Proceedings of the 1989 Winter Simulation Conference
E.A. MacNair, K.J. Musselman, P. Heidelberger (eds.)

TCDL - AN EXPERT SYSTEM LANGUAGE FOR WARGAMING

Joseph L. Sowers
Paul E. Rubin
Computer Sciences Corporation
Integrated Systems Division
Moorestown, NJ 08057

ABSTRACT

This paper discusses the design and implementation of the Tactical Control Directive Language (TCDL). TCDL is a special-purpose language designed for use within the Enhanced Naval Warfare Gaming System (ENWGS). It was developed to support the concept of Tactical Control Directives (TCDs) and to allow their use within the wargaming model. A TCD is a possibly complex series of actions which simulate decisions made by a Naval officer in response to a specified situation. For example, an aircraft carrier detecting the presence of unknown aircraft will launch fighters and, dependent on the actions of the unknown aircraft, will execute a variety of possible responses. While this and other doctrines could easily be implemented by the ENWGS development team as part of the base software system, it is more desirable for the users of the wargaming system to be able to develop TCDs. TCDL provides a language in which users who are generally neither programmers nor simulationists can combine ENWGS low-level wargaming primitives and conditions into complex operations. This capability effectively extends the ENWGS user interface and may in turn be used to build even more complex operations. Since the introduction of TCDs into ENWGS, many of the air operations missions (Rubin and Sowers, 1988) have been implemented via TCDs.

We discuss the features of the TCDL compiler and how the language constructs assist in the production of code that integrates into a larger body of existing software. Also discussed are the advantages of the rule-based language in this type of environment, along with implementation problems such as assigning priorities to the various rules by the compiler. Examples of the language are shown along with a discussion of how the wargaming simulation is affected.

1. INTRODUCTION

TCDL, the Tactical Control Directive Language, has been designed as a rule-based language involving situation-action combinations. It is similar in nature to the OPS5 language (Brownston, et al, 1985). TCDL also provides an extensive user support environment, although much of this is hidden from the ENWGS player. Library functions as well as form generation and reporting facilities are incorporated into the language and are produced without specific encoding by the TCD author. Language support within the integrated environment gives experienced Naval officers who are relatively inexperienced in programming simulation software a means of managing, developing, and executing new applications within an elaborate wargaming system (Buser and Rubin, 1988).

Let us consider an overview of the structure of the language. TCDL provides a strongly typed, rule-based language. It incorporates data hiding and modularity and enforces these without TCD author's being aware of the details. TCDL provides the author with a natural means of expressing the situation-action structure of the various doctrines. The nature of the data types is specific to wargaming. TCDL has a rich assortment of types in addition to such standard types as integer, string, and boolean. TCDL strongly enforces the use of types. It is also a highly structured language. Variables must be declared prior to any rule definition. The language uses several reserved words that define the structural sections of code written in TCDL. Each of these reserved words has the flavor of a natural language to the experienced ENWGS user.

As a special-purpose language within the wargaming simulation system, TCDL provides mechanisms for the runtime TCD to access information from the system and cause actions to occur within the running game. These mechanisms are supplied by a library of primitives (causing actions), view functions (gathering information), wait functions (conditionals), and model-outcome data (information from specific simulation models). In all cases, the TCDL compiler checks the number and type of parameters. Parameters may be declared optional, resulting in default values being passed to the library function. In addition, there are specialized primitives which control selection of rules as well as the runtime behavior of the resulting TCD. Once the TCD has been compiled and entered into ENWGS, it becomes an extension to the basic gaming operations.

TCDL communicates with the TCD Librarian. There are controls built into the compiler which control the actions of the Librarian. TCDs may cause an instantiation of another TCD using the TCDL verb "invoke." The compiler checks the contents of the user and system TCD library for the presence of the child TCD; if found, it further checks type and number of parameters. Should the compiler fail in its search, then a compilation error is generated. During development, the TCD author can direct the compiler to disregard the TCD library or to use a specific library. It is important to note, however, that while the TCDL compiler is one module

within an environment, it has language constructs which control or modify the actions of the other elements of the simulation environment.

2. DATA TYPES

Variables may be typed as speed, bearing, range, altitude, lat, and long; these are somewhat self-explanatory. Types may also be very specific. Some examples are:

ACT_TRK	-	An entity under the command of a participant executing the TCD
ALERT	-	A message to be sent to the player's console
ANY_TRK	-	Any entity in the game
TIME_DTG	-	A time-date group such as "221415Z DEC 89" (or December 22, 1989, at 2:15 p.m. Greenwich time).

TCDL is an extremely strongly typed language. Parameters, assignments, and conditions are checked by the compiler. Mixing of data types except through the use of explicit conversion functions is completely forbidden. In the context of the TCD application, this rule provides an additional safeguard in that a TCD author could not enter a variable typed as a "bearing" in a use requiring a "latitude" - even though both may be measured in degrees. The strong typing is extended to the return values of the various inquiry (or "view") functions. Where necessary, type conversions are permitted provided they are explicit.

TCDL provides three primary means of access to the underlying ENWGS game environment: primitives, view functions, and wait functions.

Primitives correspond to the interactive user language employed by an ENWGS game participant at the workstation. The subset of these commands allowed by TCDL is the action type of command. Action commands allow the TCD to control the various ENWGS game entities. For example, the MANEUVER primitive corresponds to the MANE keyword and allows the course, speed, and altitude/depth of a platform to be changed. The arguments of the primitive match the input fields of the interactive form seen by a game participant. The fields match by both TCD data type and required/optional attributes. Optional arguments whose values are not given are represented by $. Thus, to assign a speed of 600 knots to the aircraft "batman," the following would be written:

```
maneuver (batman, 600, $, $, $).
```

The three dollar signs ($) in the statement correspond to the optional arguments' course, altitude, and depth.

View functions provide the means to access values within the game data base. This class of functions loosely corresponds to the reports which a game participant may request. Due to the strongly typed nature of TCDL, each view function has a specific TCD data type return value. These values may be assigned to local variables (of the same data type) within a TCD and may then be used either as part of the situation section of a rule or as arguments to primitives, wait functions, or even other view functions. Moreover, view functions may be nested if it is not necessary to save intermediate values. An example of a view function is:

```
tr_type = track_type (robin)
```

where tr_type is a local variable of a type string, robin is a local variable or a parameter of type any_trk, and track_type is a view function which returns a string designating the type of track: air, surface, or subsurface.

Wait functions are the means by which TCDL accesses the various conditions that may occur during a game. Wait functions are the primary method of specifying conditions within the situation section of a rule. A wait function only returns the values "true" or "false," corresponding to whether or not the indicated condition has occurred. The wait function

```
launch_complete (base_cmd, $)
```

returns a "true" whenever the air base designated by base_cmd completes the launch of a flight. Note the $ in the argument list representing an unentered optional argument which will cause the argument either to use a default value or, more usually, to retain its current value. A dollar sign in the position where an altitude would be expected would result in the use of the aircraft's existing altitude.

A special feature of certain wait functions and primitives is the concept of a model-outcome. A model-outcome refers to data other than the explicit return value associated with the function or primitive. This situation arises when the condition or primitive is directly related to an ENWGS model. For example, the above wait function, launch_complete, only returns a true or false. However, since it is associated with the aircraft launch model, the identity of the aircraft (track) that was launched is available and indeed is usually required. To retrieve and use this data, the wait function is prefixed with the model-outcome construct "[&n]" when it is used, where n is a single-digit integer. The model-outcome can then be used as an argument to a view function which will transform it into the required data type. A sample sequence may be:

```
...
[&1]launch_complete (base_cmd, $);
action: "Launch completed";
flt = mo_trk_launched ([&1]);
...
```

where "..." represents the rest of a rule. In this example, "flt" would be a local variable of type any_trk. The view function "mo_trk_launched" transforms the input model-outcome into a track.

3. RULES: SITUATION AND ACTION

There are four types of rules defined in TCDL. Each has a specific use. However, each follows the same syntax:

```
ruletype: rulename;
situation: quoted-description;
situation-statements;
action: quoted-description;
action-statements;
endrule;
```

The action rule, or arule, may be used only once in a given TCD. It is executed, or "fired," when the TCD is instantiated. This rule is conditionless; that is, the situation-statements section shown above must be null (empty). The arule is primarily used for initialization. Its use is not required, and if a sequence of initial actions is not needed, the arule may be absent from the TCD. The *validation rule*, or vrule, is used to check the validity of either existing conditions within the wargame upon TCD instantiation or the validity of the parameter data input by the user. There may be as many vrules in the TCD as the author wishes. Like the arule, however, there is a restriction in the vrule. The action portion of the vrule admits only one statement. This is the function: send_error_message. Should the vrule fire, the action portion will issue a message to the player's console, and the TCD is terminated. Again as with the arule, the situation section of the vrule is tested on TCD instantiation. Once tested, a vrule either fails and is never tested again or fires, thus causing the termination process. Vrules need not be used in any given TCD; however, good programming practice usually ensures the use of one vrule for each critical parameter. The decision-making rules are of the following two types, *single-fire rules* (srule) and *multifire rules* (mrule). The major difference is underscored by the designation. The srule can fire only once and is no longer part of the rule-set of the TCD. The multifire rule can be fired as many times as its situation section returns a "true."

Each of the segments, situation or action, may be composed of a block of statements. Interpreting the situation section as an "if" or condition, the separate statements are treated as though they were joined by "AND" conditions in a conventional language. For example, the situation sequence:

```
track_type(batman) = "air";
track_type(robin) = "surface";
course(batman) ^ = course(robin)
```

would be seen as requiring "batman" to be an aircraft AND "robin" to be a surface ship AND "batman" and "robin" to be traveling in differing directions before the associated action statements could be executed. Note the use of " ^ = " to denote "not equal." Should a rule fire, all of the statements within the action section will be executed in sequence.

4. TCDL ENVIRONMENT

A substantial portion of the TCDL syntax exists to support the surrounding environment - both the general ENWGS system and, specifically, the TCD support environment (see Figure 1). The player's view of the ENWGS system is that of a geotactical display, several automatic status boards, and an alphanumeric console by which commands may be issued to the system. The human interface is by means of menus or, in ENWGS, forms. Once a TCD is compiled, it joins within the system many other procedures which may be invoked via form selection. In order for these forms to become available to the player, the TCD name must be placed on a menu; the name must also be able to generate a menu of its own to prompt and receive data relating to its input parameters. This arrangement becomes more involved than it seems because the TCD software resides on a host mainframe, while the player is interacting with a workstation. Communication must be performed through messages passing between host and workstation. The form (menu) must be generated, and data types, communications, etc., must be built by the TCD without the player's being tasked with too many details. This is done by a combination of the TCDL language syntax, a form generator, and a TCD library.

Several TCD verbs relate specifically to the form generation phase. These verbs include category, keyword, directions, and prompt. The verb category defines the menu on which the TCD's name will appear for selection. The verb keyword is used to define the shortened name used to invoke the TCD. The form will show a header composed of the lines of text following the "directions" verb in the TCD. The input data is received via fields which are prompted for by text strings defined by the TCD author using the prompt verb. In addition, a qualifier "optional" in TCDL will result in the form's requiring, or not requiring, the input of any specific field. Part of the package generated by compiling a TCD is input to the form generator providing the interface to the player. This feature is integrated into TCDL in such a way that often the player/author is not really aware that the menu used to invoke the TCD is being produced within the TCD itself.

Another element of the TCD environment is the TCD library. The library maintains all of the TCDs for a player. When a TCD is created, the librarian invokes the TCDL compiler, creates a form, and adds the (new) TCD to the player's library. Should the TCD already exist in the library, the librarian will query the author as to whether this is a replacement. The librarian also performs some type checking as to whether this TCD is "invoked" by (or is invoking) another TCD. (Invoke is a TCDL verb which causes the instantiation of a child TCD.) If this TCD is invoked, the number and types of the parameters are checked for consistency. When a player logs into an ENWGS game, a test is made for the presence of a TCD library. If found, the librarian is polled and a comparison made to the TCDs already residing on the workstation. Should the workstation not have TCDs or if they are different from those in the player's TCD library, then those in the library are downloaded to

the workstation. Depending on the number to be downloaded, the system determines whether the workstation should be purged and a full library download executed, or if a selective download of TCDs is more efficient.

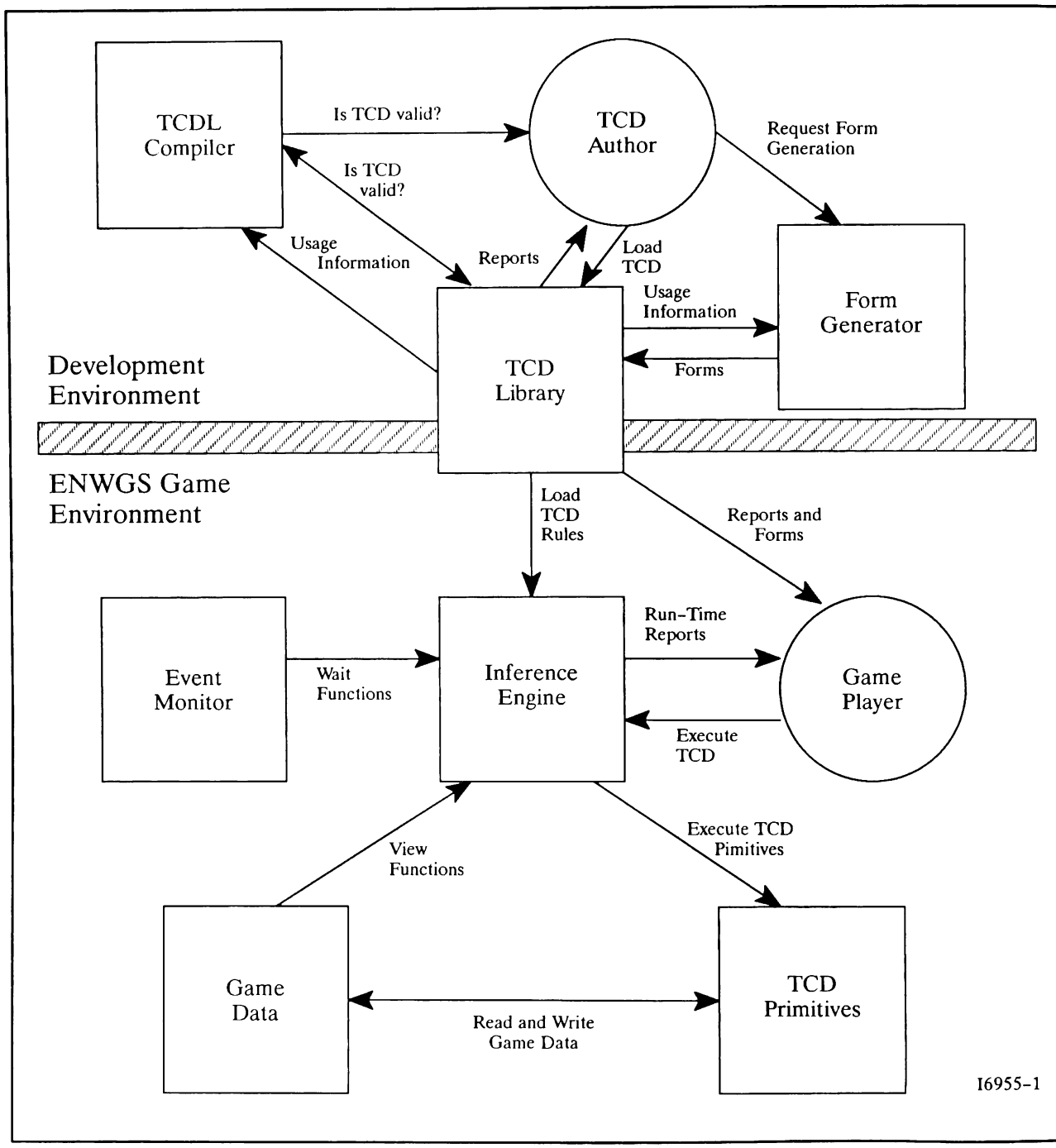

Figure 1: The TCDL Environment

The command line invocation of the TCDL compiler can, to an extent, control the use of the library. There are a number of switches on the command line:

```
tcdl TCD {-list | -map | -nocode | -nowarn | -debug
          |-nolibrary | -library LIBNAME |
          -trace }
```

where all of the switches are optional. There are default values for each option, and most may be abbreviated. The interpretation of the switches may be found in Table 1.

Table 1: The TCDL Compiler Options

Switch	Abbreviation	Default	Description
-list	-ls	NO List	Provides a numbered listing of the TCD showing errors (if any).
-map	-mp	NO Map	Provides a symbol table listing and a cross-reference.
-nocode	-nc	Code	Determines if the compiler is to check syntax only.
-nowarn	-nw	Warn	Prevents the emitting of non-fatal-error messages.
-nolibrary	-nl	Library	Specifies that the library is not to be checked. Useful when first developing a new TCD.
-library NAME	-lib NAME	SYSLIB	Allows the use of TCD libraries other than the default system.
-trace	-tr	NO Trace	Shows the actions of the parser during TCD compilation.
-debug	-db	NO Debug	Shows actions of compiler and the generation of intermediate source code. Generally used by the TCDL compiler maintainers.

5. ANALYSIS OF THE "RED_OCTOBER" TCD

Let us take a close look at the features of the sample TCD "red_october." The TCD listing may be found in Appendix A. The object of the TCD is to control the defection of a submarine (with apologies to Tom Clancy; Clancy, 1984). The first line in "red_october" defines the name and the list of input parameters. Next, the category line specifies the name of the menu on which this TCD is to appear on the player's workstation, in this case "surface engagement." Next, the keyword clause defines the string by which the player can invoke this particular TCD. The following two clauses, directions and summary, are multiple line entries. The directions (up to four lines) will appear on the player's workstation when the keyword ZOCT is entered. The summary (up to 15 lines) will be printed in response to a request for a report on the TCD and is usually much more detailed as to the functionality of the TCD. Both summary and directions are optional and need not be present in the TCD, whereas category and keyword are required. However, order of appearance is arbitrary.

The next section defines the input parameters in more detail. An entry is required for each parameter in the list on the heading line. For each parameter, however, only the parameter line itself is required, specifying the name of the parameter and its data type. In addition, the line may specify that the parameter need not be entered by the player (by using the word "optional"). If the word optional is not part of the parameter statement, then the player must enter a value on the input menu generated by the TCD. The author may also define the prompt seen by the player for each parameter by using the prompt clause. The author may also specify, by using the init clause, an initial value for the parameter, although this initial value could be overwritten by the player when filling out the menu. Order of entry is not important as long as each parameter has been qualified by a parameter block. Following the expansion

of the parameters, local variables are (optionally) declared. These are invisible to the player.

The next section of the TCD specifies the various rules. Note that comment lines are flagged by a leading "%" and continue to the end of the line. The TCD is ended by the word "endtcd;".

Now let us look at the rules for "red_october." Starting off is a vrule which checks to see if the player has input the name of a submarine. If the entry is not a sub, then an error message, error code 570, is emitted, and the TCD is terminated. If the entry is a submarine, then the TCD is instantiated and the arule is fired. In the sample arule, local variables are assigned values from functions which check to see if the player actually entered values for speed and depth. Had these optional parameters not been entered, then these switches would be set to FALSE otherwise TRUE. In addition, other local variables are initialized. The rest of the rules are now ready to fire should their situations become true. The first two srules will fire if the speed and/or depth are not entered by the player. In one case, the speed would be set to the endurance speed of the sub; in the other case, the depth would be set to 200 feet. Once fired, an srule is no longer available for consideration; it has been expended. The srule "run_for_it" will fire as soon as both the depth and speed flags become true, a situation which will occur either because of player input or because one or both of the first two srules fired. This rule will cause an intercept to the designated point at the speed and depth required. The function "intercept_pt" is known as a TCD primitive and is the primary mechanism for causing actions to occur within the surrounding simulation.

After this, the only remaining rules available for firing are the srule "defection_area" and the two mrules. The srule will fire only when the submarine has reached the required destination. Its primitives will then cause the sub to surface (depth zero) and heave-to (speed zero) at the destination point. Its final action will be to terminate the TCD and return control of the submarine's subsequent actions to the player. While proceeding to the destination, however, the mrules come into play. An inference engine continually checks the state of viable rules. In the mrule "red_force_found," the model-outcome wait function becomes true if the sub's sensors discover the presence of a "friendly" (Red) force; if this happens, the name of the detected force will be placed into the model-outcome designator [&1]. However, this rule will only fire if the sub is not currently using evasive actions as indicated by the state of the local variable "evading." The first friendly detection will cause the action segment to fire and change course to 45° off the detected entities bearing and dive to 400 feet. The other mrule, end_evasive_action, will only fire if the sub is 'evading' and the current game time is 10 minutes after the start of evasive action. If this rule fires, a new course is plotted to the destination point at the initial depth and speed. Because these are mrules, this sequence of evade-resume course may be repeated until the TCD is terminated by the final srule.

ACKNOWLEDGMENTS

We wish to acknowledge our coworkers on the TCD subsystem, Denise A. Roberts and Jon Franklin Buser. Their abilities and efforts have contributed greatly to the success of the TCD subsystem.

The work reported in this paper was developed for the Department of the Navy, Space and Naval Warfare Systems Command, under Contract Number N00039-84-C-0025.

APPENDIX A: A SAMPLE TCD

```
tcd red_october (submarine, c_speed, c_depth, d_lat, d_lon);
category: "SURENG";
keyword:  "ZOCT";

directions:   "Enter the designation of a submarine,
               its(optional)";
              "cruising speed/depth, and a destination";
summary:      "The submarine designated will attempt to reach
               the ";
              "specified location without being detected. If
               the ";
              "cruising speed is specified, it will maintain that ";
              "speed except when evading hostile detection.
               If ";
              "detected by 'friendly' forces, it will take evasive";
              "action; on reaching the destination, it will
               surface";
parameter  submarine   act_trk;
prompt:    "Submarine";

parameter  c_speed     speed optional;
prompt:    "Cruising speed (knots)";

parameter  c_depth     depth optional;
prompt:    "Cruising depth (100's ft)";

parameter  d_lat       lat;
prompt:    "Destination latitude";

parameter  d_lon       long;
prompt:    "Destination longitude";

local  depth_entered        boolean;
local  end_evade_time       time_dtg;
local  evading              boolean;
local  l_depth              depth;
local  l_speed              speed;
local  red_bearing          bearing;
local  red_track            any_trk;
local  speed_entered        boolean;

vrule: validate_sub;
situation: "Verify track is a submarine";
       track_type(submarine) ^ = "sub";
action:   "Type error";
       send_error_message(submarine, 570);
endrule;

arule: at_start;
situation: "Upon TCD initiation";
action:   "Set values";
```

```
        speed_entered = optional_entered(c_speed);
        depth_entered = optional_entered(c_depth);
        l_speed = c_speed;     l_depth = c_depth;
        evading = "false";
endrule;

srule: check_speed;
situation: "Speed not entered";
        speed_entered = "false";
action:   "Set speed value to endurance speed";
        l_speed = endur_speed(submarine);
        speed_entered = "true";
endrule;

srule: check_depth;
situation: "Depth not entered";
        depth_entered = "false";
action:   "Set depth to 200 feet";
        l_depth = 2;
        depth_entered = "true";
endrule;

srule: run_for_it;
situation: "Start defection";
        speed_entered = "true";
        depth_entered = "true";
action: "Initial intercept";
        intercept_pt(d_lat, d_lon, submarine, l_speed, $, $,
        l_depth);
endrule;

srule: defection_area;
situation: "Arrived at defection point";
        intercept_complete(submarine) = "true";
action:   "Surface sub and hold position";
        maneuver(submarine, 0, $, $, 0);
        terminate_tcd(Y);
endrule;

mrule: red_force_found;
situation: "Detected Red forces";
        [&1] class_friend(submarine, $, Y) = "true";
        evading = "false";
action:   "Evade capture";
        red_track = mo_detected_trk([&1]);
        red_bearing = mo_bearing([&1]);
        evading = "true";
        end_evade_time = add_time(now(), 0010);
        %Change course & dive to 400 ft for 10 minutes
        maneuver(submarine, $, plus_bearing(red_bearing, 45), $,
4);
endrule;

mrule:  end_evasive_action;
situation: "End evasive action";
        evading = "true";
        now() > = end_evade_time;
action:   "Return to course and depth";
        evading = "false";
        intercept_pt(d_lat, d_lon, submarine, l_speed, $, $,
        l_depth);
endrule;

endtcd;
```

REFERENCES

Brownston, Farrell, Kant, and Martin (1985). *Programming Expert Systems in OPS5, An Introduction to Rule-Based Programming.* Addison-Wesley, Reading, MA.

Buser, J.F. and Rubin, P.E. (1988). User considerations and their impact on an expert system building tool for wargaming. In: *AI Papers, 1988* (R.J. Uttamsingh, ed.). The Society for Computer Simulation International, San Diego, CA, Vol. 20, No. 1, 97-102.

Clancy, Tom (1984). *The Hunt For Red October.* U.S. Naval Institute Press, Annapolis, MD.

Rubin, P.E. and Sowers, J.L. (1988). Air operation modeling in a wargaming environment. In: *Proceedings of the Winter Simulation Conference 1988* (Abrams, Haigh, Comfort, eds.). The Winter Simulation Conference, San Diego, CA, 736-743.

AUTHORS' BIOGRAPHIES

JOSEPH L. SOWERS holds a Ph.D. in Physics from Temple University. He is a Senior Computer Scientist for the Enhanced Naval Warfare Gaming System (ENWGS) at Computer Sciences Corporation. Before joining CSC in 1987, he was with Rutgers University for 18 years. His interests lie in simulation of physical phenomena, compiler design, and computer languages.

Joseph L. Sowers
Computer Sciences Corporation
Integrated Systems Division
304 West Route 38, Box N-40
Moorestown, NJ 08057
(609) 234-1100

PAUL E. RUBIN holds a B.S. from Rensselaer Polytechnic Institute and an M.S. and Ph.D. from Drexel University, all in Physics. He is currently head of the Build, Design, and Integrate group for the Enhanced Naval Warfare Gaming System (ENWGS) at Computer Sciences Corporation. His professional interests include computer-based wargaming, simulation, and applications of artificial intelligence to such systems.

Paul E. Rubin
Computer Sciences Corporation
Integrated Systems Division
304 West Route 38, Box N-40
Moorestown, NJ 08057
(609) 234-1100