USER CONSIDERATIONS AND THEIR IMPACT ON AN EXPERT SYSTEM BUILDING TOOL FOR WAR GAMING

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

ABSTRACT

The Tactical Control Directive (TCD) environment is an extension to the Enhanced Naval Warfare Gaming System (ENWGS). TCDs let the user create new system commands by combining and reusing primitive building blocks provided with the underlying simulator. The environment's primary components are a custom rule-based programming language and the associated inference engine for executing these programs. Other components include: user reports, a data input form generator, and configuration management tools.

TCD writers are relatively inexperienced at programming. Their inexperience has a significant impact on the system's architecture and design. This paper describes how the TCD language, development environment, and runtime gaming environment have all been driven by this paramount requirement.

INTRODUCTION

Every computer system has numerous interfaces where a human must interact with the machine: the end user running applications, the programmer implementing and debugging code, and the configuration manager producing deliverable software releases. Significant effort is required to design the interface between application programs and the end user on most modern computer systems. One reason this effort is invested is to allow effective use of these systems by people who have only minimal amounts of specialized training. Fourth generation languages and operating systems, like UNIX with its' extensive tool set and pipe facility, can reduce the level of specialized knowledge that a person needs to develop certain applications. These are examples of user interfaces that reduce the gap between end users and programmers.

The Tactical Control Directive (TCD) environment is another system whose goal is to reduce the gap between the end user and programmer. TCDs provide users of an existing computer simulator, who are relatively inexperienced at programming, with an integrated environment for developing, managing, and executing new application programs (Rubin and Buser 1987).

TCDs are an extension to the Enhanced Naval Warfare Gaming System (ENWGS). ENWGS is a large scale, computer based, interactive simulator that supports curricula and studies at the Naval War College and Tactical Training Groups. ENWGS is a general war gaming utility that provides players with primitive operations such as launching aircraft, acquiring detections, and engaging in combat. System models calculate platform kinematics, detections, battle damage, and logistics expenditures. ENWGS can support a vast range of gaming activities. Scenarios can be constructed with up to 64 players, 3 sides, and 2000 individual platforms.

There is no such thing as a typical ENWGS game. Training objectives vary at the different sites with each game. Small games may be concerned with the coordination of several platforms performing anti-submarine operations, while large games could deal with the complexities of global warfare. It is this diversity of gaming objectives that drives the TCD development effort.

In general, ENWGS commands request that a single platform perform one action. Some typical commands are: change the course and speed of a platform, intercept another platform, launch an aircraft, or engage in combat. In some game scenarios, training objectives are satisfied by controlling platforms at this level. For other games, a higher level of control is more appropriate. For example, a player may want to put a carrier defense doctrine into effect that would automatically intercept and follow any hostile air detections. The more complex commands are analogous to the orders of a high ranking officer who delegates operational details to his/her subordinates.

Tactical Control Directives provide the user community with a general utility for combining primitive system operations and previously defined TCDs into new and increasingly more complex tactical and doctrinal procedures. The developers view each TCD as a mechanism to emulate the expert behavior of a naval officer over a very limited domain. They have drawn heavily from expert systems' technology to develop a rule-based definition language for this purpose.

TCDs interact with the system in much the same manner as a game player. The building blocks available to the TCD writer are similar to the actions a player can schedule, observe, or report from a gaming console. TCDs use a forward chaining inference strategy to make decisions similar to those a naval officer would make given the same circumstances.

The greatest challenge facing the developers is that these expert systems will be written primarily by the user community who are not experienced programmers. Creation and installation of new TCDs will be an ongoing user activity that will not require intervention by the system maintenance staff.

THE DEVELOPMENT AND RUNTIME ENVIRONMENTS

The TCD environment consists of six major components: TCD language (TCDL), TCDL compiler, inference engine, TCD configuration management library, automatic user input form generator, and various user reports. Figure 1 shows these components and their interactions. Note that some components run in the development environment and others run in the ENWGS game environment. The TCD library links the two environments.

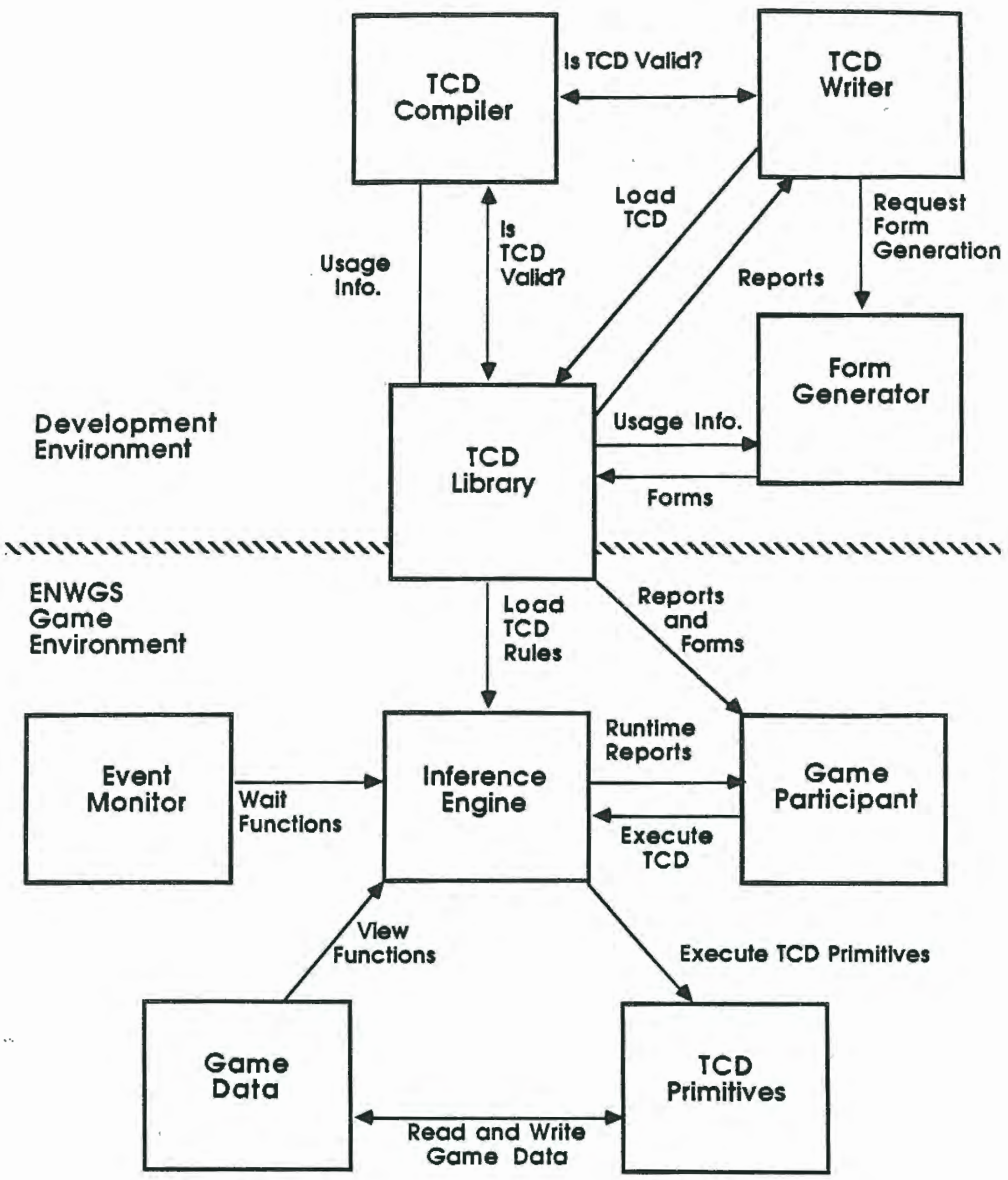

Figure 1. TCD Environments

In the development environment, TCDs are written, compiled, and loaded into the TCD library. The TCD development process is like other computer programming exercises, except that TCD "programmers" have a limited computer background (knowledge of one computer language is recommended), and a very good understanding of the ENWGS simulator. The TCD language, compiler, and library have all been designed with these users in mind. The developers have tried to minimize programming effort by defining simple language constructs and maximizing the correspondence between these constructs and the game play commands. Emphasis has been placed on static error validation to reduce runtime errors in the game environment.

When the game environment is initialized, a TCD library is selected and its TCD forms are downloaded to each participant's workstation. 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 and execution begins. While 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 game data, the same game data used by the ENWGS models for their operation. Execution reports are provided to the game participant by the Inference Engine. The format and content of these reports are very important, because the game participants do not, generally, have any knowledge of the TCD development environment.

USER CONSIDERATIONS AND THE DEVELOPMENT ENVIRONMENT

The TCD development environment is a second generation system whose design has grown out of a careful examination of user needs and abilities, and an analysis of the strengths and weaknesses of the previous system. The environment's goal is to allow a person with a good understanding of the simulator (but limited programming experience) to create new system commands, an activity that would generally be the domain of experienced software developers.

The functionality of any given TCD, once specified, could easily be implemented by the development staff in PL/1, ENWGS's development language. However, there are a number of barriers preventing this more conventional approach.

First, it is difficult to obtain concurrence within the user community as to exactly what should happen in response to varying tactical situations, and once agreement has been achieved, it is possible for requirements to change with the introduction of new military doctrine. New TCDs are also needed, sometimes on short notice, to support specific game objectives. Shifting requirements are a fundamental aspect of the problem, and are not caused by user indecision. ENWGS is used by a variety of different users whose training goals vary, even at a given site, with different game scenarios.

Second, ENWGS software is developed in accordance to a strict development methodology (DOD-STD-1679A) that follows the waterfall model (Royce 1970, Fairley 1985). In general, the methodology specifies that for each release: requirements will be established prior to implementation, design reviews will be held for each functional area, and testing will be performed both by the developer and by an independent test agent. Following this methodology aids the production of high quality software, but requires half a year to produce and distribute even a small maintenance release. Since the elapsed time required to produce a system release is greater than the time used to plan and develop many game scenarios, the normal software development cycle will not support the end user's need for new and varying TCDs.

The volatility of user requirements combined with the relatively long elapsed time between system releases suggests that the user community requires a way to produce their own TCDs. It is unreasonable to ask the users to encode their TCDs as PL/1 system code. This would require a detailed understanding of the underlying data base structure, communication message protocols, coding standards, configuration management practices, and other details that are the domain of the professional software developer. Instead, the developers sought a way to provide the user community with a high level programming capability that corresponded with the user's view of the simulator: the existing game play interface.

Primitive operations (e.g., engage a target) in the TCD development environment correspond (in name and arguments) with the ENWGS commands that players execute in the game environment. Conditional events (e.g., upon receipt of a hostile detection) are also available and correspond to their game play counterparts. A Model Outcome capability is provided to pass data generated from a condition (e.g., what track was detected) to primitive operations. A simple TCD could direct an aircraft to wait on station until it receives a hostile air detection, then vector the aircraft towards the target and engage when at an appropriate range for the weapons on board. Given a complete set of primitives and conditions, fairly complex, and seemingly "intelligent" tactical missions can be developed.

The original Naval Warfare Gaming System (NWGS) was delivered with a subsystem that implemented similar functional requirements. Composite Verbs were built by linking the form arguments of one Verb to those of another. A Verb could be a primitive operation, a condition, or a previously defined Composite. Composites were built from the bottom up creating a binary tree structure. The user interface was form driven and would prompt the user to supply the source of each argument: user input, a link to another argument, or a model outcome value. The system was a success in the sense that previously defined Composite Verbs were regularly used by game players. The Composite Verb writers, however, found the definition interface clumsy and unnatural and, as a result, very few attempts were made to expand the set of available composites beyond those originally delivered with the system.

Feedback from Composite Verb users suggested that, for the next generation system, a definition approach that used text files would be easier to manage. The original TCDs were envisioned as an interpreted language that would look vaguely like UNIX shell scripts. However, because the TCD scripts would be executed a line at a time, it was determined that this approach would not satisfy an implicit requirement to simultaneously wait on multiple conditions. Conventional procedural languages were also investigated, but multiple conditions caused TCDs represented in this way to become very deeply nested and unmanageable. Finally, a rule based production language approach was adopted.

Once the language issue was settled, the developers focused on other aspects of the development environment. It was clear that two additional components would be needed to provide a truly integrated environment: the TCD library and the Form Generator.

The TCD Library acts as the interface between the development environment and the game play environment. The library is used to assure TCD quality. Game players cannot use a TCD unless it has already been entered into a library. Before a TCD can be entered into the library, it must pass a number of validation checks that ensure library integrity: the TCD must compile without errors, the TCD cannot invoke another TCD unless it already exists in the library; and, if a TCD is being replaced by a new version and it is invoked by another TCD, the new TCD must maintain the same input parameters as the original.

The development environment supports multiple libraries. This allows TCDs to be written and tested without running the risk of damaging TCDs that are being used in active games. It also allows special libraries to be used for different games.

The Form Generator is a critical link in the TCD development environment. It ensures that the data input form displayed to the game player is in exact correspondence with the TCD that will be executed. The distributed nature of the ENWGS hardware architecture makes this a challenging problem. ENWGS uses a Honeywell MULTICS mainframe to execute simulation models and up to 44 80286-based workstations for accepting user inputs, map displays, and status board reports. The user input forms and the message protocol database, used to transmit form data, reside on the workstations. Though good tools exist to help system developers and maintainers modify these databases, the process is too involved for

TCD writers. A system was required that would automatically ensure that each TCD would get the correct form, and that the form would continue to be correct upon subsequent modification of the TCD.

The Form Generator has two parts: the generator itself and a form downloader used to transmit forms to the workstations. The generator is executed when the TCD is entered into a library. It creates an abstract form that contains all of the information needed by the workstation to produce an actual form and data transaction. The form download software is run whenever a game participant logs into the simulation. An additional performance feature of the download mechanism queries the workstation to find out what TCDs are resident. Only those TCD forms whose correct versions do not already reside on the workstation are downloaded.

The TCD definition language, library, and form generator combine to produce an integrated environment that allows the definition of new system commands without any knowledge of the simulator's underlying software architecture.

USER CONSIDERATIONS AND THE TCD LANGUAGE

The TCD language (TCDL) was designed explicitly for the definition of Tactical Control Directives. During the specification process all language constructs were scrutinized for relevance to the functional requirements and ease of use by the TCD writers. TCDL is a rule-based or production language that also incorporates features from more conventional procedural languages.

The following section describes a number of the language features and explains how user considerations affected their design. (See Example 1)

Each TCD contains Form Description Data that is used by the form generator to create game environment data input forms. This data is also used to generate game play reports. The TCD writer can specify form directions, summary text, a four letter abbreviation, and prompt text and default values for each input parameter. Prompt text, initial values, and abbreviations will be given default values by the compiler if not supplied. Specifying form description data within the TCD text centralizes modifications. All changes to a TCD, whether to modify program functionality, to change the appearance of the input form, or to modify report text, are made by modifying the TCD program and reentering it into the library.

Strong Typing is an unusual feature for a rule-based language, but is used extensively by TCDs. The TCD writer must supply a data type for each input parameter and local variable. Some data types parallel those found in other programming languages: integer, boolean, and string. However, the majority of types such as latitude, longitude, altitude, actor track (those tracks under my command), any track (those tracks I have detected and do not control), and weapon name are specific to war gaming. Data types are use for static and runtime error detection. They are also used by the form generator to determine the length and transmission type of each form parameter, saving the TCD writer from having to supply these low level details.

Input Parameters are listed within the first statement of a TCD. The order of the parameters in this list determine the order in which they will appear on the input form. Each parameter's data type, and optional prompt text and default value, is supplied later in the TCD program.

Local variables behave much like their counterparts in conventional programming languages, allocating one storage location for each variable. The developers debated at length on whether to support multiple instantiation of variables as is more common in AI languages. Finally, it was determined that this would be too alien a concept for programmers whose prior experiences focused exclusively on procedural languages.

The TCD Rule Structure was inspired by the language OPS5 (Brownston 1985): each rule contains a set of left hand side (LHS) conditions and a set of right hand side (RHS) actions. On every inference engine cycle each rule's LHS conditions are tested. If a rule's LHS evaluates to true, the rule is marked eligible to fire. If more than one rule is eligible on any given cycle, the inference engine will determine which rule is most appropriate to fire.

The TCD rule syntax has been augmented with Situation and Action text. The Situation text is associated with the rule's LHS and the action text with the RHS. The text is used to document the functionality of each rule. Building the rule documentation into the language syntax has several advantages: it promotes self-documenting code, it melds the knowledge acquisition and coding phases, and it can be used to produce static and runtime summary reports with a natural language flavor.

TCDs support four different Rule Types: Single-fire rules, Multiple-fire rules, Action-only rules, and Validation rules. Single-fire rules will only fire one time. After its first firing, the Inference Engine will no longer evaluate the LHS of a Single-fire rule. Multiple-fire rules, on the other hand, will fire each time a new set of game circumstances match the LHS. Action-only rules have no LHS, their only precondition is that the TCD has begun execution. Each TCD is allowed one Action-only rule, which is fired immediately after TCD initialization. Validation rules are used to validate TCD input parameters. The only action allowed in a Validation rule is to call a service routine that will redisplay the user input form along with an error message. Validation rules are used to write TCD specific error checks. For example, if a particular TCD is intended for use with air tracks only, the TCD can test that a particular input parameters is, in fact, an aircraft.

Substantial effort went into developing a simple but effective Inference Strategy for TCDs. The developers considered using criteria like recency of events and priority of condition types. Both these criteria were dropped because of unanswered important questions such as: is it more important to process an old detection or a new one? and, is a hostile detection more important than an indication of low fuel? The inference strategy finally agreed upon was based on specificity. A rule with more conditions has greater priority than a rule with less. If two rules have an equal number of conditions, the order of the rules in the program is used. This strategy has two advantages: it is easy to explain and the relative priority of rules can be calculated when the TCD is compiled. Calculating the rule priority at compile time allows use of a much more efficient runtime conflict resolution algorithm.

TCD Primitives are used in the RHS of rules to request that ENWGS perform some action. These primitives are patterned in name, functionality and arguments after the player commands used during gaming. A parameter list is used to supply arguments to the TCD Primitives. The data type of each argument is tested for compatibility at compile time, with the goal of decreasing runtime errors.

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

 directions:
  "The TCD air_engage is used for air_to_air";
  "and air-to-surface engagements.  The system";
  "will automaticlly choose the appropriate"
  "weapon";

 summary:
  "The TCD air_engage is used for air-to-air";
  "and air-to-surface engagements.  The TCD will";
  "recover the interceptor when weapons are low,";
  "or when fuel is low."

 keyword: "ZAIR";

 parameter interceptor act_trk;
     prompt: "INTERCEPTOR";

 parameter roe boolean;
     init: "Y";

 parameter target any_trk;
     prompt: "TARGET";

 parameter base base_cmd;
     prompt: "RETURN BASE";


vrule: validate_interceptor;
situation: "Interceptor is not an air track";
    track_type (interceptor) ^= "air";
action: "Send an error message";
    send_error_msg (interceptor,"must be air
    track");
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;

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

srule: 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;

srule: 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;

end_tcd;
```

Example 1. Example TCD

View Functions are analogous to the player's ability to report dynamic game information. They allow access to the ENWGS game database for items such as: fuel level, location, track type, current game time, and maximum speed. There are also functions to perform simple calculations (e.g., addition, multiplication, or the distance between two tracks). View Functions can be used in the LHS or RHS of rules. Each returns a value of a specific data type that can be tested in a rule's LHS or used as a parameter to TCD Primitives, Wait Functions, or other View Functions.

Wait Functions return a boolean value and can only be used in the LHS of rules. They test whether certain conditions have taken place in the game. Wait Functions emulate a set of conditional events that the game player can schedule, in the same way as the TCD Primitives emulate player commands. The Wait Function syntax is similar to View Functions; however, their implementation is completely different. Wait Functions are triggered by asynchronous game events; such as, receiving a detection, completing an intercept, or running low on fuel. It is expected that Wait Functions will be the dominant construct used in the LHS of rules, with View Functions being used to test very specific conditions beyond the ability of the Waits. Because Wait Functions are an implementation of an interrupt handler, they are much more efficient than View Functions which poll the system for information.

The Model Outcome construct is used to pass data specific to Wait Function firing into the TCD. For example, if a detection Wait Function fires, it is very likely that the player intends to take some action against the detected track. This construct corresponds to the player's ability to observe game events.

To promote reuse of existing TCDs, it is possible to Invoke one TCD from another. Invoked TCDs execute in parallel with their parent. When compiling a program with an Invoke statement, the TCD being invoked must already exist in a TCD Library. The developers have also specified a Rule Block construct. Rule Blocks, when implemented, will allow the TCD writer to group rules into internal subroutines that will restrict other rules in the TCD from firing until the Rule Block completes.

A TCD is allowed to Terminate itself or any other TCD that it has invoked. In this way the TCD can return platforms to player control when their TCD mission is complete. Future TCD implementations may include an On-terminate Wait Function and simple message passing to report termination reasons.

Bags are a simple data structure that can be used to store collections of homogeneous items. The initial use of Bags will be to select items from the Bag based on certain selection criteria; for example, select from a Bag of interceptor aircraft the one closest to an incoming hostile detection. Future implementations may allow Bags as arguments to Wait Functions. A useful application of this construct is to schedule a Wait Function that will return the name of any interceptor track that has run low on fuel.

These language constructs provide the TCD writer with building blocks for emulating the decision making process normally performed by a game player. The developers believe that the language can be used effectively by the targeted users: persons with a good understanding of the ENWGS simulator but limited programming background.

USER CONSIDERATIONS AND THE RUNTIME ENVIRONMENT

The runtime environment is used to execute TCDs during ENWGS games. It consists of the TCD user interface (as seen by the game participant), the Inference Engine, and various interfaces with the existing simulator. This section will focus on the user interface issues related to the runtime environment.

Requirements for the TCD game play interface were established by the user interface standards for the rest of the system. ENWGS is a form-driven system whose forms can be reached by entering a command's name or four letter abbreviation. The system also supports a set of selection menus for those less familiar with the system. ENWGS forms begin with direction text and are followed by data input fields. The player can request help text for each input field. If a player transmits a form containing invalid data, the form is redisplayed with an appropriate error message. Each data field in error is also displayed in red. The TCD forms which were automatically generated in the development environment support all of these interface conventions.

Input data validation is critical to TCD processing to ensure that when a rule fires the resulting actions are supplied with correct data. TCD input data is validated at three levels. First, simple data format errors are detected by the workstation software. For example, the software can test that a latitude field includes a North/South indicator. Format information is determined from the data type of each TCD parameter. The second level of validation, performed by the host computer, tests whether the input data itself is correct. Each input parameter is validated with respect to it's TCD data type. This level can test, for example, whether a given platform is subordinate to a player. The final level of validation is specific to the TCD and is implemented using validation rules. The goal of these three levels of data validation is to detect incorrect data before the TCD is loaded into the Inference Engine. Once inferencing begins, error detection and correction become much more difficult.

Reports in the runtime environment were designed for use by game players with no TCDL background. The reports fall into two basic categories: static and dynamic. The static reports provide summary information on the TCD library and individual TCDs. The dynamic reports provide information on each instance of TCD execution. They reports input parameters, the execution progress of each TCD, and the tracks and participants associated with the TCDs. The TCD progress report is especially informative. As each TCD is processed, a record is kept of the rules executed and the time of execution. The report lists the situation and action text associated with each rule fired and each rule pending in the conflict set. This provides the game participant with a "natural language" summary of each TCD mission's progress, without requiring any knowledge of TCDL syntax.

SUMMARY

Shifting requirements are a fundamental aspect of some computer applications. Instead of attempting to solidify these requirements, Tactical Control Directives address the problem by providing the users of an existing simulator with a high level programming environment. The primitive syntactical constructs of the TCD Language map to the commands, conditions, and data observations normally made by system users. The user is provided with a tool for producing new system commands by combining familiar components. These new commands can emulate decision making processes that would normally require human interaction with the system.

ACKNOWLEDGEMENT

We wish to acknowledge the substantial contributions of our co-workers: Dr. David Slater, Dr. Joseph Sowers, Mark Blais, and Denise Roberts.

Work reported in this paper was developed for the Department of the Navy, Space and Naval Warfare Systems Command, under contract no. N00039-84-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.

Fairley, R. 1985. Software Engineering Concepts. McGraw-Hill, NY, NY.

Royce, W. 1970. "Managing the Development of Large Software Systems." In Proceedings, IEEE WESCON., pages 1-9. Reprinted in Proceedings, 9th International Conference on Software Engineering (Monterey, CA, March 30 - April 3, 1987). IEEE, Piscataway, NJ, 328-338.

Rubin, P.E.; Buser J.F. 1988. "Development of an Expert System Environment for use with a Wargaming System." In Proceedings, SCS Multiconference (San Diego, CA, Feb. 3-5). SCS, San Diego, CA).

Department of Defense. 1983. Military Standard Software Development. DOD-STD-1679A (Navy). US Government Printing Office, Washington, DC (1983-705-040/5447).