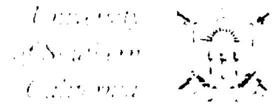
DTIC FILE COPY

ISI Research Report
ISI/RR-89-224
July 1989

(4)

AD-A211 572



David Mizell
Scott Carter

ISI's SDI Architecture Simulator: The "KMAC" Battle Manager Specification Language

DTIC
ELECTE
AUG 18 1989
S D
D

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

INFORMATION SCIENCES INSTITUTE

213/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

89 8 18 096

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified

1b. RESTRICTIVE MARKINGS:

2a. SECURITY CLASSIFICATION AUTHORITY:

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE:

3. DISTRIBUTION / AVAILABILITY OF REPORT: This document is approved for public release; distribution is unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S): ISI/RR-89-224

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: Office of Naval Research

6c. ADDRESS (City, State, and ZIP Code):
4676 Admiralty Way
Marina del Rey, CA 90292--6695

7b. ADDRESS (City, State, and ZIP Code):
800 N. Quincy Street
Arlington, VA 22217

8a. NAME OF FUNDING / SPONSORING ORGANIZATION: SDIO

8b. OFFICE SYMBOL (If applicable):

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N00014-87-K-0022

8c. ADDRESS (City, State, and ZIP Code):
Strategic Defense Initiative Organization
Office of the Secretary of Defense
The Pentagon, Washington, DC 20301

10. SOURCE OF FUNDING NUMBERS

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

11. TITLE (Include Security Classification): (Unclassified)
ISI's SDI Architecture Simulator: The "KMAC" Battle Manager Specification Language

12. PERSONAL AUTHOR(S): Mizell, David; Carter, Scott

13a. TYPE OF REPORT: Research Report

13b. TIME COVERED: FROM ______ TO ______

14. DATE OF REPORT (Year, Month, Day): 1989, July

15. PAGE COUNT: 23

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 programming, SDI architecture, software engineering

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

This report describes "KMAC," the production-rule-based high-level language for specifying abstract representations of battle management computations for execution within ISI's SDI architecture.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION: Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL: Victor Brown Sheila Coyazo

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

ISI Research Report
ISI/RR-89-224
July 1989

David Mizell
Scott Carter

ISI's SDI Architecture Simulator: The "KMAC" Battle Manager Specification Language

Accesion For		
NTIS	CRA&I	✓
DTIC	TAB	☐
Unannounced		☐
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

INFORMATION SCIENCES INSTITUTE

213/822-1511

4676 Admiralty Way/Marina del Rey/California 90292-6695

This research is supported by the Strategic Defense Initiative Office under Office of Naval Research Contract No. N00014-87-K-0022. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of the Strategic Defense Initiative Office, the Office of Naval Research, or the U. S. Government. No official Government endorsement should be inferred.

1. Introduction

In 1987, ISI's parallel and distributed computing research group implemented a prototype sequential simulation system, designed for high-level simulation of candidate SDI architectures. The design approach and the resulting software system are documented in [1]. One of our main design goals was to produce a simulation system that could incorporate non-trivial, executable representations of battle management computations on each platform that were capable of controlling the actions of that platform throughout the simulation. We used the term BMA (battle manager abstraction) to refer to these simulated battle management computations.

In our first version of the simulator, the BMAs were C++ programs that we wrote and manually inserted into the system. Since then, we have designed and implemented "kmac," a high-level language for writing BMAs. The kmac preprocessor, built using the Unix tools *lex* [2] and *yacc* [3], translates kmac source programs into C++ programs and passes them on to the C++ compiler. The kmac preprocessor has been incorporated into and operates under the control of the simulator's interactive user interface, documented in [4]. After the kmac preprocessor has translated a program into C++, the user interface system invokes the C++ compiler, and incorporates the resulting object code into the simulator load module for execution as part of a simulation run.

This report describes the kmac language and its preprocessor. Section 2 provides background material on the design of the simulation system that is necessary for understanding some of the parts of kmac and some of the reasons it is structured the way it is. Section 3 describes the syntax and semantics of the language, and Section 4 discusses design of the preprocessor. A complete example of the kmac sources for two BMAs that make up a simple, laser-based defense architecture is provided in the Appendix.

2. Influences of the Simulation System on the Design of the Language

The kmac language is based on production rules. The left-hand side of each rule is the name of an event that has occurred in the simulation system and caused the BMA to be activated. The right-hand side of each rule is a set of one or more C++ statements that the BMA will execute upon activation by the event named on the left-hand side. A set of rules can be associated with a state value. This allows the BMA to respond differently to a type of event, depending on what state it is in.

The first author originally saw examples of the use of this sort of rule-based notation to abstractly describe battle management computations in a presentation by F. Zussman in 1985 [5]. We have found this notation to be a fairly natural way to abstractly describe the computations and decision making being performed by both computer and human components of a simulated battle management/C3 system. Use of the production rule-based

notation was also influenced, however, by the nature of our simulator and the interface it provides between the simulated battle managers on each platform and our model of the external environment.

2.1 Technology Module Interfaces

Our intention was to design a simulation system that could incorporate non-trivial executable models of battle management computation on each platform. We wanted this executable code to be able to receive information about the simulated environment and then react to it in ways capable of affecting the environment. For example, we wanted it to be possible for the BMA of a simulated weapon platform to discover, via simulated sensor input, that a new target has come within range of its weapons. By deciding to launch a weapon, perhaps destroying the target, the BMA would be reacting in a way that would change the state of the rest of the simulation.

The *technology modules* of our simulation system design make up the software interface between BMAs and the external environment. Technology modules are software objects that contain all code necessary to model some defense system component that could be attached to a defense platform (see [1]). They include the simulation models of all sensors, weapons, communication devices, and propulsion capabilities. BMAs are activated and receive information from the external environment when a technology module causes an activation event for a BMA. Symmetrically, BMAs are able to affect the external environment by making procedure calls to a technology module. This is reflected in the kmac language in the set of events whose names appear on the left-hand sides of rules in a kmac program and the set of function calls that can appear within the C++ code on the right-hand sides. Both are determined by the set of components that are attached to the platform controlled by that BMA. These, in turn, are specified by the user when he chooses the configurations of each platform type.

2.2 The Simulator's Event Queue and Scheduler

The primary way in which the simulation system design influences the structure of the BMA specification language is that the language had to be designed to work with a discrete event simulator. That is, there is an implicit assumption that the BMAs cannot run continuously in this simulation system; they must be activated to respond to events of concern to their platform and then cease activity once having responded. Each kmac BMA specification is translated into a C++ program, which is then incorporated into the simulator. There is an entry point into that procedure for each of the events named by left-hand sides of production rules in the kmac program. An activation event causes a BMA to be scheduled by the simulator. It executes the right-hand side of the production rule and then exits, returning control to the simulator. In our implementation of the simulation

system, we assume the execution of a BMA to take zero time on the simulation "clock." We provide a delay function that users can call if they wish to represent in the simulation the amount of time it takes for some significant battle management computation to complete.

2.3 Persistent Variables

"Continuity" between activation events is approximated in our simulator by the BMA's state. This consists of a set of kmac program variables, specified by the user, which we refer to as *persistent variables*, or "pvars". Persistent variables are guaranteed in our implementation to retain, upon the activation of their BMA, the value last stored in them in a previous activation of the BMA.

3. Kmac Syntax and Semantics

Figure 1 illustrates the structure of a kmac program. The "functions" section can contain C++ source code that defines (or "#include" statements that reference) user-defined C++ procedures that this BMA will use. The "pvars" section holds C++-style variable declarations. Declaration of a variable within this section of the kmac program causes the variable to be treated as persistent, i.e., the simulation system will guarantee that these variables will retain their contents between BMA activations. The "locals" section provides the user the option of having a central place for declaring dynamic variables -- those which will be re-allocated upon each BMA activation. The user's other choice is to declare these temporary variables within the right-hand side of a kmac rule, but, because of the way the preprocessor translates kmac into C++, the syntactic scope of a variable so declared would be limited to that right-hand side.

These declaration sections are followed by the production rules. A subset of the rules can be associated with a state name. It precedes the set of rules associated with it and is separated from them by a colon. The semantics is that, if a kmac program is in "state1" and "event1" occurs, then the production rule grouped with state1 will be activated, but if the program is in, say, "state2" when "event1" occurs, the event1 rule in that group will be activated. As is shown in Figure 1, the programmer can set a new program state in reponse to an event, by making an assignment of one of these state names to a special variable named STATE. Procedure calls to technology module operations also occur on the right-hand-sides of the rules.

4. Implementation of the Language Translator

The kmac preprocessor consists of three lexical scanners, each produced using *lex*, under the control of an LR(1) parser, produced using *yacc*. Usually when one designs a language processor, only one lexical scanner is required. Three were used for kmac because this

```
FUNCTIONS [[
        function definitions
                ]]

PVARS  [[
                int i;
                float x[100];
                etc.
        ]]

LOCALS [[
                int ktemp;
                float ytemp[30];
                etc.
        ]]

state1:

event1 >>> { ---- c-code -----; delay(n); }

event2 >>> { ---- c-code -----; STATE = state2; }

state2:

event1 >>> {  ---- c-code -----; sense3(...); --- c-code ----;
              STATE = state1; }

event3 >>> etc.
```

Figure 1. Example kmac program structure.

seemed to be the simplest way to take into account the fact that the preprocessor switches between three distinct states during the translation of a kmac program. The preprocessor needs to handle pvar declarations, right-hand sides of rules, and the main program struc-

tural delimiters differently from one another. This different handling is performed in the three lexical scanners. When scanning the pvar declarations, the preprocessor is building a symbol table of the variable identifiers, so that it can treat them specially when they are encountered on the right-hand sides of rules. When a right-hand side is being processed, except for the special marking of the pvar references, the preprocessor is usually just copying C++ input into C++ output. Only when the delimiters of the main sections of the kmac program or the left-hand sides of rules are being processed is the preprocessor generating output code that is substantially different in structure from its input. For example, when the preprocessor is processing the left-hand side of a rule, the nesting of kmac rules within states is being translated into nested C++ "case" statements. The branching of the inner case statement is controlled by an event identifier passed as a parameter to the BMA procedure by the simulator. The outer case statement is controlled by the value of the built-in (and, of course, persistent) "STATE" variable. The processing of the contents of the FUNCTIONS, LOCALS, and PVARS sections and the right-hand sides of rules, since they already consist of C++ code, is so much simpler that the parser does not even participate; the lexical scanners handle the translation and just return the parser a token that indicates that a section of C++ code has been completely processed.

Figure 3 is a sketch of a BNF for kmac, adapted from the *yacc* grammar that defines the

kmac_program	::=	functions pvars_declaration locals state_block_set
functions	::=	ϵ | "FUNCTIONS" "[[" *c-code* "]]"
pvars_declaration	::=	"PVARS" "[[" *c-style variable declarations* "]]"
locals	::=	ϵ | "LOCALS" "[[" *c-style variable declarations* "]]"
state_block_set	::=	state_block_set state_block | state_block
state_block	::=	*statename* ":" event_list
event_list	::=	event_list rule | rule
rule	::=	left-hand-side ">>>" right-hand-side
left-hand-side	::=	*eventname*
right-hand-side	::=	"{" *c-code* "}"

Figure 3. Simplified BNF description of the kmac syntax

preprocessor's LR(1) parser. The start symbol is "kmac_program." Terminal symbols are delimited by double quotes. The identifiers and sections of C++ code that the lexical scanners handle independently of the parser are shown in italics. The symbol ϵ represents the empty string.

5. Suggested Improvements

The following three improvements to the design of kmac would have been at the top of our list for future work on the language:

(1) *error diagnostics* – the present version of the kmac parser makes no attempt to provide the BMA programmer with useful kmac syntax error messages. The *yacc* grammar should be updated to include printing meaningful messages as soon as it is known that the parse is unsuccessful.

(2) *user view of events* – it would simplify kmac programs if, in some cases, the event that is activating the BMA could be classified into some category meaningful to the BMA even though it is not known to the simulator. For example, the current implementation of the communication technology module activates a BMA with the "BMA_COMM" event when a message arrives. The BMA program would be simpler, however, if the preprocessor could automatically insert code that checked a type field in the message header and then branched to code that responded only to that type of message. Instead of the general "BMA_COMM" event appearing on the left-hand-side of a rule, more semantically meaningful event names like "Receive_Alert_Message" or "Receive_Target_List" could be used in the kmac program.

(3) *consistency check of platform configurations* – it obviously makes no sense for a BMA to expect an activation event or call a function associated with a technology module that is not attached to that BMA's platform. In the present implementation, the consistency check between the technology module calls in a BMA and the platform configuration table is not performed until runtime. This is wasteful. Since these assignments are certainly going to be static, this consistency checking should be done at compile time.

References

[1] Mizell, D., Tung, Y., Coatney, S., Carter, S., Sherman, R., and Najjar, W., "On the Design of ISI's Initial Prototype SDI Architecture Simulator," ISI Research Report, ISI/RR-88-205, March 1988.

[2] Lesk, M., "Lex -- A Lexical Analyzer Generator," Computer Science Technical Report No. 39, Bell Laboratories, Murray Hill, NJ, October 1975.

[3] Johnson, S., "Yacc: Yet Another Compiler Compiler," Computer Science Technical Report No. 32, Bell Laboratories, Murray Hill, NJ, July 1975.

[4] Coatney, S. and Mizell, D., "ISI's SDI Architecture Simulator: Initial Prototype User Interface," ISI Research Report, in preparation.

[5] Zussman, F., Operations Research International, Briefing presented to the Eastport Study Group, June 1985.

Appendix -- A Complete BMA Example

I) Introduction

The two BMAs (newbma1.bma and newbma2.bma) implement a subset of a centralized command and control SDI architecture. They utilize most of the current capabilities of the simulator:

- The kmac language
- Derived message classes
- BMA data structures which are class objects
- Multi-threaded BMAs

Preceding the source code of the BMAs is the BMA header file that was developed along with them. Most BMAs will require the use of various user-defined, possibly BMA-specific, new types and classes. The simulator's user interface supports the use of a BMA header file that contains any necessary declarations of user-defined types. This allows the user to include user-defined types among a BMA's pvars, which can result in far more readable and modular BMA programs.

The platform type definition menu of the user interface allows the user to specify the name of the BMA header file that the BMA(s) will require. The front end will prepend the following to the BMA executable files produced by kmac:

```
#include "bma_interface.h"

#include "newbma.h"

#include "bma_pvars.h"
```

The declarations of all the external (technology module) functions as BMA is allowed to call are contained in bma_interface.h.

Newbma.h is the user-specified BMA header file

Bma_pvars.h contains the C++ declarations of structures containing each BMA's pvars. Bma_pvars.h is included after the user BMA header file so that any user-defined types will be in scope for pvar definition. The user BMA header file is also included when the simulator user interface automatically generates the file init_plats.c, where the functions that allocate space for pvars are defined.

II) General description of prototype simulation architecture

The architecture consists of a single geosynchronous sensor/command platform and a large number of weapon platforms. All missile launch events are assumed to take place

within the sensor field of the single geosynchronous sensor/command platform. In addition to a sensor and a computational capability, the sensor/command platform has a single communication capability ("CHANNEL0") that can reach all the weapon platforms.

The weapon platforms each contain a communications channel and a single laser weapon. Local sensor capabilities of the weapon platform are abstracted into the behavior of the laser weapon. The weapon platforms occupy Walker orbits converging over a point in the northern USSR chosen to be near the latitude the missiles will be crossing during the window when they can be fired upon currently the boost phase, where the missiles are still very near their launch points.

This is a fully centralized architecture, where the sensor/command platform is assumed to have complete global knowledge, and the weapon platforms merely blindly obey "shoot" orders.

The architecture is intended to give a reasonably efficient (in weapon expenditure) response to a threat, subject to the constraint that a given target will only be shot at once. The basic operation consists of the sensor/command BMA detecting a new target, selecting the weapon platform with the highest probability of kill (Pk) for the new target. The command BMA then passes the targeting information to the platform that was selected to make the attack. Note that the allocation of weapon platform shots to targets is one pass, i.e., a shot is committed when the target first appears and is never rescinded/replaced by a different shot, even though a "backtracking" algorithm would probably result in a more efficient allocation of shots.

III) Operation of the sensor/command BMA

Data Structures

The primary data structure of the sensor/command BMA is the shot_assignments array. This array contains one entry for each weapon carrier platform. Each such entry contains a list of shots currently assigned to that weapon platform. A shot is defined by the target of the shot and when the shot is to take place.

The class shot_assignment is declared in newbma.h, but newbma1.bma contains the code that implements the member functions best_pk(), nearest_before(), and nearest_after(). Only best_pk() is called by the BMA proper; the other member functions are used inside best_pk() but are not called directly by the existing BMA. The function best_pk() returns the best Pk achieved by a given platform against a given target (assuming a LASER_WEAPON capability) and when the shot that achieves that Pk should occur, given

the constraints of the weapons behavior and the current commitments of the particular weapon. Best_pk() can be viewed as a "higher-level" BMA helper. It is written as part of the BMA only by historical accident: the stub function recharge(), intended to model the recharge delay of the weapon, should really be part of the BMA helper associated with the laser weapon technology module.

A inter-platform message format ("target_assignment_message") is shared by both BMAs. The message content and operations are described in newbma.h. In this case, the message format is simple (e.g., no variant fields), so the constructor simply takes the minimum information content arguments (target reference and time to shoot).

Execution

The initialization is fairly straightforward. The number of weapon platforms could be determined by examining the platform list, but it is hard-coded in this simple example. The shot_assignments array is created by the C++ dynamic allocation function new() rather than simply being declared (with static scope) to illustrate a BMA programming point: variables whose scope is static cannot ordinarily be used in a BMA, since any such variable would in effect be shared among all instances of the platform containing that BMA. Pvars are allocated separately for each platform, but the allocation takes place before BMA initialization. Since the pvars for a given BMA are implemented as a C++ class, they cannot be initialized (initialization is used here in the C/C++ language sense), because that would require kmac to emit the appropriate constructor, a function not currently implemented.

The main loop of the sensor/command BMA is executed each time a BMA_SENSE activation takes place. Targets are obtained from the sensor one at a time. For each weapon carrier platform, the member function best_pk() is used to compute the Pk of the best possible shot, taking into account constraints imposed by the shots the weapon carrier is already committed to. The returned value is compared to the best Pk found so far. If it is higher, the current trial shot becomes the current optimal shot. At the end of the loop over all weapon platforms, a target_assign_message is created containing the targeting information and the time of the shot. The message is then immediately enqueued for the chosen weapon carrier platform.

IV) Operation of the weapon BMA

Data Structures

The primary data structure of the weapon BMA is the shot_targets array. This array indicates which target a given shot is aimed at. The weapon BMA also maintains an implied data structure that maps shots in the shot_targets array to times at which each

shot is scheduled. This data structure is maintained within the simulator event queue by means of the delay call.

Execution

The activation of the weapon carrier BMA at the BMA_COMM condition indicates receipt of a new target assignment, as no other message types are expected. The targeting information is saved in the shot_targets array, and the BMA is scheduled for activation at the appropriate time by a delay() call.

The activation of the weapon carrier BMA at the BMA_DELAY condition indicates that the simulation time is now that of an assigned shot. The implied calling parameter user_parm indicates which target should now be attacked.

V) Conclusion

The simulation environment and the kmac preprocessor make it possible to write a nontrivial BMA in reasonably clear code without a large number of extraneous details. The advantages of the object environment of C++ aid the BMA programmer in structuring the BMA. It is gratifying to note that a nontrivial BMA can be written in 300 lines of code:

newbma.h	64 lines
newbma1.bma	210 lines
newbma2.bma	39 lines

“newbma.h”: the BMA Header File

```
#ifndef NEWBMAH
#define NEWBMAH
/* structures for the new BMA */

const int SHOTS_PER_PLAT = 10;

class shot_assign {
public:
        char shots_used;
        // next two arrays ordered by increasing time
        target *assigned[SHOTS_PER_PLAT]; // which target to shoot at
        stime shot_times[SHOTS_PER_PLAT]; // when

        // constructor
        shot_assign()
        { shots_used = 0; }

        // insert a new target-time pair in the allocated shot list
        // maintaining time-ordering
        boolean insert(target*,stime);

        // whether a shot can legally be made at a given time,
        // given the other shots the weapon is currently committed to
        boolean legal(stime);

        // time nearest to the desired time that is before the desired time
        // and is legal
        stime nearest_before(stime);

        // time nearest to the desired time that is after the desired time
        // and is legal
        stime nearest_after(stime);
        //  best pk that can be obtained under constraints
        prob_type shot_assign::best_pk(platform_id plat, target *tar, stime& when);
};
```

```
// one message type for a target_message - better hidden here than in one of the
// BMA source files

const int TARGET_ASSIGN_MESSAGE_TYPE = 1;

class target_assign_message : public comm_message {
public:
        target *tar;
        stime when;

        // usual constructor
        target_assign_message(target *tp, stime time)
        { tar = tp; when = time; type = TARGET_ASSIGN_MESSAGE_TYPE; }

        comm_message* new_copy()
        { return (comm_message*) new target_assign_message(tar,when); }

        void print(ostream& to)
// print target message in ASCII but otherwise no special formatting
        {
                comm_message :: print(to);
                to << "target contained is ";
                to << *tar;
                to << "shot time = " << when << "\n";
        }
        // destructor - nothing needs to be done
};
#endif
```

"newbma1.bma": Centralized Battle Manager on the Geosynchronous Sensor

```
/* bma for hq/sensor platform */

FUNCTIONS[[

#include <math.h>
const int TRUE = -1;
const int FALSE = 0;

stime recharge()
// conservative model of weapon's recovery time after a shot
{
        const stime recharge_time = 1.0;
        return(recharge_time);
}

stime comm_delay()
// conservative model of communication delay
{
        const stime safe_comm_delay = 5.0;
        return(safe_comm_delay);
}

// insert this new shot in the assignment table maintaining time-ordering
boolean shot_assign::insert(target *tar, stime when)
{
        if (shots_used >= SHOTS_PER_PLAT) return(FALSE);
        // guess that on the average shots will be allocated FIFO order
        // so start looking for the insertion point from the tail end

        int     shot = shots_used;
        while ((--shot >= 0) && (shot_times[shot] > when));
        int     index = ++shot; //index to insert new shot
        for (shot = shots_used++; shot > index; shot--) {
                assigned[shot] = assigned[shot-1];
                shot_times[shot] = shot_times[shot-1];
        }
        assigned[index] = tar;
        shot_times[index] = when;
        return(TRUE);
}
```

```
boolean shot_assign::legal(stime when)
{
// returns FALSE if the shot is impossible in terms of the recharge
// time constraints caused by the shots already allocated

        if (when < (sim_time + comm_delay())) return(FALSE);
        // command won't have time to reach weapon carrier platform
        if (shots_used >= SHOTS_PER_PLAT) return(FALSE);
        int shot;
        for (shot = 0; shot < shots_used; shot++) {

                if (fabs(when - shot_times[shot]) < recharge())
                        return(FALSE);
        }
        return(TRUE);
}

stime shot_assign::nearest_before(stime when)
{
// returns 0 if no shot is possible

        int shot;
// find first allocated shot after time = when
        for (shot = 0; (shot < shots_used) && (shot_times[shot] < when); shot++);
// now go backward until a wide enough window is found
        while (shot >= 0) {
                if (shot_times[shot] < sim_time) return(0);
                if ((shot_times[shot] - shot_times[shot-1]) >
                   2*recharge())
                {
                        // window wide enough to fit another shot in
                        // now is there time to get the command out?
                        stime t = shot_times[shot] - recharge();
                        if (t > sim_time + comm_delay())
                                return(t);
                        else
                                return(0);
                }
                shot--;
        }

}
```

```
stime shot_assign::nearest_after(stime when)
{
// return the nearest time after the desired time when a shot could be made
// given currently assigned shots
// returns 0 if no shot is possible
        int shot;
// find last allocated shot before time = when
        stime first = sim_time + comm_delay(); // first possible
        first = ((when > first) ? when : first);
        for (shot = 0; (shot < shots_used) && (shot_times[shot] < first);
    shot++);
// now go forward until a wide enough window is found
        while (shot < shots_used ) {

                if ((shot_times[shot+1] - shot_times[shot]) >
            2*recharge()) {
                        // window wide enough to fit another shot in
                        return(shot_times[shot] + recharge());
                              }
                shot++;
        }
        // shots are too tightly spaced, return a time after the last currently
// allocated shot
        if (shots_used < SHOTS_PER_PLAT)
                return(shot_times[shots_used -1] + recharge());
        else
                return(0);
}

prob_type shot_assign::best_pk(platform_id plat, target *tar, stime& when)
{
// returns 0 if there is no possible shot from the platform to the target,
// in which case the return value of parameter "when" is undefined

        if (shots_used >= SHOTS_PER_PLAT) return(0); // pointless
        stime opt_time = opt_pk(tar,plat);
        if (opt_time == 0) return(0); // never in range
        opt_time = ((opt_time > sim_time + comm_delay()) ?
                  opt_time : sim_time + comm_delay());

        if (legal(opt_time)) { // optimal shot is legal
                when = opt_time;
```

```
                return(pk(tar,plat,opt_time));
        }

        stime nearest_before_t = nearest_before(opt_time);
        stime nearest_after_t = nearest_after(opt_time);
        prob_type pkbefore = 0, pkafter = 0;
        if (nearest_before_t > 0) pkbefore = pk(tar,plat,nearest_before_t);
        if (nearest_after_t > 0) pkafter = pk(tar,plat,nearest_after_t);
        if ((pkbefore == 0) && (pkafter == 0)) return(0);

        if (pkbefore > pkafter) {
                when = nearest_before_t;
                return(pkbefore);
        } else {
                when = nearest_after_t;
                return(pkafter);
        }
}
]]

PVARS [[
int     num_killers; // number of weapon carrier platforms
int     first_killer; // platform_id of first weapon carrier
int     last_killer;  // platform_id of last weapon carrier
shot_assign *shot_assignments; // pointer to array of shot assignments of
                               // all weapon carrier platforms
]]

LOCALS[[
]]

state1:

BMA_INIT >>>
{
        init_staring_sensor(pid, 0.3, 0);  // sees all Asiatic USSR

        // figure out number of killer platforms somehow - hardcode for
  now
        num_killers = 2000;
        first_killer = 1;
        last_killer = 2000;
        shot_assignments = new shot_assign[num_killers];
}
```

```
// major loop - for new target list
BMA_SENSE >>>
{
        target_list *tgtlist = get_target(pid); // all targets since last sense
        target_list_iterator next_tar(tgtlist);
        target *this_target;

        while ((this_target = next_tar()) != NULL) {
                if ((this_target->get_visibility()) == INVISTOVIS)
                        { // new target

                        shot_assign *p = shot_assignments;
                        platform_id plat,best_plat_so_far;
                        stime when, time_to_shoot;
                        prob_type trial_pk,current_best_pk = 0;

                        // search all platforms for the one with the best Pk
                        // vs this target
                        for (plat = first_killer; plat <= last_killer; p++, plat++) {

                                if (
                        (trial_pk = p->best_pk(plat,this_target,when)) >
                        current_best_pk)
                                {
                                        // new best shot among those tried
                                        current_best_pk = trial_pk;
                                        time_to_shoot = when;
                                        best_plat_so_far = plat;
                                }
                        }
                        if (current_pk > 0) { // some shot worth trying

                                target_assign_message* mp;
                                mp  =
                new target_assign_message(this_target,time_to_shoot);
                                send_msg(mp, best_plat_so_far, CHANNEL0,
                  pid, INTR_OFF, 0);
                        }
                        else {  // no weapon carrier shot available
                        }
                }
        }
}
```

"newbma2.bma": The Laser Weapon Platform

```
/* bma for weapon carrier platform */

FUNCTIONS [[
]]

PVARS [[
int        which_shot; // lowest number shot unallocated
target*    shot_targets[SHOTS_PER_PLAT];
]]

LOCALS [[
]]

state1:

BMA_INIT >>>
{
        which_shot = 0;
        set_msg_flag(pid,CHANNEL0,INTR_ON,0);
}

// new target assignment received from hq (event is message on channel 0)

BMA_COMM >>>
{
        comm_message *msg = get(pid, CHANNEL0);
        target_assign_message *tarmsg = (target_assign_message *) msg;
        // set up an activation at the scheduled time for the shot
        delay(pid,(tarmsg->when - sim_time),which_shot);
        // save targeting information
        shot_targets[which_shot++] = tarmsg->tar;
        delete tarmsg;
}

BMA_DELAY >>>
{
        // shoot at previously assigned target
        shoot_at(shot_targets[user_parm],pid);
}
```