name,
                string threat_file_name, string cap_file_name);
static void output_init(string &);
static void print_objects(ostream& to);

filebuf         dbevnt; // used only in simulator.c
filebuf         dbpos;  // used only in simulator.c
```

main

```
main()
{
        //cout << "enter main\n";

        // initialize freestore package
        init_freestore();

        // used to point to popped event
        event*      current_event;

        // current label with which BMA is called
        int         label;

        // id of current platform instance in case of a BMA event type
        platform_id pid;

        // id of current missile instance for missile events
        missile_id mid;

        // pointer to the descriptor of current platform instance
        // in case of a BMA event type
        platform*   ptr_instance;

        // pointer to persistent variables of platform instance
        pvar_ptr    ptr_persistent;
```

```
// pointer to the bma function to invoke
bma_ptr      bma_fun;

// name of platform file to read
string          input_file_name;
cout << "name of platform file = ";
cin  >> input_file_name;

// name of threat file to read
string              threat_file_name;
cout << "name of threat file = ";
cin >> threat_file_name;

// name of capability file to read
string              cap_file_name;
cout << "name of capability file = ";
cin >> cap_file_name;

// get time frame for printing object positions
stime start_print, stop_print, delta_print;
cout << "start generating positions at time: ";
cin >> start_print;
cout << "stop generating positions at time: ";
cin >> stop_print;
cout << "interval: ";
cin >> delta_print;
cout << "start: " << start_print << " stop: " << stop_print
        << " delta: " << delta_print << "\n";

// initialise global output file buffers
output_init(input_file_name);

// bind global output file buffers to output streams
ostream             dbevntout(&dbevnt);
ostream             dbposout(&dbpos);
ostream             evntout(&fbevnt);
ostream             posout(&fbpos);
ostream             msgout(&fbmsg);
ostream             trgout(&fbtrg);

// write log data on output files and stdout
write_logdata(cout,input_file_name,threat_file_name,cap_file_name);
write_logdata(evntout,input_file_name,threat_file_name,cap_file_name);
write_logdata(posout,input_file_name,threat_file_name,cap_file_name);
write_logdata(msgout,input_file_name,threat_file_name,cap_file_name);
write_logdata(trgout,input_file_name,threat_file_name,cap_file_name);

// read from capability file & initialise capability table
// capability_table: global variable of type capability_array
// see objects.c
// NOTE: capability_table MUST be initialised BEFORE table_of_instances
capability_table.init(cap_file_name);

// read from platform file & create a table of platform instances
// table_of_instances: global variable of type platform_list
// see objects.c
table_of_instances.init_from_file (input_file_name);

// LHS: global variable; see global.c
// RHS: member function of class platform_list; see objects.h
num_instances = table_of_instances.size();

//read the threat file
missile_table.init_from_file (threat_file_name);

// initialise event queue
// external function for class event; see event.c
init_event_queue (table_of_instances,start_print);

// initialise communication server
// post_office: global variable of type comm_msg_queue;see comm.c
post_office.init(table_of_instances.size());

//cout << "begin simulation...\n";

// pop events
while (current_event = get_next_event()) {
```

```
// update global variable with current simulation time
sim_time = current_event->get_event_time();

//increment event counter
current_event->inc_evnt_counter();


// pointer for target
target* curtarget;

// always print event to output file
evntout << *current_event;

// print current event to event database file
current_event->dbprint(dbevntout);

 event* tmpevt;

 // time for next print of object positions
 stime next_print;

// evaluate event according to type
switch (current_event->get_event_type()) {

  case PRINT_POS:
                // print positions of all platforms & missiles
                print_objects(dbposout);

                // if event queue is not empty
                // schedule another PRINT_POS event
                //if (! tmpevt->empty_queue()){

                // if not time to stop printing,
                // schedule another PRINT_POS event
                if ( sim_time <= stop_print ){
                        next_print = sim_time + delta_print;
                        tmpevt = new event(next_print,
                                           PRINT_POS);
                        tmpevt->insert_event();
                }
                break;
  case BMA_SENSE:
  case BMA_COMM:
  case BMA_DELAY:
  case BMA_INIT:
                pid = current_event->get_platform_id();
                label = current_event->get_label();
                ptr_instance = table_of_instances[pid];
                ptr_persistent = ptr_instance->get_pvar_ptr();
                bma_fun = ptr_instance->get_bma();

                // call bma code
                (*bma_fun) (label , pid , ptr_persistent);
                break;
  case SENSE_APPEAR :
  case SENSE_DISAPPEAR :
  case SENSE_BLOWNUP :
                pid = current_event->get_platform_id();
                curtarget = current_event->get_target_ptr();
                accumulate_sense(curtarget, pid);
                break;

  case (LAUNCH) : mid = current_event->get_missile_id();
                execute_launch_event(mid);
                break;

  case (SAMPLE) : pid =
                current_event->get_platform_id();
                sample(pid);
                break;

  case (SHOOT) : curtarget = current_event->get_target_ptr();
                pid = current_event->get_platform_id();
                execute_shooting(curtarget, pid);
                break;

  case (PDESTROY) : mid = current_event->get_missile_id();
                boolean resultpd = destroy_missile(mid);
                if (resultpd)
```

```
                                    evntout << "SUCCESS" << "\n";
                                else
                                    evntout << "FAILED" << "\n";
                                break;

                case (PNUKE) : mid = current_event->get_missile_id();
                               boolean resultpn = execute_pnuke(mid);
                                if (resultpn)
                                 {
                                    evntout << "SUCCESS" << "\n";
                                cout << "WE SHOULD HAVE GONE TO NEW-ZEALAND"
                                    << "\n"; }
                                else
                                    evntout << "FAILED" << "\n";
                               break;

                default        : sim_error("main",E_ILEVENT);
                               break;
                }
                // free event
               delete current_event;
        }
        cout << "end simulation\n";
}

/* initialize output file buffers for events, positions, messages,
 * and errors.
 * written by slc 7-1-87
 * modified by Walid 7-2-87
*/
static void output_init(string& input)
{
        extern char *system(char *);

        // read name of output files
        string    output_filename;
        cout << "output file name = ";
        cin >> output_filename;

        // open ".err" file
        string errfile = output_filename + ".err";
        if (fberr.open(errfile.readstr(),output) == 0){
                cout<<form("ERROR - cannot open error file (simulator.main)\n");
                exit(1);
        }

        // open ".evnt" file
        string filename = output_filename + ".evnt";
        if (fbevnt.open(filename.readstr(),output) == 0)
                sim_error("simulator.main [evnt]",O_FILEWT);

        // open "db.evnt" file
        filename = "db.evnt";
        if (dbevnt.open(filename.readstr(),output) == 0)
                sim_error("simulator.main [db.evnt]",O_FILEWT);

        // open "db.pos" file
        filename = "db.pos";
        if (dbpos.open(filename.readstr(),output) == 0)
                sim_error("simulator.main [db.pos]",O_FILEWT);

        // open ".pos" file
        string posfile = output_filename + ".pos";
        if (fbpos.open(posfile.readstr(),output) == 0)
                sim_error("simulator.main [pos]",O_FILEWT);

        // open ".msgs" file
        string msgsfile = output_filename + ".msgs";
        if (fbmsg.open(msgsfile.readstr(),output) == 0)
                sim_error("simulator.main [msgs]",O_FILEWT);

        // copy platform file to ".in" file
        string copy = "cp " + input + " " + output_filename + ".in";
        if (system(copy.readstr()) != 0)
                sim_error("simulator.main[in]",O_FILEWT);

        // open ".trg" file
        string trgfile = output_filename + ".trg";
        if (fbtrg.open(trgfile.readstr(),output) == 0)
```

```
                        sim_error("simulator.main [trg]",O_FILEWT);
        }

        /* Write log data on an output stream: log data consists of
         * user name, date, name of platform description file, and
         * name of missile description file.
         * written by slc 7-1-87
         */
        static void write_logdata(ostream& to,string input_file_name,
                        string threat_file_name,string cap_file_name)
        {
                extern char *getenv(char *);
                extern long time(long *tloc);
                extern char *ctime(long *);
                char* us = "USER";

                long t = time((long*)0);
                char* date = ctime(&t);
                if (to == cout){  // output to stdout
                        to << "\nUSER = " << getenv(us) << "\n";
                        to << "DATE = " << date;
                        to << "PLATFORM FILE = " << input_file_name;
                        to << "THREAT FILE = " << threat_file_name;
                        to << "CAPABILITY FILE = " << cap_file_name;
                }else{          // output to file
                        to << getenv(us) << "\n";
                        to << date;
                        to << input_file_name;
                        to << threat_file_name;
                        to << cap_file_name;
                }
        }

        // Print positions of all platforms & missiles which have not
        // been destroyed. Use database format. Form of output:
        //      object_type | id | x | y | z | plot
        static void print_objects(ostream& to)
        {
                position        tmp_pos;
                static int      first = 1;

                // print platform positions
                plist_iterator  next(table_of_instances);
                platform*       plat;
                while (plat = next()){
                        to << "P | " << plat->get_id() << " | ";
                        plat->get_traj()->eval(&tmp_pos);
                        tmp_pos.dbprint(to);

                        // do not plot initial position of platforms
                        if (first)      to << " | 0\n";
                        else            to << " | 1\n";
                }

                // print missile positions
                missile_list_iterator   nextmiss(missile_table);
                missile*                miss;
                while (miss = nextmiss())
                        if (miss->get_state() != DESTROYED){
                                to << "M | " << miss->get_id() << " | ";
                                miss->get_traj()->eval(&tmp_pos);
                                tmp_pos.dbprint(to);

                                // do not plot initial position of missiles
                                if (first)      to << " | 0\n";
                                else            to << " | 1\n";
                        }
                first = 0;
        }

        // This should go in global.c
        // overloaded stream operator for type stime
        ostream& operator>>(ostream& from, stime& var)
        {
                float tmp;
                from >> tmp; var = (stime) tmp;
                return from;
        }
```

APPENDIX C.

BMA.DOC

On–line documentation of the
battle manager abstractions (BMAs)
used to debug the initial prototype.

```
--------------------------------------------------------------------
This file (bma.doc) is a document for bma.c, version 1.3;
                                         written by Yu-Wen, 12/3/87;
--------------------------------------------------------------------

PURPOSE:
          Battle management abstraction, or BMA, is a software program
          that specifies behaviors of defense platforms.

          It is possible for two platforms to use the same piece of BMA
          code, but each platform must have its own copy.

          The BMAs collected in file bma.c are only examples.  The real
          BMA package is to be done by the designer of defense architect
          to be simulated.

          In this example, three different types of BMA are provided:
          BMA0, BMA1 and BMA2, each is written as a function: bma0(),
          bma1(), bma2().

BMA0:
          This BMA code is designed for a headquarters platform that receives
          target data from geosync platforms and sends a "weapons release"
          message back.  This decision-making headquarters platform has
          only communication capabilities.

          bma0(): is called with the following arguments:
          1. label: either LABEL0 or LABEL1:
                     : if is LABEL0, it will initialize communication channel 0,
                     : if is LABEL1, it will get a message at channel 0, if the
                       the message says "target appear," send a "weapons release"
                       message back to the source platform (to LABEL1 of its
                       BMA code),
          2. platform_id: has only one ID in this case;
          3. pvar_ptr: pointer to the set of persistent variables to be
                     passed to the BMA code.  There is none persistent variable
                     in BMA0.

BMA1:
          This BMA code is designed for a geosync platform that has sensor S1
          and communication channel CHANNEL0.  The function of this platform
          is:
          1. if targets appear in peace, then send a "target appear" message
             to head-quarter,
          2. if targets appear, disappear or are destroyed during a battle,
             then broadcast target list to carrier vehicles,
          3. if a "weapons release" message is received from the headquarters,
             then broadcast target list to carrier vehicles;

          bma1(): is called with the following arguments:
          1. label: LABEL0, LABEL1 or LABEL2:
                     : if is LABEL0, it will initialize communication channel 0
```

and sensor S1, so that an incoming sense event will call LABEL2. Set state to peace,

: if is LABEL1, it will get a message at channel 0, if the the message says "weapons release," broadcast a list of targets to all carrier vehicles (to the LABEL2 of their BMA codes), change the state to battle,

: if is LABEL2, then get target list. If it is in peace, create a "target appear" message and send it to the headquarters, and save the target list in the persistent variables, otherwise create a "new targets" message and send it to all carrier vehicles.

2. platform_id: has only one ID in this case;
3. pvar_ptr: pointer to a set of persistent variables which includes:
   (1). peace: 1 represents peace and other integers represent battle,
   (2). list of targets.

BMA2:

This BMA code is designed for 200 CVs (carrier vehicles); each has some laser weapons and a communication channel CHANNEL0. Each CV functions as described below:
If receive target data from geosync sensor,
then update target list, and shoot targets while shootable targets left and shots available.

bma2(): is called with the following arguments:
1. *label: either LABEL0 or LABEL1:*

   : if is LABEL0, it will initialize communication channel 0 and persistent variables;

   : if is LABEL1, it will get a message at channel 0, if the the message says "new targets," update platform list of targets, initialize list of left targets, pick list of targets and shoot while shootable targets exist and there are available shots left.
2. platform_id: one of the 200 CVs.
3. pvar_ptr: pointer to a set of persistent variables which includes:
   (1). list of targets,
   (2). number of shots.

APPENDIX D.

BMA.C

Source code of the battle manager abstractions (BMAs).

```
/* @(#)bma.c      1.3       8/4/87
   This is the source file of BMAs
   Each BMA is preceded by a description of its corresponding
   platform type, and a description of its persistent variables.

   Every BMA is passed 3 arguments:
        int              label;           label at which to begin
                                          execution of the BMA
        platform_id   instance_id;        id of the platform instance
                                          for which the BMA is being invoked
        pvar_ptr*       pvars;            pointer to the instance's persistent variables

   modified 6/10/87 by slc
   update 6/23/87 by slc: changed include files
   update 6/24/87 by slc: changed communication package interface
   update 6/29/87 by sjc: added call to print_pos to test trajectory
   update 7/1 /87 by rivi: change bma0 to check trajectory+communication
   update 6/29/87 by slc: changed communication package interface
   update 8/11/87 by rivi : a "real" bma
   update 11/4/87 by sjc : try to find out why bma isn't shooting
*/

#include        <stream.h>
#include        "bma_interface.h"

enum label_type { LABEL0, LABEL1, LABEL2, LABEL3 };

/*************** BMA0 ***********************
   BMA for Platform HQ0
   --------------------

   receive target data from GEO platform
                        -> send "weapon release" message  to GEO

   Platform type : P0

   Capabilities :
        Communications:  CHANNEL0

   Persistent variables:
        NONE
*/
void     bma0(int label, platform_id instance_id, pvar_ptr pv)
{
        /* output streams for positions and messages and targets */
         ostream  posfile(&fbpos) , msgfile(&fbmsg) , trgout(&fbtrg);

        /* pointer to persistent variables */
        // pvar0_ptr       pvars = (pvar0_ptr) pv;

        const target_list* null_ptr = 0;
        switch (label){

        case LABEL0: /* initialisation */
                /* initialise communication at channel0 */
                set_msg_flag (instance_id , CHANNEL0 , INTR_ON , LABEL1);
                break;

        case LABEL1: /* for input messages interrupts */
                /* get the message */
                comm_message* msg = get (instance_id , CHANNEL0);
                M2_message*   target_appear_msg = (M2_message*) (msg);
                target_appear_msg->print(msgfile);
                msgfile.flush();
                /* if this is the right message */
                if (*(target_appear_msg->textptr) == "target appear") {
                        /* create a "weapon release" message */
                        M2_message* weapon_release_msg = new M2_message
                                ("weapon release", null_ptr);
                        /* send the message to GEO platform */
                        platform_id GEO_id = get_platform_id ("GEO0");
                        send_msg (weapon_release_msg , GEO_id , CHANNEL0 ,
                                instance_id ,INTR_OFF , LABEL1);
                        /* free memory */
                        delete target_appear_msg;
                        }
                else  { /* this is not the right message */
                        /* ERROR */
                }
            break;

        case LABEL2:
            break;

        case LABEL3:
            break;

        default: sim_error("bma0",E_ILLABEL);
        }
}

/*************** BMA1 ***********************
   BMA for Platform GEO0
   ---------------------

   targets appear  in peace
                 -> send "target appear" message to HQ
   targets appear/disappear/blowup in battle
                 -> broadcast target list to CVs
   receive "weapon release" message from HQ
                 -> broadcast target list to CVs

   Platform type : P1

   Capabilities :
      Sensors:            S1
      Communications:     CHANNEL0

   Persistent variables:
      int              peace    (if 1 then peace else battle);
      target_list* list_of_targets;
*/
void    bma1 (int label, platform_id instance_id, pvar_ptr pv)
{
        /* output streams for positions and messages and targets */
        ostream  posfile(&fbpos) , msgfile(&fbmsg) , trgout(&fbtrg);

        /* pointer to persistent variables */
        pvar1_ptr          pvars = (pvar1_ptr) pv;

        const target_list* null_ptr = 0;
        switch (label){

        case LABEL0: /* initialisation */
                /* initialise communication at channel0 */
                set_msg_flag (instance_id , CHANNEL0 , INTR_ON , LABEL1);
                /* initialise staring sensor */
                init_staring_sensor (instance_id , 0.3 , LABEL2);
                /* initialise persistance variables */
                pvars->peace = 1;
                break;

        case LABEL1: /* for input messages interrupts */
                /* get the message  */
                comm_message* msg = get (instance_id , CHANNEL0);
                M2_message* weapon_release_msg = (M2_message*) (msg);
                weapon_release_msg->print(msgfile);
                msgfile.flush();
              /* if this is the right message */
                if ((*weapon_release_msg->textptr)=="weapon release") {
                        /* create a "new targets" message */
                        trgout << "platform id " << instance_id << "\n";
                        trgout << *pvars->list_of_targets;
                        M2_message* target_msg = new M2_message
                                ("new targets" , pvars->list_of_targets);
                        /* broadcast the  message to CV's */
                        broadcast_msg (target_msg , CHANNEL0 , instance_id ,
                                INTR_OFF , LABEL2 , P2);
                        /* change state to battle */
                        pvars->peace = 0;
                        }
                else { /* if this is not the right message */
                        /* ERROR */
                        }
                /* free memory */
                delete weapon_release_msg ;
                break;
        case LABEL2: /* for sensor interrupts */
                /* get target list */
                target_list* tglist = get_target (instance_id);
                trgout << "platform id " << instance_id << "\n";
                trgout << *tglist;

                /* if in peace */
                if (pvars->peace) {
                        /* create a "target appear" message */
                        M2_message* target_appear_msg = new M2_message
                                        ("target appear",null_ptr);
                        /* send to HQ platform */
                        platform_id HQ_id = get_platform_id ("HQ0");
                        send_msg (target_appear_msg , HQ_id , CHANNEL0 ,
                                        instance_id , INTR_OFF , LABEL2);
                        /* save the target list in the persistance variables */
                        pvars->list_of_targets = tglist;
                        }
                else {/* in battle */
                        /* create a new targets message */
                        M2_message* target_msg = new M2_message
                                        ("new targets" , tglist);
                        /* broadcast the message to CV's */
                        broadcast_msg (target_msg , CHANNEL0 ,
                                        instance_id , INTR_OFF , LABEL2 , P2);
                }
                break;
        case LABEL3:
                break;

        default: sim_error("bma1",E_ILLABEL);
        }
}


/*************** BMA2 ***********************
   BMA for Platform CV0 - CV100
   ----------------------------

   receive GEO sensor data
             -> { update target list ;
                  while (shootable targets left and shots left) do
                            { pick target ;
                              shoot;
                              update number of shots left;
                            }
                }

   Platform type : P2

   Capabilities :
        Weapon:               KKV
        Communications:       CHANNEL0

   Persistent variables:
        target_list*          list_of_targets
        int                   no_of_shots
*/

/* update current_list of targets according to update_list */
void  update_target_list (target_list* update_list ,
                          target_list* current_list)

/*      if target_state==INVISTOVIS
             add target to current_target_list
        if target_state==VISTOINVIS or VISTOBLOWN
             find target and delete from current_target_list
*/
{
        target* update_tpnt = update_list->head() ;
        target* tmp_update_tpnt;
        /* iterate on update_list */
        while (update_tpnt != NULL) {
                tmp_update_tpnt = update_tpnt->next();
                if (update_tpnt->get_visibility()==INVISTOVIS) {
                    /* remove the target from update list so it
                       will not be destrpyed when update_list is deleted */
                    update_list->remove (update_tpnt);
```

```
                        /* add appearing target to current list */
                        current_list->append (update_tpnt);
                        }
                    else {
                        /* delete all disappearing and blownup targets */
                        missile_id mid = update_tpnt->get_missile_id();
                        target* current_tpnt = current_list->head();
                        int found = 0;
                        /* search this target in current_list */
                        while ((current_tpnt != NULL) && !found )
                           if (current_tpnt->get_missile_id()==mid) {
                              /* when found delete : remove and destroy */
                              current_list->delete_item (current_tpnt);
                              found=1;
                              }
                           else
                              current_tpnt = current_tpnt->next();
                      /* if corresponding appearance target not found then error*/
                        if (!found)
                           cout << "error : disappear and no appear !";
                        }
                  update_tpnt = tmp_update_tpnt;
                  }
}

struct int_list {
                  int        exist;
                  int_list* next;
                  } ;

/* initialise targets_left list */
int_list*  init_targets_left (int num)
{
        int_list* last_item = 0;
        for (int i=0 ; i<num ; i++) {
             int_list* item = new int_list;
             item->exist = 1;
             item->next  = last_item;
             last_item = item;
             }
         return (last_item);
}

/* free targets_left list */
void  free_targets_left (int_list* targets_left)
{
        int_list* p = targets_left;
        int_list* tmp_p;
        while (p != NULL) {
               tmp_p = p->next;
               delete p;
               p = tmp_p;
               }
}

/*  pick a target from the target list */
int pick_target (target_list* current_list ,
                  int_list*     targets_left ,
                  platform_id  instance_id)
{
  /* pick targets using pknow(target* , instance_id)
     put exist=2 in targets_left for those targets which should be shot now */
  target_list_iterator next (current_list);
  target*     current_target;
  int_list*  max_prob_p = 0;
  float       max_hit_prob = 0.0;
  int_list*  p_left = targets_left;

  /* iterate on list of targets
     find the target with the maximal probability to be hitted by this
     platform */
  while (current_target = next()) {
          float hit_prob = pknow (current_target , instance_id);
//        cout << "hit prob " << hit_prob << "\n";
          if ((hit_prob > max_hit_prob) && (p_left->exist)) {
               max_hit_prob = hit_prob;
               max_prob_p = p_left;
               }
          p_left = p_left->next;
```

```
            }

    /* if the probability to hit > 0.
       put 2 in the max_prob_p pointer   */
    if (max_hit_prob > 0.) {
        max_prob_p->exist = 2;
        return (1);
        }
    else
        return (0);
}

void    bma2(int label, platform_id instance_id, pvar_ptr pv)
{
        /* output streams for positions and messages and targets*/
        ostream  posfile(&fbpos) , msgfile(&fbmsg) , trgout(&fbtrg);

        /* pointer to persistent variables */
        pvar2_ptr        pvars = (pvar2_ptr) pv;
        const int NUM_SHOTS = 10;

        switch (label){

        case LABEL0: /* initialization */
                /* initialize communication */
                set_msg_flag (instance_id , CHANNEL0 , INTR_ON , LABEL1);
                /* initialize persistance variables */
                pvars->no_of_shots = NUM_SHOTS;
                pvars->list_of_targets = new target_list;
                break;

        case LABEL1: /* for input messages interrupts */
                /* get the message */
                comm_message* msg = get (instance_id , CHANNEL0);
                M2_message* target_msg = (M2_message*) (msg);
                target_msg->print(msgfile);
                msgfile.flush();
                /* if this is the right message */
                if (*(target_msg->textptr)=="new targets") {
                        /* update platform list of targets */
                        update_target_list (target_msg->tlist_ptr ,
                                            pvars->list_of_targets);
                        /* free memory */
                        delete target_msg->tlist_ptr;
                        delete target_msg;
                        /* initialize list of left targets  */
                        int_list* targets_left =
                            init_targets_left (pvars->list_of_targets->size());
                        /* pick list of targets */
                        int shootable_targets ;
                        /* while shootable targets exist and shots left */
                        while ((pvars->no_of_shots > 0 ) &&
                               (shootable_targets =
                                   pick_target (pvars->list_of_targets ,
                                         targets_left, instance_id))) {

//                             cout << "shootable targets for platform " << instance_id << "\n";
                              /* iterate on target_list */
                               target* current_target;
                               target_list_iterator next(pvars->list_of_targets);
                               int_list* p = targets_left;
                               while (current_target = next()) {
                                     /* if this targets signed to shoot at */
                                     if (p->exist==2) {
                                        /* shoot this target */
//                                      cout << "shooting at " << *current_target << "\n";
                                        shoot_at (current_target , instance_id);
                                        /* update number of shots */
                                        pvars->no_of_shots--;
                                        /* update targets_left : target was shot */
                                        p->exist = 0;
                                        }
                                     p = p->next;
                                     } /* end of target_list iterator */
                               }
                        /* no more shootable targets or no more shots */
                        /* free targets_left memory */
                        free_targets_left (targets_left);
                        }
        else {
                /* ERROR */
            }
        break;

case LABEL2:
        break;
case LABELS:
        break;

default: sim_error("bma2",E_ILLABEL);
}
```