DEVELOPMENT OF AN EXPERT SYSTEM ENVIRONMENT FOR USE WITH A WARGAMING SYSTEM

Paul E. Rubin, Ph.D. and Jon Franklin Buser
Computer Sciences Corporation
304 West Route 38, P.O. Box N
Moorestown, NJ 08057

ABSTRACT

This paper presents a discussion of the design of the Tactical Control Directive (TCD) environment for the Enhanced Naval Warfare Gaming System (ENWGS). TCD's are a system extension which allows user's to construct and execute complex naval doctrine and tactics within the paradigm of a rule based expert system. Each of the major TCD components - TCD language (TCDL), compiler, inference engine, TCD library configuration management tools, automatic user input form generation, and run time reports - is presented with respect to major design issues and decisions; and implementation constraints imposed by a military user community.

INTRODUCTION

The Enhanced Naval Warfare Gaming System (ENWGS) is a large scale, computer based, interactive wargaming system. It supports curricula and studies at the Naval War College and Tactical Training Groups. Although the system provides players with the capability of using primitive operations such as launching aircraft, acquiring detections, or engaging in combat, it is often desirous to automatically create more extensive sets of commands to be exercised by the system. Tactical Control Directives (TCD's) will provide an environment for the construction and execution of such naval doctrine and tactics within the paradigm of a rule based expert system. The goal of this system extension is to provide the user community with a general utility that can be used to combine primitive system operations and previously defined TCD's into new and increasingly more complex tactical and doctrinal procedures. TCD's will be written primarily by the user community who are relatively inexperienced with programming in rule based languages. Creation and installation of new TCD's will be an ongoing user activity that will not require intervention by the system maintenance staff.

TCD's interact with the system in much the same manner as the game player. The commands available to TCD's are basically the same as those available to the player, and the events and data available to the TCD writer are similar to those the player can schedule, observe, or report. A TCD can appear to make decisions similar to those a human expert, in this case a naval officer, would make given the same situation. For this reason, we view TCD's as small expert systems. The TCD's differ from other expert systems in that the decision making process is highly structured; predictability is valued above the creativity that would be exhibited by an expert naval officer.

HIGH LEVEL DESIGN

The TCD environment consists of six major components: TCD language (TCDL), TCDL compiler, inference engine, TCD library configuration management tools, automatic generator of user input forms, and various reports. Figure 1 shows these components and their interactions. The development process is not unlike a computer programming development process. TCD's are written in a development environment, compiled, and written into a TCD library. The game setup and initialization process then selects the TCD's to be used for a game and loads them into the gaming environment.

Some components run in the development environment, and some run in the ENWGS game environment. The TCD library is the component that connects the two operating environments. TCD's are written, compiled, and loaded into the TCD library in the development environment. At game initialization, the TCD forms are downloaded to the participant's workstation.

TCD's are executed in the game environment. To execute a TCD, a participant fills in and transmits a TCD form. The subject TCD is loaded from the TCD library into the Inference Engine prior to execution. In the course of executing a TCD the Inference Engine may call TCD primitives, extract game data information using View functions, or monitor asynchronous events that trigger Wait functions. The TCD primitives make changes to the game data. This game data is the same data used by the ENWGS models for their operation. Execution reports are provided to the game participant by the Inference Engine. Static library reports are available in both the development and game play environments.

TACTICAL CONTROL DIRECTIVES LANGUAGE (TCDL)

The Tactical Control Directive Language (TCDL) was developed explicitly for defining TCD's. TCDL is a rule-based language that incorporates features of more conventional programming languages. The rule structure implemented was patterned after that in OPS5 (Brownston et al. 1985), and then specialized

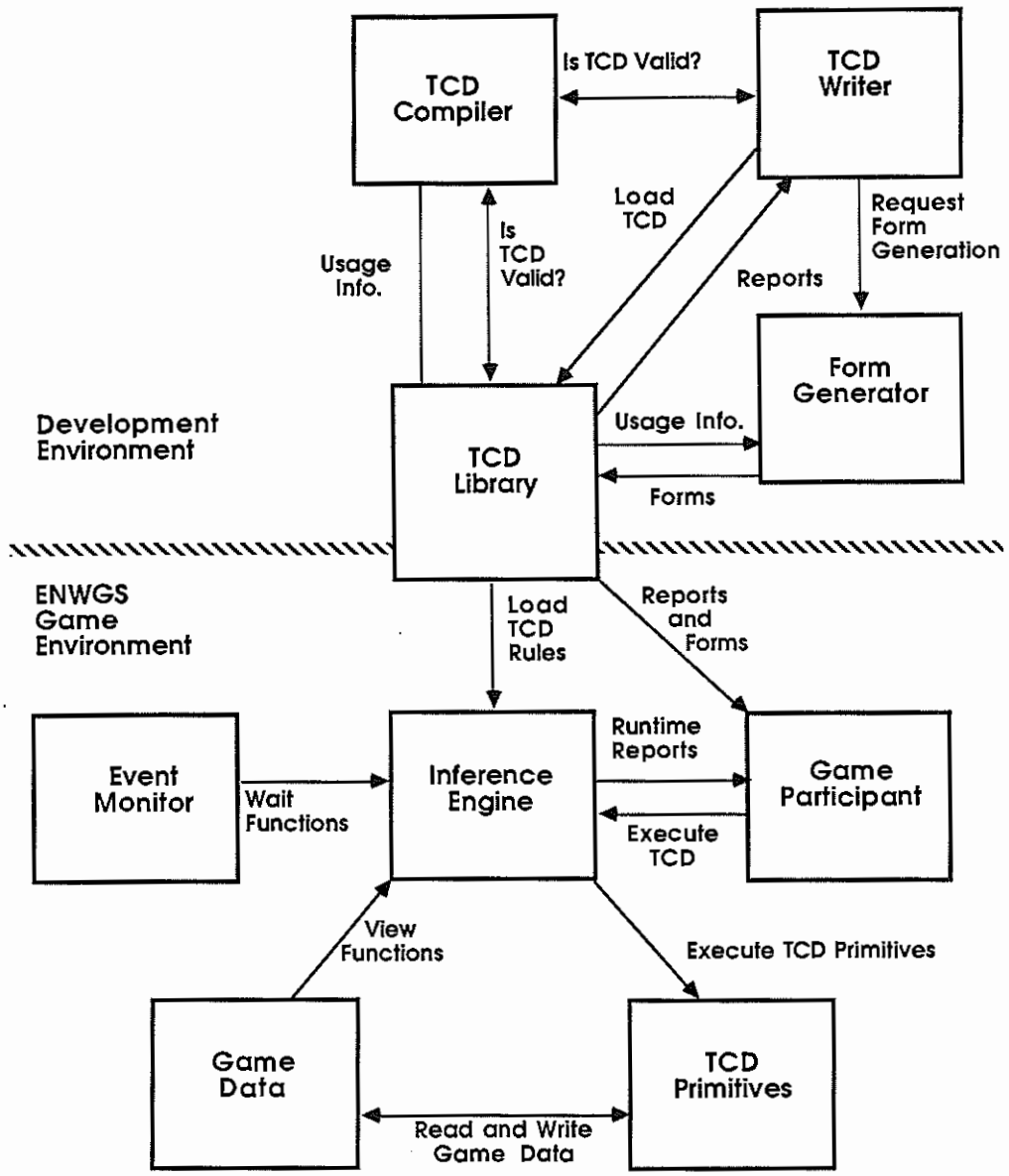

Figure 1. TCD System Components

for the ENWGS TCD application. These specializations were driven by the design goal of putting effective TCD writing within the reach of relatively inexperienced programmers. To achieve this goal, language design focused on three primary areas:

1. Rule structure
2. Error detection/validation
3. Data structures.

Rule Structure

In defining the syntax for the specification of TCDL rules, much effort was placed on allowing the TCD writer to use terms that were familiar to his/her environment. The development staff views rules as a collection of conditions on a left-hand-side (lhs), which, when all true resulted in the execution of procedures on a right-hand-side (rhs). However, in the interests of ease of presentation to the prospective user community, it was felt that the abstract concepts of lhs and rhs would complicate the understanding of TCD's. From the user's perspective, each rule represents an action or set of actions (rhs) which should be taken when a given situation (lhs) arises. Thus, the syntax developed that a rule consisted of a SITUATION part, where the various conditions are specified, and an ACTION part where the procedures to be followed are specified.

The conditions of the SITUATION part of a rule are written as a set of implied IF statements. Implied, because the "IF" itself is not part of the condition. The TCD writer has a great deal of flexibility in the specification of conditions. Previously set local variables or input parameters may be tested as in traditional programming languages. View Functions are also provided which give access to the game data base, thereby allowing these quantities to be tested. Most importantly, however, is the capability to test for asynchronous events within the gaming environment. This feature is provided by a set of Wait functions. Each Wait Function is associated with a type of event, such as a detection, a launch completion, or a low fuel alert. Wait Functions return boolean values which are false until the referenced event occurs, at which time they return a value of true.

The ACTION side of a rule resembles a conventional procedural programming language. TCD primitives are commands that correspond to keywords that are used by a player during game play. These are specified with parameters that map form fields that a player would fill in. In this manner, execution of the ACTION part of a TCD rule mirrors the activities of a human game player. Therefore, the writer of a TCD can think in the familiar terms of ENWGS game play keywords when defining TCD actions. The use of View Functions and local variables within the ACTION section simulate the human player's ability to observe and record data that is generally provided via the various ENWGS reporting mechanisms.

Error Detection/Validation

The distributed nature of TCD's allows ample opportunity for data errors to enter the process which could potentially render the entire TCD mechanism useless. TCD'S are defined, requested, and eventually executed at different times with respect to the gaming environment. Definition, i.e., writing a TCD, takes place outside of the ENWGS environment. Request of a TCD is performed by a player during game play, at which point values for the TCD input parameters are provided. Execution of an action within the TCD, and use of the TCD input values may occur at some indeterminate future time. Several steps were taken in the development of TCDL to deal with data validation issues.

TCDL is a strongly typed language like Pascal. All TCD input parameters and local variables must be declared with a data type prior to use within any rule of a TCD. The data types of TCDL include the usual types of "integer" and "character", but also include specific ENWGS data types. Examples of these are "command id", "course", "speed", and "latitude". The parameters for functions and primitives, and the return value of the functions are specified using these data types. The TCDL compiler enforces the strong typing and provides rigorous parameter type checking at compile time. Thus, it will not be possible to have a TCD execute during play and cause a fault because it tried to provide a "command id" to a primitive that required a "speed". Data types are also used to implement runtime data validation.

Data Structures

Because the members of the intended user community are not "programmers", it was not considered prudent to endow TCDL with a rich set of user accessible data structures. By the same token, a complete absence of user data structures would also be unsatisfactory. The approach finally adopted was to provide a limited, but useful set of structures.

The capability to declare, set, and reference local variables is provided by TCDL. As stated above, these must by typed before use, and the compiler will enforce data type compatibility when assignments and references are made. Although local variables could be used to implement an involved state space within a TCD, the judicious use of Wait and View Functions renders this unnecessary in most cases. Thus, it is envisioned that local variables will find most use as a scratch pad within a TCD, or to save values returned by some system models.

The other major data structure to which the TCD writer has access is the "bag". A bag is an unordered collection of (similar) items. A bag most closely resembles an array from conventional programming. Since TCD's are intended to implement both single and

multi-platform tactics, a method was needed to deal with an unspecified number of "actors" within the TCD. Clearly, it would not be adequate to have one TCD for a one helo exercise, another for a two helo exercise, and so on. The bag construct helps solve this problem. Actors, such as launched aircraft, can be added to a bag after they launch. Assignment to specific activities within the TCD can then be controlled by selecting aircraft from the bag until the bag is empty. Moreover, various criteria can be placed on these selections. For example, the "closest" aircraft to a hostile can be selected to intercept and engage.

TCDL COMPILER

A key element in the development of the TCD facility is the TCDL compiler. Several initial questions could be addressed through the use of a prototype compiler providing minimal functionality. For example, there was initial concern over whether the language constructs being specified were rich enough for TCD's to be written to replace the existing functionality. A concomitant was whether we would be able to specify and parse the needed constructs. In order to help resolve these issues, development of TCDL and its compiler were the first development tasks initiated.

An early decision was made to utilize the parser generator and lexical analysis tools available on our development system (Multics) for building the compiler. The parser generator - called reductions, or rdc - allowed easy separation of language construct, syntactical and semantic analysis, and code generation issues. The language syntax was developed via a BNF specification, and translated into the format accepted by rdc. Although a bit more primitive than other compiler generators, such as yacc on Unix, rdc was found to be completely adequate for the task. Using this tool allows relatively easy modification of the language. In addition, we were able to specify a superset of the TCDL features to be in the initial delivery.

A running parser was the first goal of the system. This provided us with a starting point for driving other aspects of the system, such as forms generation and reports. Also, a running compiler, even if not complete with respect to code generation or error messages, provides an extremely useful tool for developing the initial set of TCD's. Since the requirement was to (at least) replace the current functionality provided by composites, early warning of problems in writing such TCD's was paramount. Moreover, TCD's developed in this manner provide the test cases for eventual system acceptance.

INFERENCE ENGINE

The TCD Inference Engine is the software that causes the execution of rules in a TCDL program. Abstractly, the Inference Engine can be thought of as a device that constantly cycles through the rule set of each executing TCD. On each cycle it will identify the conflict set: the set of rules that are eligible to fire at that time. It will then select one rule from the conflict set to fire. The selection process is based on the conflict resolution strategy. This abstract description of Inference Engine functionality suggests that it is embodied in a single module. In reality, its functions are decentralized. Some of its functions are performed by code that resides within the TCD object code of each TCD. Others are performed by an external, with respect to the TCD object code, support environment.

The Inference Engine's job is to analyze TCD rules so that a single rule can be chosen to fire. The principle criterion used by the conflict resolution algorithm is specificity. This means that a more specific rule will be chosen to fire over a less specific rule. A rule's specificity is measured by the number of conditions necessary for it to fire. The more conditions given in the situation part of a rule, the more specific it is. For example, suppose rule1 requires only an "on-hostile-detection" condition in order to fire. Assume that rule2 gives two conditions: "fuel-status > 25%" as well as "on-hostile-detection". Both rules will be in the conflict set if both the conditions of rule2 are true. However, rule2 will be selected to fire because it is more specific than rule1.

LIBRARY AND CONFIGURATION MANAGEMENT (CM) TOOLS

The ENWGS development contract requires that strict configuration control be maintained on all code and data developed for use within the system. TCD's are viewed as user written extensions to the ENWGS command repertoire. Being part of the system, they therefore fall under the control of CM. However, CM as performed on contract deliverable software would defeat the objective of providing the end user with a tool to generate their own commands. The TCD Library implementation is meant to satisfy contractual CM requirements and at the same time provide the user community with useful tools for the continued development and maintenance of TCD libraries.

TCD libraries are implemented using Multics directory structures. Libraries are implemented at a minimum of two levels: the CM or system level, and the user level or levels. A TCD may be added to any TCD library only after it meets the following criteria:

1. It compiles correctly.

2. Any TCD's called from the candidate TCD must already reside in the library, and must be called with correct parameters.

3. The name of the new TCD must not conflict with other TCD's already in the library.

In order to enforce these criteria, system procedures have been developed to manage TCD libraries. Thus, before the TCD is checked into the library, the TCDL compiler is called in order to force static validation. Outputs from the compilation process are then used by other tools to enforce criteria two and three above.

The system level library is CM controlled, and project Configuration Control Board (CCB) approval is required before TCD's may be added to it. For inclusion in the CM library, TCD's must meet the following additional criteria:

1. Output from all TCD reports must be approved.

2. All functional testing of the TCD must be passed.

In addition to this topmost system library, user and game specific TCD libraries are allowed. For the most part these may be manipulated on a more casual basis by the TCD development staff in order to support TCD and game development. Each level of library, however, is complete within itself. A game is linked to a specific library, and a workstation will only have TCD's from one library resident at any given time.

Within the context of the library system, each TCD is viewed as a subdirectory of the library directory to which it belongs. Each TCD subdirectory consists of five files, which can be considered the complete specification of the TCD. These files are:

1. The TCDL source for the TCD.

2. The PL/1 code generated by the TCDL compiler for the TCD. This is retained primarily to assist any required debugging operations.

3. The executable TCD code, generated by the compiling the above PL/1 code.

4. The user input form associated with this TCD. This file is downloaded to the workstation when a game is initiated, thus making the TCD available to that player.

5. A file of the comments and Situation and Action statements extracted from the TCDL source code. These comments and statements are used to support the generation of static description and execution reports.

AUTOMATIC FORM GENERATOR

A TCD is invoked by a player within ENWGS the same as any keyword: the TCD name is entered at the workstation as a command to ENWGS. The workstation responds with a form soliciting user input specific to that TCD. Definition of user input forms for ENWGS is normally a development/maintenance activity. Although good tools have been provided for this effort, they are not suitable for use by the writer's of TCD's. In addition, it was not deemed desirable to require a maintenance staff to develop forms in support of user TCD development. Therefore it was necessary to provide an alternate means of developing TCD forms without the explicit participation of either the TCD writer or the development/maintenance staff.

An analysis of requirements for input forms revealed that essentially all of the information necessary to define a form is known, or can be known, by the TCDL compiler at compilation time. Thus a means of automatic form generation was selected. The user input forms are generated from the data type information provided in the declaration of the TCD's input parameters. The close coupling of TCD parameter declarations and input form generation provides an excellent means of achieving TCD completeness, consistency, and form correspondence.

TCD completeness means that a value is supplied for every parameter of every TCD, primitive, etc. referenced within the TCD. The TCDL compiler verifies completeness at one level by checking all such references against its database of valid calling sequences. In addition, the compiler verifies that local variables used in such calls are assigned values somewhere within the TCD rule set. By ensuring that all declared TCD input parameters appear on the user input form, an additional level of checking is achieved, since such appearance forces the employer of the TCD to supply a value.

A TCD is defined to be consistent if all parameters used in calling sequences for other TCD's, primitives, and functions, are of the correct type. Here again, the TCDL compiler provides support by enforcing rules of data type validity when processing such argument lists. This enforcement is eased by the strong typing characteristic of TCDL. As with completeness, consistency is further ensured by deriving valid input form data types from the TCD parameter declarations.

TCD form correspondence is the one-to-one mapping between TCD input parameters and fields on the user input form. Correspondence is an extension of completeness and consistency, and is absolutely guaranteed via the automatic generation process. This is a key point. If any partially or completely manual system were used, none of these three characteristics could be rigorously enforced.

The automatic form generation process within ENWGS is complicated by the distributed nature of the system: host and

workstation. This can be seen by observing that the host has knowledge of the TCD, and hence its form requirements, but the workstation is the system element that must process the form. Moreover, both host and workstation must agree on the messages that will be used to communicate the user's input from the form (workstation) to the TCD (host). This issue has been solved by generating the necessary form data on the host when the TCD is compiled, and downloading it to the workstation at game initiation time. The workstation then essentially integrates this data with the standard form data, and the TCD becomes part of the ENWGS vocabulary.

The download process assists in maintaining consistency of TCD's at a workstation. Depending on the specific TCD library used for a game, different forms can be downloaded for different games. In addition, TCD's and their accompanying forms may be selectively downloaded to a workstation based on side or specific player. Category identification embedded in the download messages for each TCD form enables the workstation to generate selection menus containing the currently available TCD's as choices.

REPORTS

During the course of a game, a player will require information about TCD execution progress, or may need to know some overview description of a given TCD. In order to satisfy this requirement, various runtime reports have been defined for the TCD environment. These reports have been designed so as to be useful to someone that has no understanding of TCDL. Moreover, care was taken to minimize the burden of report generation on the executing software, the TCD writer, and the system developers. The solution was to associate essentially free text with each TCD as a whole, and also specifically with each SITUATION and ACTION statement of the TCD. This allows the reports to be natural language summaries of the rules. The free text describing the TCD is written as a set of tagged comments. The text associated with the SITUATION and ACTION statements is an integral part of these statements. The language has been designed so that the text and code are contiguous. This results in self-documented TCD's, which in turn yields easy review and modification.

The following game play reports are currently supported:

1. Doctrine Summary Report. This report will consist of tagged comment text entered by the TCD writer and extracted by the TCDL compiler.

2. TCD Structure Report. This report must be intelligble to someone with no knowledge of TCDL, yet still present the rule structure of the given TCD. Again, our solution was to provide the text associated with the SITUATION and ACTION statements of each rule in the TCD.

3. TCD Progress Report. This report also uses the SITUATION and ACTION text. Unlike the previous two reports which are static in nature, this report represents the dynamic nature of an executing TCD. The report will summarize which rules have fired. It will also list for which conditions the TCD is currently waiting.

4. Tracks on TCD Report. Also a dynamic report, this presents a list of tracks that are controlled by the input TCD.

5. TCD's of Track Report. Another dynamic report, this produces the list of TCD's, if any, that control the input track.

In addition to the game play reports above which support the player, there are sets of reports to support TCD writers and software development/support personnel:

1. TCD Cross Reference Report: a list of all TCD's in a (specified) library, what TCD's are invoked, and which TCD's invoke them.

2. TCD Parameter Report: for each TCD in the input library, a list of the parameters and their data types.

3. Doctrine Summary Report: the same as the game play report of the same name, but in the login, as opposed to the game play environment.

4. Symbol Table Report: a runtime report to assist debug activities. This report will list the name of each object in the runtime symbol table of the input TCD and its current value.

5. Conflict Set Report: also designed to assist software development. This report will list which rules are currently eligible for firing and what data or conditions are necessary for them to fire.

EXAMPLE TCD

Figure 2 presents a simple example TCD called air_engage. This TCD is used to perform air-to-air and air-to-surface engagements. The engagement model in ENWGS monitors the platforms involved and determines when each is capable of firing. When an engagement (take) is requested in advance, the model will automatically fire weapons when the platforms are within range. The rules of engagement (roe) control whether or not a platform is allowed to fire (roe free). The intercept model automatically slaves the interceptor to the target when the initial intercept is complete.

```
% The TCD "air_engage" is used for air-to-air and air-to-surface engagements.
% This TCD will recover the interceptor when weapons
% are low, when fuel is low, or when contact is lost on the target.

tcd air_engage (interceptor, roe, target, base);

dcl interceptor act_trk parameter;
dcl roe boolean parameter init ("Y");
dcl target any_trk parameter;
dcl base base_cmd parameter optional;

   vrule: validate_interceptor;
   situation: "Interceptor is not an air track";
     [track_type (interceptor) ] != "air";
   action: "Send an error message";
     send_error_message (interceptor, "track must be type air");
   endrule;

   arule: intercept_target;
   situation: "At beginning of tcd";
   action: "Modify roe and intercept target";
     modify_roe (interceptor, roe);
     intercept (target, interceptor, [max_speed (interceptor)]);
   endrule;

   rule: engage_target
   situation: "Rules of engagement = free";
     [roe_weapons_free (interceptor)] = "true";
   action: "Engage the target";
     take (interceptor, target);
   endrule;

   rule: weap_low_recover;
   situation: "Interceptor is low on weapons";
     [weapon_alert_level (interceptor)] = "true";
   action: "Recover aircraft, mission complete";
     recover_ac (interceptor, base);
     terminate_tcd ();
   endrule;;

   rule: fuel_low_recover;
   situation: "Interceptor is low on fuel";
     [low_fuel (interceptor)] = "true";
   action: "Recover aircraft, mission complete";
     recover_ac (interceptor, base);
     terminate_tcd ();
   endrule;

   rule: contact_lost_recover;
   situation: "Interceptor has lost contact on the target";
     [lost_contact (interceptor, target)] = "true";
   action: "Recover aircraft, mission complete";
     recover_ac (interceptor, base);
     terminate_tcd ();
   endrule;
```

Figure 2. Example TCD

air_engage

The TCD air_engage is used for air-to-air and air-to-surface engagements. The engagement model will automatically choose the appropriate weapon. If a return base is not supplied, the interceptor will return to its home base.

Interceptor _____

ROE Free Y

Target _____

Return Base ___

Figure 3. TCD Input Form

Figure 3 shows the user input form associated with the example. Note that the fields on the form correspond to the TCD parameters. Field values are defaulted using the term "init" in the parameter declaration. Roe is defaulted to "Y", meaning free.

The TCD performs the following operations:

1. Validates that the interceptor is an air platform.

2. Modifies the interceptor's roe to that supplied by the user's input.

3. Intercepts the target at maximum speed, where maximum speed is supplied by a view function.

4. Engages whenever the roe is free.

5. The interceptor returns to base and the TCD terminates if weapons are low, fuel is low, or detection is lost on the target. Contact will be lost if the target is destroyed or is able to evade the interceptor's sensors. The interceptor will normally return to its home base. However, the player can optionally provide an alternate return base.

A more advanced version of this TCD could request more forces if the interceptor has to return without destroying the target.

ACKNOWLEDGEMENT

We wish to acknowledge the contributions of our co-workers, especially Dr. David Slater and Ms. Cheryl Williams. A hard working, dedicated development team is a necessity when developing complex systems from new ideas.

The work reported in this paper was developed for the Department of the Navy, Space and Naval Warfare Systems Command, under Contract No. M00039-84-C-0025.

REFERENCES

Brownston, L.; Farrell, R.; Kant E.; and Martin, N. 1985. Programming Expert Systems in OPS5, An introduction to Rule-Based Programming. Addison-Wesley. Reading, Mass.