ARI Research Note 91-63

Developing a General Contingency Planner, Phase II

Paul Young
PAR Government Systems Corporation

for

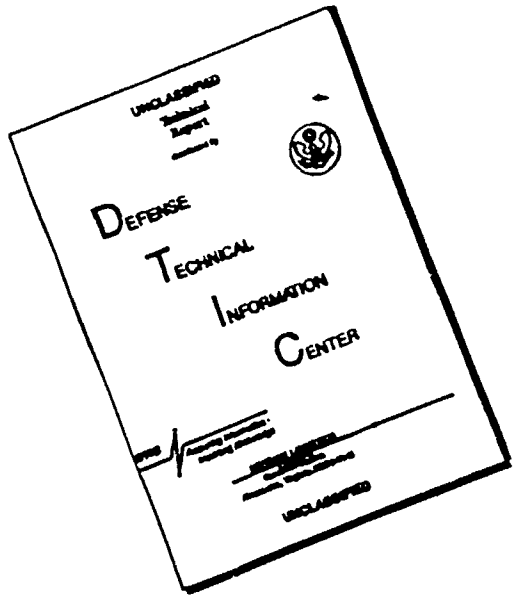
Contracting Officer's Representative
Michael Drillings

Office of Basic Research
Michael Kaplan, Director

June 1991

United States Army
Research Institute for the Behavioral and Social Sciences

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS --
2a. SECURITY CLASSIFICATION AUTHORITY --	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE --	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) --	5. MONITORING ORGANIZATION REPORT NUMBER(S) ARI Research Note 91-63
6a. NAME OF PERFORMING ORGANIZATION PAR Government Systems Corporation 6b. OFFICE SYMBOL (If applicable) --	7a. NAME OF MONITORING ORGANIZATION U.S. Army Research Institute Office of Basic Research
6c. ADDRESS (City, State, and ZIP Code) PAR Technology Park 220 Seneca Turnpike New Hartford, NY 13413	7b. ADDRESS (City, State, and ZIP Code) 5001 Eisenhower Avenue Alexandria, VA 22333-5600
8a. NAME OF FUNDING/SPONSORING ORGANIZATION U.S. Army Research Institute for the Behavioral and Social Sciences 8b. OFFICE SYMBOL (If applicable) PERI-BR	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA903-85-C-0106
8c. ADDRESS (City, State, and ZIP Code) Office of Basic Research 5001 Eisenhower Avenue Alexandria, VA 22333-5600	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO.: 61102B PROJECT NO.: 74F TASK NO.: N/A WORK UNIT ACCESSION NO.: N/A

11. TITLE (Include Security Classification)
Developing a General Contingency Planner, Phase II

12. PERSONAL AUTHOR(S)
Young, Paul

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 85/04 TO 87/04	1991, June	121

16. SUPPLEMENTARY NOTATION
Michael Drillings, Contracting Officer's Representative

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Artificial intelligence Problem solving Army maneuvers

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

This report summarizes the work performed in the final phase of a 3-year effort to investigate basic mechanisms for solving planning problems in domains characterized by adversity. In the first phase of the effort, a general purpose planning mechanism was developed. This planner represented plans in a manner consistent with the goal tree formalism characteristic of Artificial Intelligence action planning research and used plan generation/search techniques derived from both AI action planning and knowledge-based game-playing theory. In the second phase of the effort, advanced planning mechanism was also extended from its initial domain (the two-player boardgame of Othello) to an Army maneuver planning domain.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Michael Drillings	22b. TELEPHONE (Include Area Code) (703) 274-8722 22c. OFFICE SYMBOL PERI-BR

DD Form 1473, JUN 86 Previous editions are obsolete.

U.S. ARMY RESEARCH INSTITUTE FOR THE BEHAVIORAL AND SOCIAL SCIENCES

A Field Operating Agency Under the Jurisdiction of the Deputy Chief of Staff for Personnel

EDGAR M. JOHNSON
Technical Director

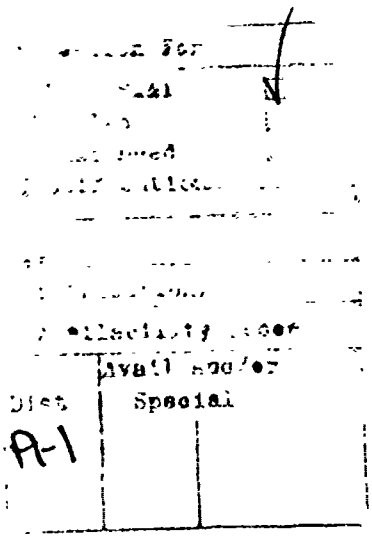
JON W. BLADES
COL, IN
Commanding

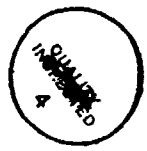
Research accomplished under contract
for the Department of the Army

PAR Government Systems Corporation

Technical review by

Michael Drillings





NOTICES

DISTRIBUTION: This report has been cleared for release to the Defense Technical Information Center (DTIC) to comply with regulatory requirements. It has been given no primary distribution other than to DTIC and will be available only through DTIC or the National Technical Information Service (NTIS).

FINAL DISPOSITION: This report may be destroyed when it is no longer needed. Please do not return it to the U.S. Army Research Institute for the Behavioral and Social Sciences

NOTE: The views, opinions, and findings in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other authorized documents.

DEVELOPING A GENERAL CONTINGENCY PLANNER, PHASE II

CONTENTS

1. INTRODUCTION

A good deal of the research conducted in Artificial Intelligence (AI) over the last thirty years has focused on the development of systems for generating plans of action for agents faced with numerous, complex, and conflicting goals. Because planning is such a vital part of the military function, the promise of these systems is of great interest to the military community. Foremost among the contributions that could be made by reliable AI-based planning systems would be 1) the ability to monitor complex situations where large quantities of data need to be assimilated quickly, and 2) the ability to project outcomes of possible courses of maneuver in dynamically evolving battlefield environments.

This report discusses the problem of extending the action-planning techniques developed in Artificial Intelligence research to problems involving the need to plan against an intelligent adversary. The focus of the report is work completed by PAR Government System Corporation (PGSC) during the final phase of a three-year basic research effort. The remainder of this section of the report presents a general discussion of the nature of adversity and the special considerations which must be taken into account by any automated planning system which is to operate in a domain characterized by adversity. Section 2 explains the basic representational structure at the heart of the planner developed on this effort: Contingency Goal Trees. Section 3 explains the generation of these trees. Section 4 focuses on the representation of planning knowledge in the form of goal elements and how these are structured syntactically. Sections 5 and 6 provide two illustrative examples of how the planning system functions. Section 5 is an example of the planner's operation in the two-player board game of Othello, which was the domain of interest during the first phase of work. Section 6 presents a planning scenario which is oriented toward army maneuver planning. This was the domain of interest during the second phase. A summary of the entire effort, along with assessments of the degree of success of the work, is given in Section 7, and a complete listing of the source code for the planning system is provided in the Appendix.

The ability to act for an end has long been used to characterize the special nature of human beings. Aristotle was among the first to recognize this fact, which has served as a definition of intelligent activity ever since. If computerized agents are to be

successful and truly useful in a complex world, ways must be found that will allow them to copy, or closely approximate, the planning skills of men.

The attainment of a goal does not happen automatically. It is not enough for an agent to simply have a goal in mind: an effort is required to accomplish the goal. To understand this we need only attend to the fact that the world or environment that confronts any agent, either human or not, is one that is full of obdurate and self-insistent things that oppose the agent. Among these are other agents with their own goals, objects like rocks and trees that can get in the way, and roads with forks in them. Each of these can oppose the agent in the attainment of its ends. The collection of stubborn realities which can impede an agent can be seen as forming a general condition of adversity which permeates every environment or domain where an agent would attain a goal. Adversity arises in all domains no matter how simple or complex, and it is this condition of adversity that separates every agent from its goal. The central issue in automated planning is then the problem of dealing with adversity -- adversity created by the stubbornness of whatever is the non-agent.

In a sense, adversity has always been recognized as the impediment to action planning by researchers in this field. In every case, however, the general problem of adversity has remained hidden from view and unattended to as a result of the attention given to the special form of adversity in a given domain. Examples of this kind of oversight can be seen in the kinds of problems usually dealt with in automatic problem-solving research and the way in which these problems are approached.

Usually, a problem-solving program will be centered around the solution of a single problem (e.g., the familiar blocks world) or a family of related problems. In most of these treatments, the result has been, at best, a problem solver which is successful only in the realm of the class of problems under consideration. This limitedness arises from taking a particular problem or problem set as representative of problem solving in general and then building a solution for that particular instance. Created are a multitude of problems solvers, each of which may be fairly successful in its own restricted domain, but is of little value in a general sense. In each of these approaches, there are special devices for constraining search among alternatives, resolving conflicts among alternatives, and resolving conflicts among competing subgoals. What has been missed by a good deal of the previous research in automatic problem solving is that all action planning shares a common feature -- it is

only necessary because of the general condition of adversity which confronts any agent acting for an end. If the other features of the world (the non-agent) were incapable of resisting and opposing the agent, there would be no need for planning, subgoals, or any other effort: all ends would be achieved instantaneously by the power of the agent's will. However, because the real world has an integrity and status apart from the agent, the agent is rigorously opposed, and the collection of oppositions with which the agent is confronted can be seen as a condition of adversity. Once this commonality has been discovered, it becomes possible to propose a model or paradigm for problem solving in general and thereby arrive at a truly domain-independent problem solver.

A simple example will serve to demonstrate that adversity is a common condition for all problem solving activity. Consider the planning problem depicted in Figure 1-1. Here, the goal is to capture the city with the given units which are initially on the wrong side of the river. The objective can be expressed in the form of a single-node plan (utilizing Sacerdoti's procedural network formalism) as shown in Figure 1-2.

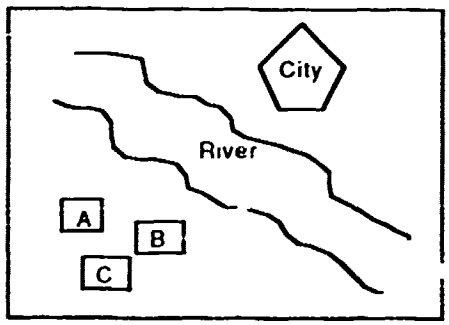


Figure 1-1

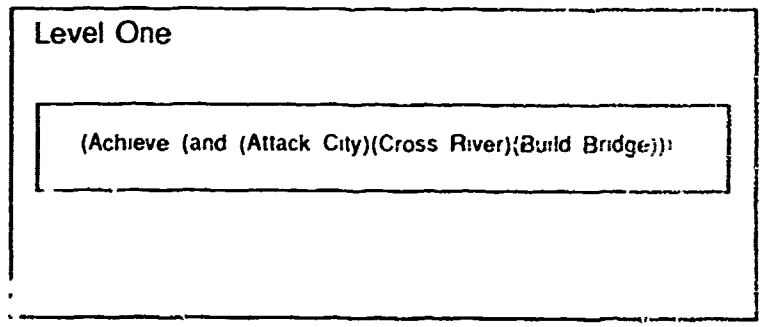


Figure 1-2

This single node can be expanded into a more detailed plan by breaking the conjunction into its components. The resulting decomposition is shown in Figure 1-3. The three subproblems are shown in parallel in order to indicate that there is no a priori commitment to a temporal sequence. Obviously, however, for the goal to be achieved, the subproblems must be solved in a particular order. In this case, the city cannot be attacked until the units are across the river, and the units cannot get across the river until the bridge has been built. Such an interaction between subproblems has been termed a "conflict" by Sacerdoti. Conflicts are resolved by the application of a device known as a "critic," which looks for special kinds of interaction in a developing

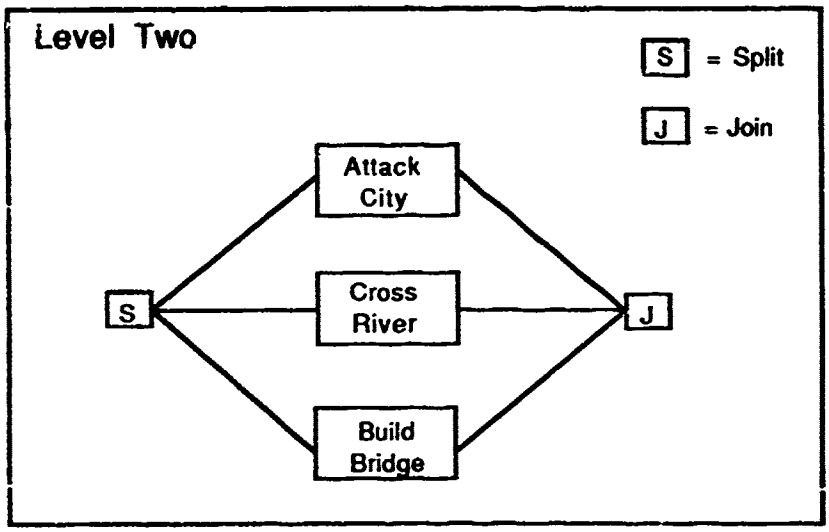


Figure 1-3

plan and adjusts the plan accordingly. Critics, because they are intended to be applicable to multiple planning situations, are necessarily pre-defined.

The conflicts in the simple plan developed so far would be resolved by a critic (or critics) that knew that a precondition for being near something on the other side of a river would be the use/construction of a bridge to get across. After resolution of the conflicts, the CAPTURE_CITY plan would look as shown in Figure 1-4.

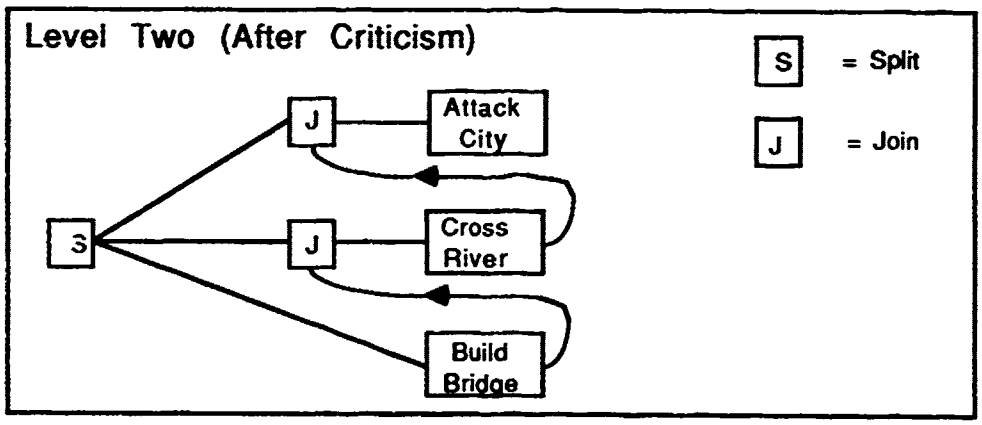


Figure 1-4

A close look at the initial problem (Figure 1-1) and the current plan (Figure 1-4) shows that there is still some ambiguity in that there are three distinct units to be moved across the river. Therefore, the plan can be further decomposed into a new set of subproblems. Each of these new subproblems would represent a step for moving each of the three units across the river. The new, more-complete plan is shown in Figure 1-5.

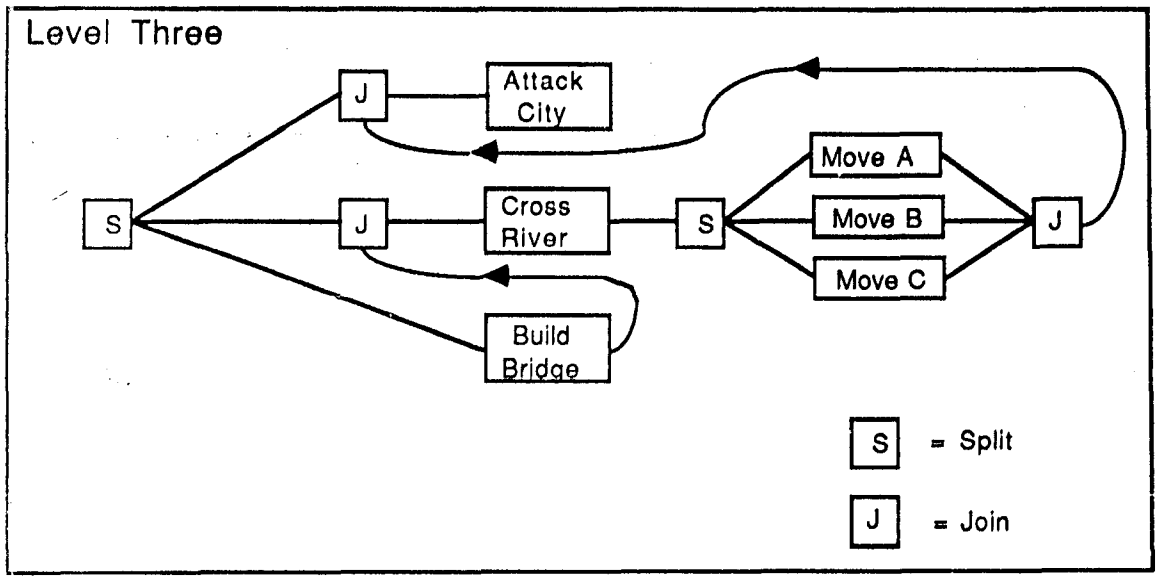


Figure 1-5

Once again, the subproblems are shown in parallel, and, as long as the three units represent the same or similar resources, this plan is fine. In a real military-planning situation, however, unit A may be artillery, unit B infantry, and unit C armor. In this case, the order of sending the units across may make a great deal of difference to the success of the operation. If the infantry crosses first, it will be subjected to the full fury of the enemy's resistance, while the armor (unit C) will be following uselessly behind. There is, therefore, a new conflict among subproblems in the plan, and this conflict will have to be resolved by yet another critic. One possible criticism might be represented by a rule which holds that armor always precedes infantry and that the artillery follows the infantry. Another possibility might be a critic which knows that, within a certain range, the artillery could be left on the far side of the river and fire on the city from there. A simple resolution to the conflict is shown by the new plan in Figure 1-6.

Conflict resolution forms another primary problem in the development of automatic-planning systems from the standpoint of Command and Control problems. Notice that both resolutions in the river-crossing example were achieved by the addition of new knowledge into the planning process: specifically, knowledge about the sequencing of actions with respect to time for attacking the city and how to use different kinds of resources in that attack. The way this knowledge is usually handled is in the form of special general-purpose rules. These rules, by virtue of the fact that they must be pre-defined, can be described as static, that is, they cannot take any account of factors or conditions external to the explicit ones embodied in their own

logic. The rule, or critic, looks for particular interdependencies and resolves these conflicts in some prescribed manner. As such, it has no capability to reason about the particular situation at hand and what the implications of the application of the general criticism are for that special circumstance.

The following three sections discuss in depth PGSC's approach to meeting the challenges presented by planning in adversarial domains. First, Section 2 discusses a framework, known as Contingency Trees, for representing adversarial plans of action. Section 3 discusses in detail the planning mechanism which is utilized to generate these trees, and Section 4 presents the knowledge-representational scheme for representing the actual goal knowledge.

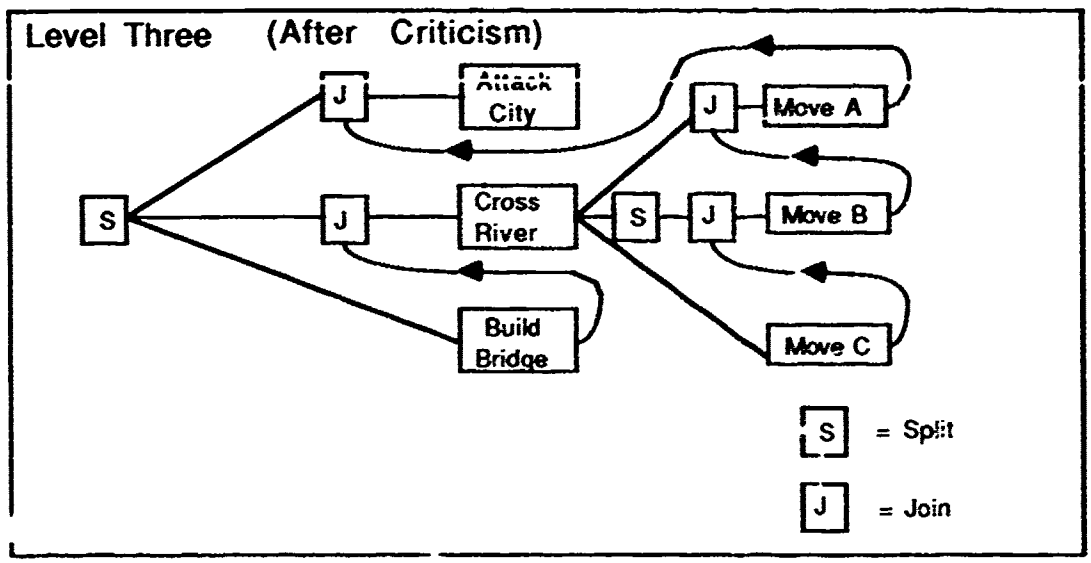


Figure 1-6

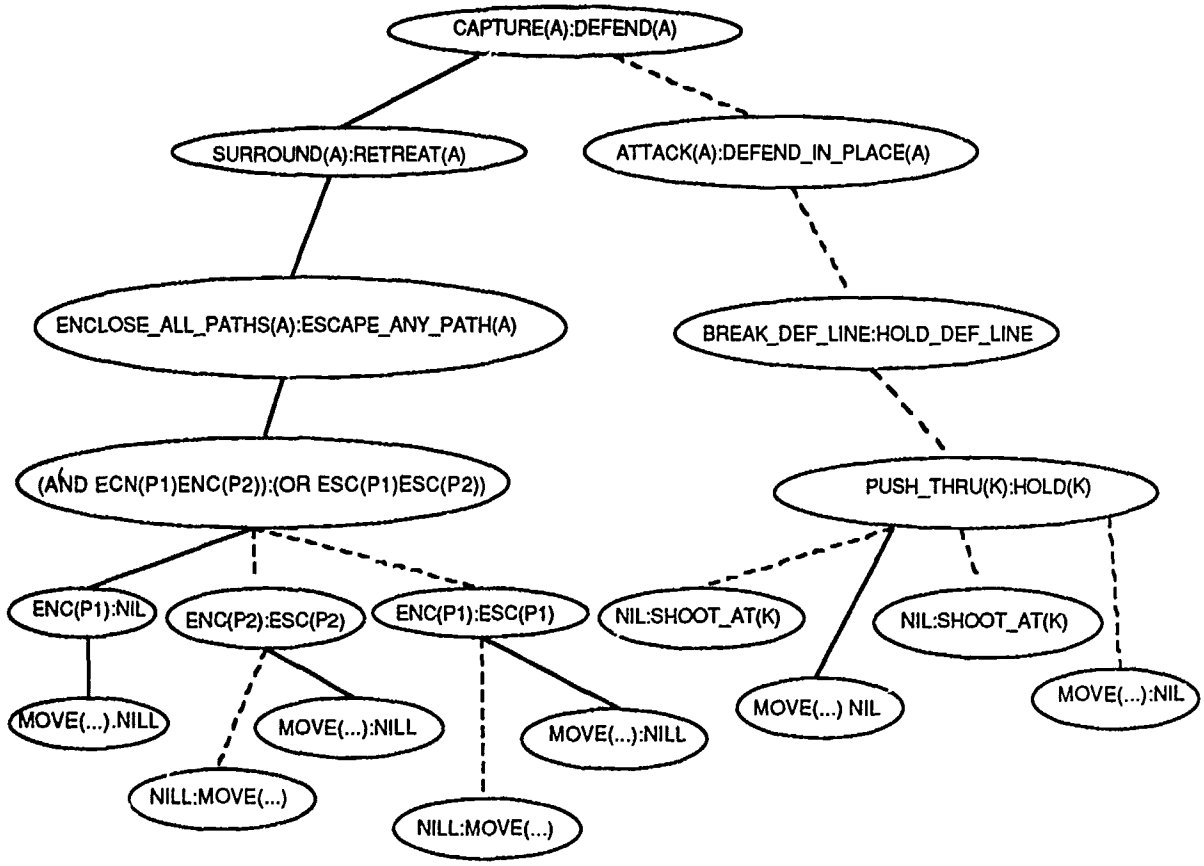
2. THE STRUCTURE OF CONTINGENCY GOAL TREES

The planning system which PAR Government Systems Corporation (PGSC) developed represents plans as contingency goal trees (CGTs). Contingency goal trees are a hybrid plan-representation structure embodying aspects of traditional hierarchical plan representations and of move trees from computer game playing. Figure 2-1 is a sample CGT with 19 nodes. The 19 nodes of the CGT are organized into six levels of abstraction. Abstraction refers to the need in planning systems to represent problems in different degrees of resolution or generality. What is intended by the concept of abstraction can be seen in terms of the frequently used "pasta-making"example. One might want to create a plan for making pasta, which can be represented as the goal: (MAKE_PASTA). This goal can be thought of as having two sub-goals: (BOIL_WATER) and (PLACE_PASTA_IN_WATER). These two subgoals are said to be at a lower level of abstraction because, while they represent an equivalent notion to the initial goal, the problem is nonetheless represented in a more-detailed, specific, and therefore less-abstract way. Similarly, the MAKE_PASTA goal may be less abstract than an even-higher level-goal: HAVE_DINNER, for example.

Abstraction is a key idea in planning because it allows for multiple views of the same problem, each of which is at a different level of detail. Details are usually numerous, hard to keep track of, and not always important. Therefore, the ability to represent plans at more than one level of detail or abstraction allows for the convenient collection of specific parts of planning knowledge under broader concepts which may be more easily manipulated and traced.

The lowest level of abstraction in any computer plan represents what may be called "acts" in the world of the planning system. The purpose of an automated planner is to develop plans for achieving some goal. Regardless of how many levels of abstraction the planning problem may be decomposed into, the plan must terminate in nodes which are not subgoals requiring further development, but rather actions for implementing the plan in the pre-defined world of the planning system. Thus, a robot planner may have numerous goal nodes ranging across multiple levels of problem abstraction (e.g., (OPEN_DOOR DOOR1)). In the end, however, the plan must be made applicable to the world in which the planner is to operate. Applicability is achieved through the generation of bottom-level nodes which correspond to acts in the planner's world.

A planner for developing a sequence for piling or unpiling a set of blocks would have acts corresponding to the movement of particular blocks, while a planner designed to maneuver a fighter jet in a dog fight would have acts corresponding to aileron adjustment and so on. Above each act is a network of goals which are the ancestors of the act and, in a sense, explain why the act is being recommended.



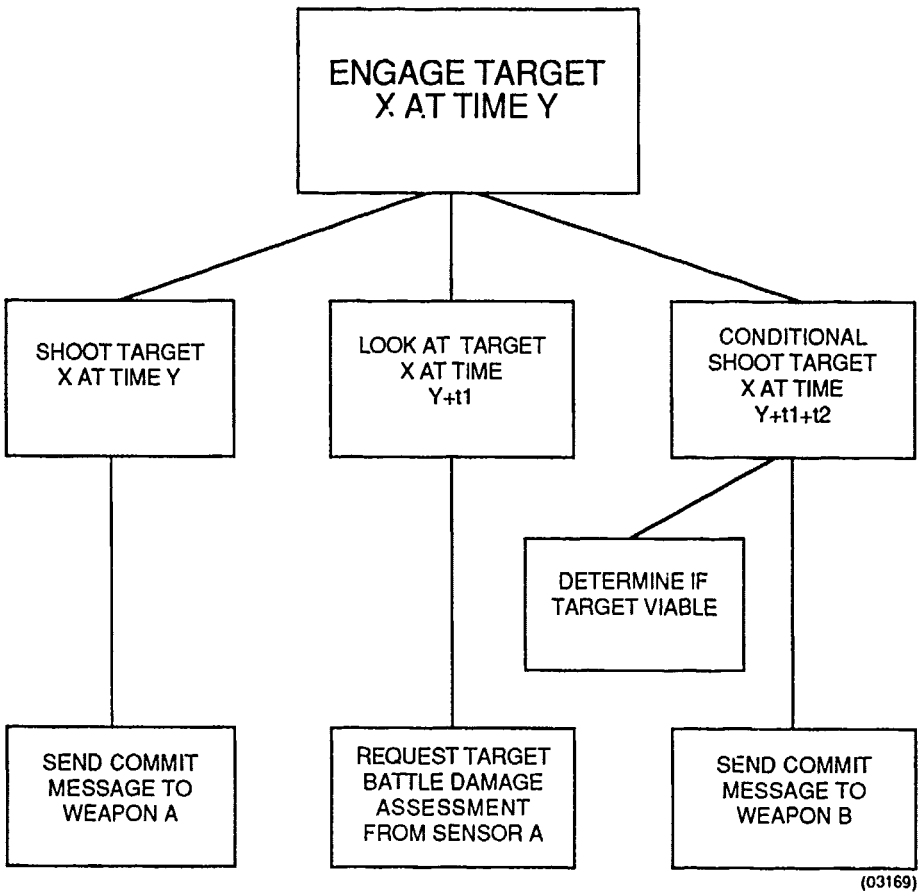
LEGEND

◯	GOAL:COUNTERGOAL PAIR	A	TARGET
(AND	CONJUNCTION	P1	PATH
(OR	DISJUNCTION	P2	PATH
ENC	ENCLOSURE	K	POINT ON DEFENSIVE LINE
ESC	ESCAPE		

(03168)

Sample CGT with 19 Nodes
Figure 2-1

To illustrate, consider the representation depicted in Figure 2-2. This figure shows a hierarchically decomposed solution to the high-level problem "Engage Target X at Time Y." In this extremely simplified example, there are three levels of abstraction. The first, or top level, states the initial problem. This is, in turn, resolved into a three-step plan at a lower level of abstraction. At this more concrete level, an explicit strategy ("SHOOT_LOOK_SHOOT") for achieving the higher-level goal is instantiated. Finally, at the lowest level of abstraction are the acts or discrete steps required to perform the three steps shown in Level 2.



Example of Hierarchical Decomposition
Figure 2-2

It is important to recognize that the question "to what level of decomposition or abstraction a problem should be reduced" will vary with the aims of the particular planning system. That is, there is no pre-defined lowest (or highest) target level. Instead, these will be defined within the context of the given system. It would have been possible, for example, to build a planner which immediately instantiates the goal "Engage Target X at Time Y" into some solution. In that case, the "Engage Target" goal would also be considered an act. As one pushes the level of acts further and further away from the level of the initial problem state, thereby creating more levels of abstraction, the planner is made more flexible and more able to apply to general situations. Thus, the initial representation (Figure 2-2) has four acts, each of which is related to a procedure for instantiating that act. This is in contrast to the situation that would hold if there were only a single procedure for achieving "Engage Target X at Time Y." The former is more flexible, because the four acts with their associated four procedures can be combined in numerous ways to solve the problem differently or even to solve completely different problems.

To address planning problems that involve an intelligent adversary, this research effort has taken the approach of generalizing the goal-tree/procedural-network formalisms to represent plans in a manner that transmits incorporating the type of planning that is done in knowledge-based game-playing research. To this basic framework has been added explicit consideration of adversarial countergoals.

The basic premise of the research effort was that most of the previous work in AI planning, such as that derived from robot problem solving, cannot be readily applied to military-planning problems. Specifically, these planners lack a satisfactory capability to explicitly incorporate an adversary's goals and actions into the planning process. Consequently, they cannot effectively plan against an adversary that is simultaneously planning against them. As an example, consider the case of planning under conditions of uncertainty. One accepted technique is to include information-seeking goals in the plan whenever information necessary to complete a plan is not initially available to the planner (e.g., FIND_LOCATION (X), if the location of X is not known). When an adversary is present, however, that adversary is likely to have a countergoal of preventing the collection of the required information. Consequently, this adversary will use various tactics, perhaps including deception, to prevent information collection. Unless the adversary's countergoals and actions are explicitly taken into account and planned against, the original information goal is not likely to be achieved.

The planner which this research developed, ARES (Adversarial REasoning System), represents plans in a structure called a Contingency Goal Tree (CGT). Figure 2-1 is a sample CGT showing a plan for some hypothetical battle situation. At each node in a CGT is a Goal Pair (GP) which includes a proposed friendly goal and a possible countergoal for the adversary. Sub-nodes of GPs are Sub-Goal Pairs (SGPs). Reading the left-most branch of the example CGT, a sub-goal of CAPTURE (A) is SURROUND (A), while the corresponding sub-goal of DEFEND (A) is RETREAT (A). Consequently, the GP CAPTURE (A) : DEFEND (A) has as its first SGP SURROUND (A) : RETREAT (A).

Simultaneous multiple tasks are treated as conjunctive (AND) or disjunctive (OR) goals. Thus, (AND ENC(P1) ENC(P2)) is a single goal of enclosing both escape path 1 and escape path 2, while (OR ESC(P1) ESC(P2)) is a goal of escaping through either path 1 or path 2. Other "generic" goals, such as AND_IN_SEQUENCE, are possible.

In many instances, GPs will include a goal or countergoal of NIL. This occurs when there is no projected opposing goal, such as when ARES is exploring an option where one side pursues an independent course of action while ignoring the adversary's goals. It also happens to occur in example CGT because this example depicts a linear sequence of moves and countermoves, as would be found in alternating-move games such as Chess.

When a CGT contains a NIL element in every GP, it is isomorphic to a standard goal tree. Also, CGTs can have parallel branches and, if desired, pre-conditions and post-conditions attached to nodes. Consequently, CGTs represent a straightforward extension of the goal-tree/procedural-network representation.

3. GOAL-COUNTERGOAL PAIRING: CGT GENERATION

Section 2 described the means of representing plans in the planner PAR Government Systems Corporation (PGSC) developed. This section deals with the process used to generate contingency goal trees.

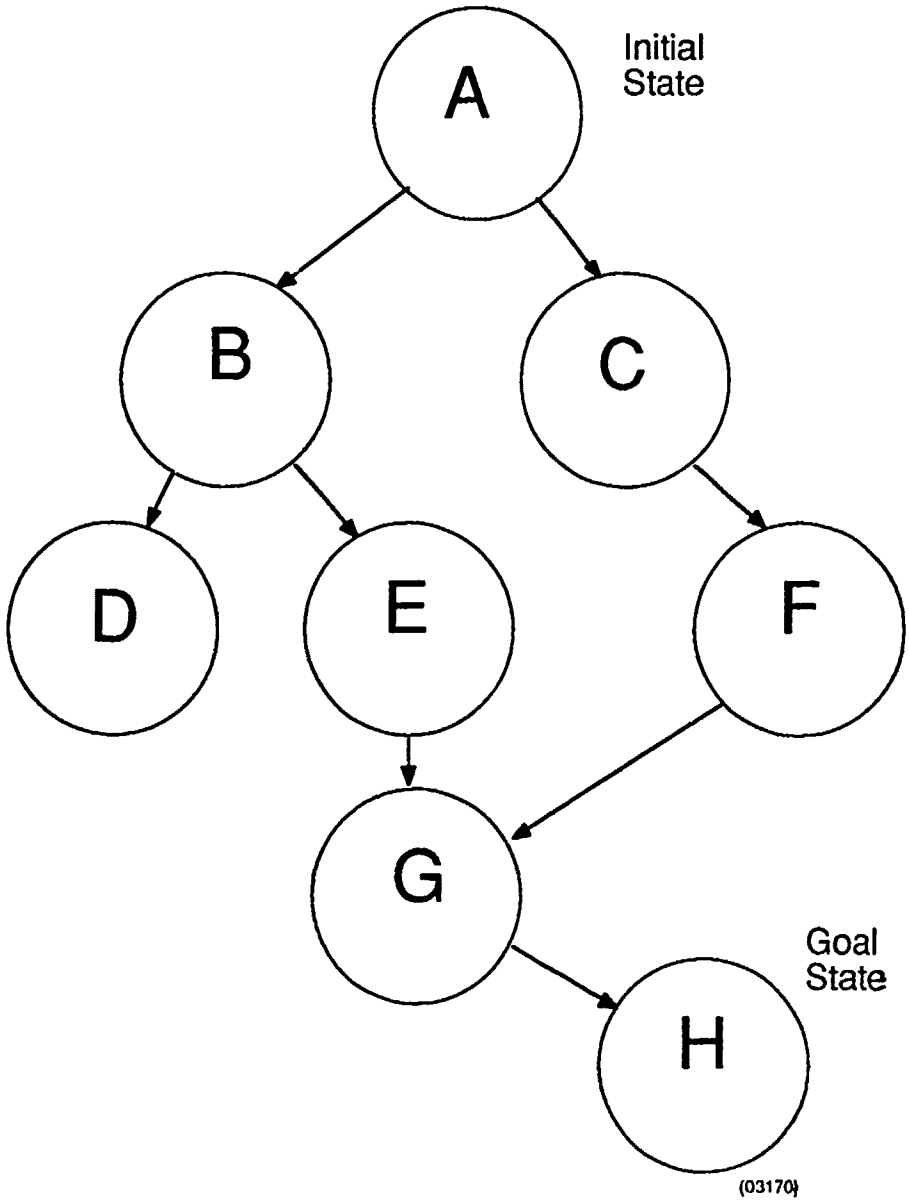
Given that plans of action are to be represented in the form of hierarchically decomposed networks where higher-level abstract goals are resolved into more detailed, less-abstract component parts, the next question pertains to what mechanism is to be used to accomplish the actual reduction of one level of goal into its more detailed components. Given some initial problem, an automated planner requires some means of generating the network of goals and subgoals that represent the solution.

The plan-network-generation problem is in a class known as state-graph search problems, where, given an initial state, an exploration of possible sequences of future states is performed in the hope of isolating a path that will lead to a goal state. Figure 3-1 shows a state graph for some imaginary problem. In this example, Node A represents the initial state and Node H the desired goal state. The problem is to find a path that leads from A to H.

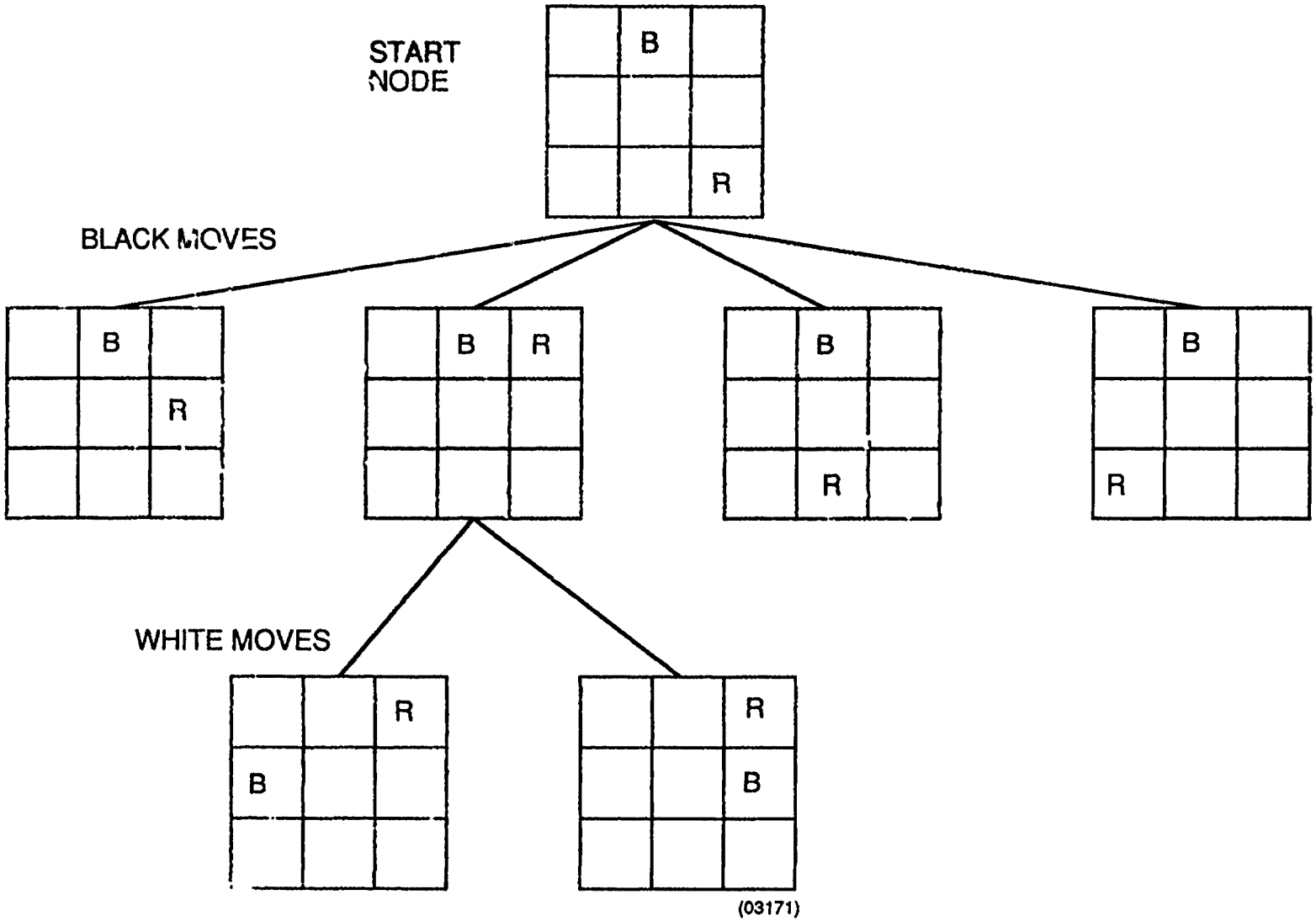
Another related form of this representation is the commonly known move tree from computer game playing. In a move tree, a particular node represents some legal arrangement of playing pieces on the board. The "children" of that given node will be the complete set of possible legal moves for the side on the move. Each of these nodes becomes, in turn, the starting point for proposing a set of subsequent moves for the opposite side. Figure 3-2 shows a sample move tree for a very simple Chess game where there are nine squares, a White Bishop, and a Black Rook. Reading this tree from the top down, first all the possible moves are found for the Black Rook. At the next level, each of the possible White responses to the Black moves is postulated.

In procedural networks, state graphs, and move trees, the basic problem is how to find a path that will lead to the desired state. Numerous alternatives are possible. The first and most obvious means is to exhaustively enumerate all the possibilities. Inevitably this results in a great deal of wasted effort because usually only a relatively

few paths will lead to success. The expansion also falls victim to a combinatorial explosion where the search space grows exponentially.



Sample State Graph
Figure 3-1



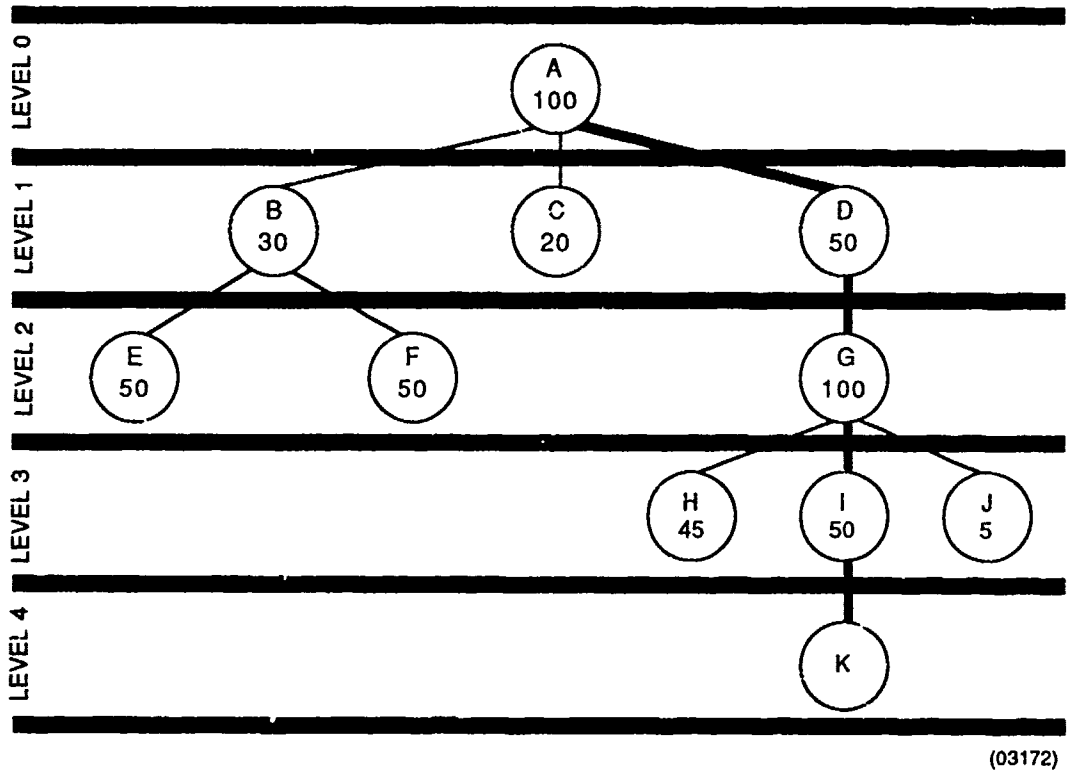
Sample Move Tree For Chess
Figure 3-2

There are numerous possible ways to cut down the amount of computational effort required to navigate through the network. Breadth-first Search examines all the elements at each level (horizontally) before proceeding depth wise to the next level. Depth-first Search expands the state graph vertically until a path either leads to a success or a dead end. If a dead end is reached, the search backs up to the immediate prior node and seeks an alternative at that level, and so on.

In a recent article, deKleer (1985) identifies three principal ways of exploring a search space. The first, and most common, brute-force enumeration of all possibilities, was described above. An alternative approach is chronological backtracking (a variant of depth-first search). However, the discovery of a failure in the developing plan may result in a large amount of wasted work because the real reason for the contradiction in the plan may be some choice further back in the plan, not necessarily the immediately prior one. As deKleer writes: "When a contradiction is discovered the search should backtrack to a choice which contributed to the contradiction, not to the most recent choice." A second alternative to brute-force enumeration is known as dependency-directed backtracking (another variant of depth-first search). In dependency-directed backtracking, records are kept about the dependency relationship that each choice in a plan has with regard to prior choices. When a failure in the plan occurs, dependency records are examined to determine which prior choices (not necessarily the last one) set up this situation.

In actuality, most planners employ a hybrid approach for developing plans of action: neither pure breadth first nor pure depth first. A popular hybrid is known as best-first, where the most promising alternative at each level is explored first.

Figure 3-3 is an example of a best-first expansion. At each level, some procedure is used to select the "best" node for expansion. Once a node is selected within a level, the planner proceeds to evaluate the children of that node seeking to identify the best possible one to expand, in turn. This continues in the same manner as a depth-first search, except that some discrimination is performed at each level among the possible nodes to expand. The expansion results in the location of either a successful path or a dead end. If the path has led to a dead end (a failure), backtracking is undertaken to some prior node, where the expansion begins anew.



Best-Fit Example With Sample Scores
Figure 3-3

The approach taken to planning in ARES is a hybrid of these other techniques. By reading the terminal nodes of a CGT from left to right, a specific sequence of actions is defined. Consequently, each CGT specifies a contingency plan if the adversary pursues the goals indicated in the tree. In an adversarial world, "good planning" means generating contingency plans for each reasonable course of action that either the adversary or the agent may pursue. To this end, ARES attempts to generate a set of CGTs that provides contingency plans for the full scope of reasonable options available to the agent and the adversary.

For a given level of abstraction, ARES proceeds through CGT generation/expansion in a depth-first manner. Beginning with the most recent GP added to the CGT, ARES recursively adds SGPs until a terminal GP is reached. A terminal GP includes a procedure for making an action in the world, which means that it is the lowest level in the expansion. Processing a GP involves accessing a knowledge base of goal definitions in order to determine possible reasonable options to instantiate for either the agent or the adversary. In alternating-move games, these accesses are for the goal for the side that is currently on the move. In more

complicated domains, like maneuver planning, where actions can occur simultaneously, both parts of the GP are processed at the same time.

In the example CGT, the solid line identifies those SGPs that were generated by processing the parent goal of side A, on the left, while dotted lines indicate SGPs that are descended from the parent goal of side B, on the right.

The general procedure employed by ARES in the generation of plans may be characterized as an iterative process of proposing possible courses of action and, based on the outcome of those actions, proposing better action options. ARES, in effect, plays out a series of hypothetical games in order to "learn" about the way objects in the domain interact. The knowledge that is gained is further used in the construction of the network of goals and countergoals.

Table 3-1 presents a high-level, English-language description of the actual procedure used by ARES to generate CGTs. The procedure "RESOLVE" is recursively called until goal decomposition ends with either a failure or an act in the world state.

TABLE 3-1

Procedure **RESOLVE**

1.0 Given an initial input goal pair, designated *gpair*, determine whether it will result in success for either agent or counter agent. Contingency Goal Trees created in this process, taken together, form a plan.

2.0 **IF** a word update action can be performed, **THEN** perform it and **RETURN** *T*.

3.0 Generate a new sub-goal pair (*sub-gpair*) for *gpair* and add to *sublist*

4.0 **CASE:**
sub-gpair : go to 7.0
(**RESOLVE** (*sub-gpair side*))≠NIL: go to 3.0
END{case}.

5.0 Replace most recent *sub-gpair*.

6.0 **IF[AND(** *sub-gpairT)* ***(RESOLVE****(sub-gpair side))]*, **THEN** go to 3.0

7.0 **CASE:**
sublist > 0 **AND** (side effects) : go to 5.0
sublist > 1 : Remove one sub and go to 5.0
sublist = 1 : **RETURN** *T*
sublist = 0 : **RETURN** *NIL*
END {case}.

4. KNOWLEDGE REPRESENTATION IN PLANNING

Computer game playing and robot problem solving are generally considered to represent distinct classes of planning problems. The normal model for examining two-player, perfect-information game problems starts with a game tree which maps the various possible situational evolutions that can occur, given different action options for the two sides. As pointed out in Section 2, for even the simplest games, such trees can develop to unmanageable size after only a few levels, introducing the need for clever techniques for isolating the best move based on only a partial look-ahead.

Several possible methods, including minimax and alpha-beta (see Nilsson, 1980 for a review) exist for constraining the search problem. Other researchers have coupled these techniques with the use of heuristics and the incorporation of domain-specific techniques to limit searching in a given game problem (Ballard, 1982).

Two issues appear to be most important for the creation of a successful planner in any domain: 1) a knowledge-representation strategy and 2) a mechanism for applying knowledge in an attempt to constrain the search space. This report section details the major results of PGSC's investigations with respect to this important aspect of automated planning.

PGSC contends that the classes of problems described generally as computer game playing and automatic problem solving do not represent unrelated domains within AI but are rather special cases of a broader problem set which can be best categorized as dynamic plan generation.

A major impetus behind PGSC research in this area has been the state of two-player, perfect-information games over the last 20 years. This sad situation has been summarized by Berliner (1973) with his criticism that one of the most serious deficiencies in game-playing programs, like those being developed for Chess, was the absence of long-range or global plans for winning the game. Wilkins (1979), with his PARADISE system, provided further guidance by restating the game-playing problem in terms of planning issues. The essence of Wilkins' approach is to develop move sequences based upon the goal-directed development of a plan, instead of picking the "best" move from a tree expanded to some arbitrary depth according to some arbitrary method. The result is that the emphasis in developing successful game-playing

programs is shifted from questions of computational complexity to the construction of effective goal-based knowledge-representation schemes.

The use of long-range planning and goal-based knowledge has been successfully employed in a number of planning mechanisms (see Berliner, 1975; Reitman and Cox, 1979; and Wilkins, 1979). In all of these systems, knowledge is employed in order to constrain a space of available options. Once candidate strategies are identified, thereby eliminating many branches of the move tree, action alternatives can be investigated to considerable depth without suffering from the effects of a combinatorial explosion.

Controlling the generation and/or search of a set of action alternatives represents one of the primary problems facing development of successful computer planners in all domains, not simply game playing. The crux of the problem, as it is commonly presented, is that finding solutions to planning problems requires either an epistemologically intensive, or a time-intensive approach. The former requires that sufficient knowledge (e.g., about strategies, good skeletal planning sequences, etc.) be included in the planner to guide the decision making required to select among competing courses of action. The latter approach, on the other hand, foregoes this kind of goal directedness, developing all branches to a certain level and then picking the best move by applying some criteria to the family of alternatives.

Several researchers have recognized this dilemma and suggested that these two-solution strategies are arrayed along a kind of continuum and that other methods that attempt to combine the two lie between the two extremes. McCarthy and Hayes (1969) and Berliner (1973) are examples.

As Berliner pointed out, approaches to planning action sequences which depend either entirely on knowledge or search are essentially uninteresting from the standpoint of Artificial Intelligence. The question then becomes one of creating a model which is both epistemologically and heuristically adequate (following McCarthy and Hayes). In other words, the aim is to use just enough knowledge and just enough search to obtain successful and interesting results.

Important innovations in representing planning knowledge which emerged from PGSC's second-phase effort center upon 1) the use of search as a form of knowledge and 2) a framework for representing this explicitly.

The phase-two planner, ARES, builds plans by engaging a given world situation or environment in a planning dialectic. The plans are constructed out of generic goal structures representing generalized planning knowledge about the domain. Based on the system's analysis of the current world situation, these general pieces of knowledge are informed and related into a network of specified goals. In the current implementation, uninformed goals are s-expressions in the LISP Language. Each goal has a number of properties which are slots in the atom's property list. The goal name identifies the atom or s-expression. The property slots include the following:

(1) COUNTERGOAL -- an assumed adversarial countergoal,
(2) SUBGOAL -- a list of possible subgoals for when the agent is the curren. actor in the environment,
(3) SUB_NOT_ON_MOVE -- a list of possible subgoals for when agent is not the current actor in the environment,
(4) FEASIBLE -- a list of feasibility conditions,
(5) SUCCESS -- a list of success conditions, and
(6) FAILURE -- a list of failure conditions.

While a value or list is initially attached to each uninformed goal, these values are used only as a starting point for producing specified or filled-in attachments as the goal is developed during the course of a planning sequence.

PGSC distinguishes between informed and uninformed goals on the basis of a specification which is necessary in order to instantiate a particular goal. An informed goal is defined as a goal which has been developed in a context-sensitive fashion by transforming its initial attachments or properties into specified attachments. The difference between an uninformed goal and an informed goal is illustrated as follows. Imagine a human planner operating in a tactical combat situation. The planner has a number of general-purpose maneuvers or "goals" which he can use, given varying circumstances -- an example would be "march to contact." As such, this operation would be similar to what we are calling an uninformed goal -- it is devoid of any

contextualizing circumstances and lacks "filling-in." Once the planner chooses to pursue the goal, however, he decides to march some particular set of forces to contact with some adversary at a given point and time. In this way the goal becomes specified or informed.

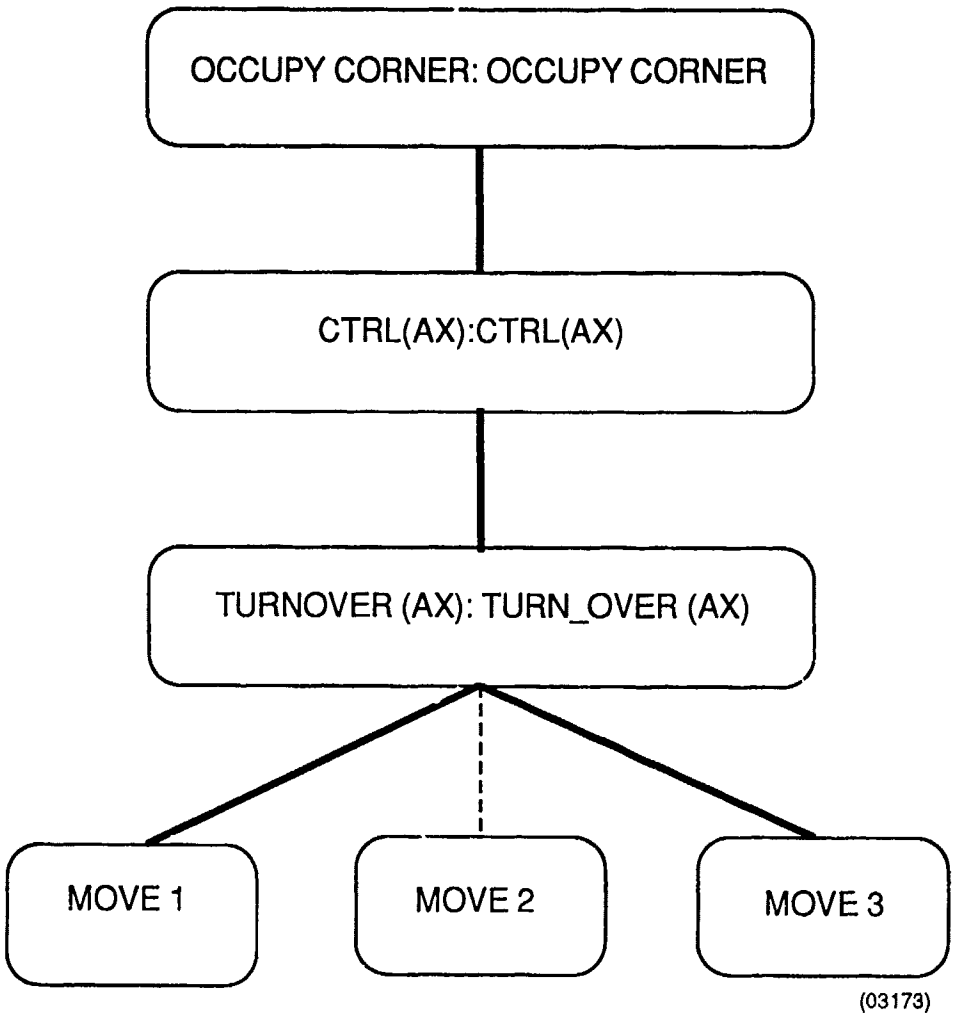
The actual filling-in of an uninformed goal occurs when, in the development of a Contingency Goal Tree (CGT), ARES expands an existing node. In planning, not only must an agent's plan be consistent between levels of abstraction (see Section 2), but also the agent's model of its adversaries' countergoals and the success/failure tests for the goal must reflect this same consistency. Thus, while an agent's goal of "CAPTURE (X)" might be generally paired with and adversarial countergoal of "DEFEND (X)," it is not necessary that in a specific situation the adversary have that countergoal. It might be the case that in the parent node the adversary's countergoal was NIL (indicating that ARES expects the adversary to ignore its goal), and, therefore, it makes no sense for there to be a countergoal of any child of that node.

ARES specifies the countergoal attachment by conjoining the set of generic countergoal attachments for the goal with the specified or informed set of counter-subgoals from the parent, thereby making sure that each step in the adversary's counterplan is consistent with its predecessor. Similarly, it is also necessary to make sure that the success or failure tests of any node in a network include consideration of the parent nodes. This is because, for ARES, the planning of a move/countermove sequence is dependent upon the ability to recognize success or failure across levels of hierarchical abstraction. As an example, consider the case depicted by the move tree in Figure 4-1. This example is taken from the game of Othello (see the detailed description of ARES' behavior in this domain that follows). The planner's goal is to occupy corner X, and the assumed countergoal is also to occupy X. To occupy corner X, ARES has identified CTRL(AX) as a subgoal which is paired with CTRL(AX) as a counter. Since AX is already occupied by an adversarial piece, the only way it can be occupied by ARES is to flip or turn the piece over. TURN_OVER(AX), then, becomes ARES' next subgoal with a countergoal of TURN_OVER(AX).

The knowledge base assumes that, if the agent's objective is to turn over a piece, the adversary will want to turn the piece back over again. ARES first makes move 1, which flips AX over. Next, playing for the adversary, ARES finds move 2, which will flip AX over again (back to its original configuration). Move 2 just happens

to be a play into corner X, however, and, by making this move, the adversary would be able to thwart ARES' level-1 goal of controlling that corner. It therefore makes no sense for ARES to consider this sequence any further -- it is a bad plan. ARES can only realize this based on a failure-conditions analysis which it would make before processing the next goal (a potential move 3).

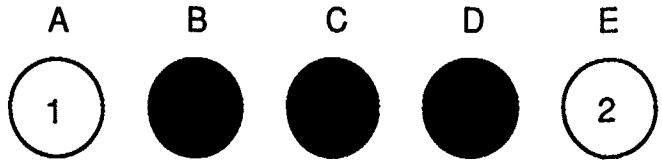
The specified failure lists, used to determine when some sequence represents a bad plan, results from the conjunction of the generic goal attachments of the goal as it is in its uninformed state with the set of all specified failure conditions from parent goals in the same path. Thus, ARES has encountered a failure if the white side has turned over (AX) and white occupies (X).



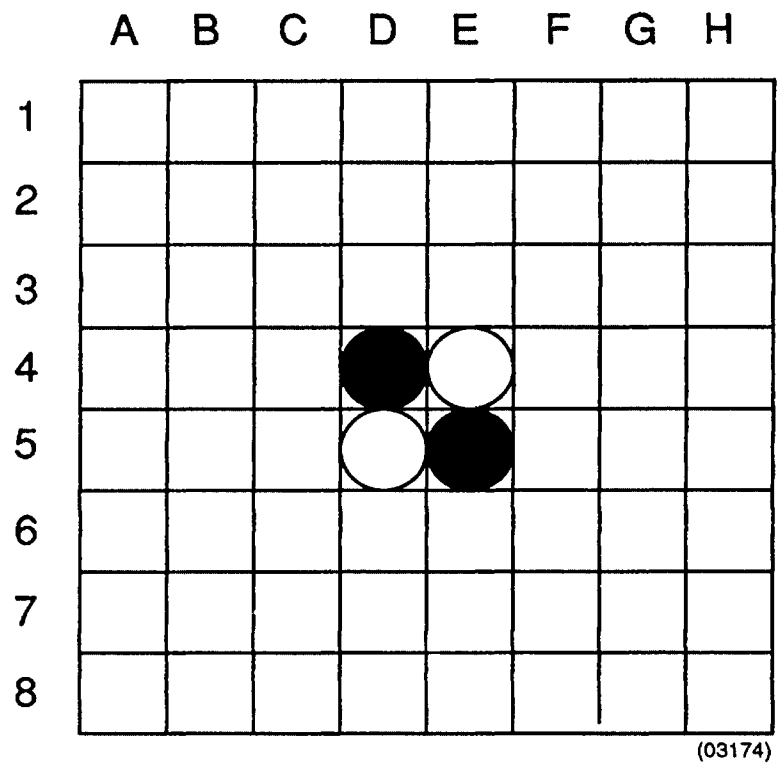
OTHELLO Move Tree
Figure 4-1

5. OTHELLO EXAMPLE

This section presents a detailed example of an *RES planning sequence from the popular two-player, perfect-information game Othel Othello is played on a board of 64 squares (8 x 8). Two colors of stones (white and black) are used for the respective players. The object of the game is for a player to occupy more squares with his color of stones than the opponent. A square is captured by outflanking an opponents stone (or a row of stones). Outflanking is accomplished by being able to make a play such than an opponent's stone(s) is(are) enclosed by two friendly stones, one at each end. The following is an illustration:



White has Stone 1 at Position A, and Position E is open. By playing a stone at Position E, white is able to outflank the three black stones located at B, C, and D. A player (either white or black) is able to make a valid play any time a stone can be placed in such a fashion so as to outflank one or more of the opponent's stones. Once such a play is made, the opponent's stones in the intervening spaces are "flipped," that is they are changed to stones of the color of the player making the outflanking play. For instance, in the example shown above, once white played Stone 2 at Position E, the black stones at Positions B, C, and D would be changed to white stones. If a player cannot make an outflanking play, he must "pass" on that turn and wait for another opportunity. The game ends when all of the spaces on the board are occupied by stones. The player whose color stones occupy the majority of spaces wins. In the computer implementation of Othello which has been built for demonstrating the planner, the computer plays for one side against a person (or itself). The planner plays for the white side and the opponent plays for black. The initial configuration of the sixty-four (8 x 8) square game board is as shown in Figure 5-1. Two black stones and two white stones are initially placed in the center of the board. Black (the opponent) is allowed to move first. In the example, which is taken from an actual game, the moves of the planner (white) will be explained in terms of the contingency-goal trees used to arrive at those moves. To make the planning sequence more



Initial Configuration for OTHELLO Game Board
Figure 5-1

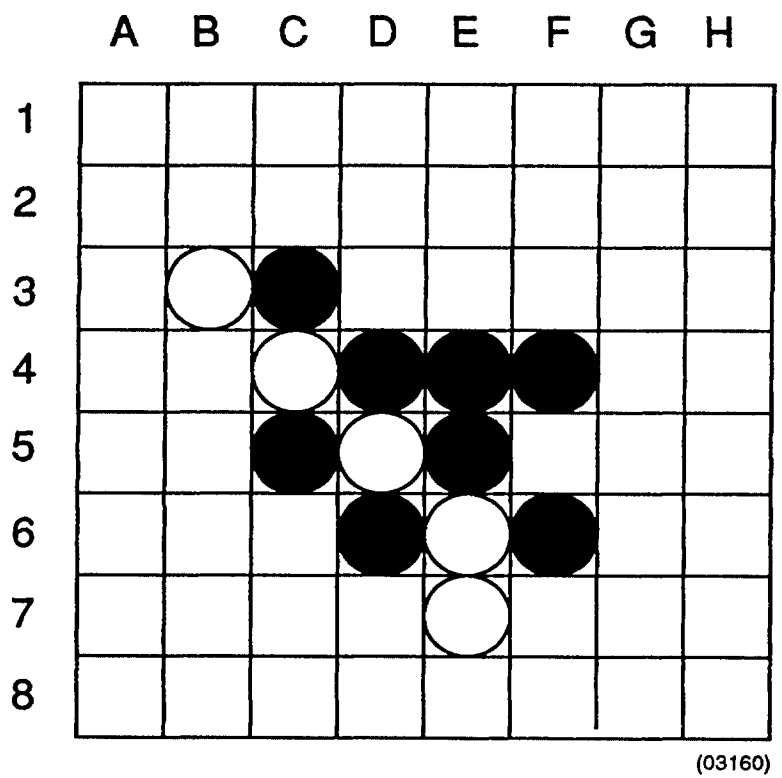


Figure 5-2

interesting, a number of moves already played by both the black and white sides have resulted in the configuration of the playing board shown in Figure 5-2.

Figure 4-1 showed the current contingency-goal tree that the planner is considering. In the bottom left corner of the board, white, which in this case is ARES, has a play which will allow it to occupy Square (C6), which is two squares away from the Corner (A8), thus satisfying the Subgoal "ctrl_1_2_away". Having found this move, ARES plays it on a hypothetical-move board (see Figure 5-3) and then switches sides to play for black to see if there is an adversarial countergoal which will be achievable as a result of ARES' move. Black's assumed countergoal of white's "play" is "NIL," and so ARES (playing for black) backs up the tree one level to the parent to check the countergoal there. The parent goal to ARES "play" is "play_safe_(C6)," and the assumed adversarial countergoal is also "play_safe_(C6)." "Play_safe" is a goal object defined such that a play will be made on a given square and the opponent will not be able to turn the stone over. Black now has the countergoal of trying to occupy (C6). It can accomplish this in one of two ways: 1) play on the space if it is not occupied and represents a valid move, or 2) attempt to turn over an opponent's piece occupying the position. Since white is (in the hypothetical-move environment) occupying (C6), black's only hope is to attempt to turn over that piece. Since a black play at (C7) will allow it to do so, ARES makes this play for black on a new hypothetical-move board (see Figure 5-4). ARES now switches back to its side again and considers the most recently added node to the CGT (the black play at (C7)). Since the countergoal of the black play is "NIL," ARES backs up to the black parent goal of "turn_over_(C6)," the countergoal of which is also "turn_over_(C6)." ARES now attempts to find a new white play which will result in the turnover of (C6). A play at (B6) is such a move and so, as before, a new hypothetical board is created and ARES' new move is played on it (see Figure 5-5). ARES switches sides back to black and backs up as before to white's goal of "turn_over_(C6)," which is paired with "turn_over_(C6)" as a countergoal. Since black has a play at (A6) which will turn (C6) over again, this move is played on a hypothetical-move board at yet another level (Figure 5-6). Switching back to white, ARES locates another move, at (B7), which will flip the target piece (C6) back, and this play is made on the next-level hypothetical-move board (Figure 5-7). Another switch is made to the black side,with ARES backing up as before the the "turn_over_(C6)" goal. Black now has a play at (A8), which will flip (C6) over again, and this move is, in turn, played on a hypothetical-move board (Figure 5-8). At this point, ARES does not yet realize that implicit in black's play at (A8)

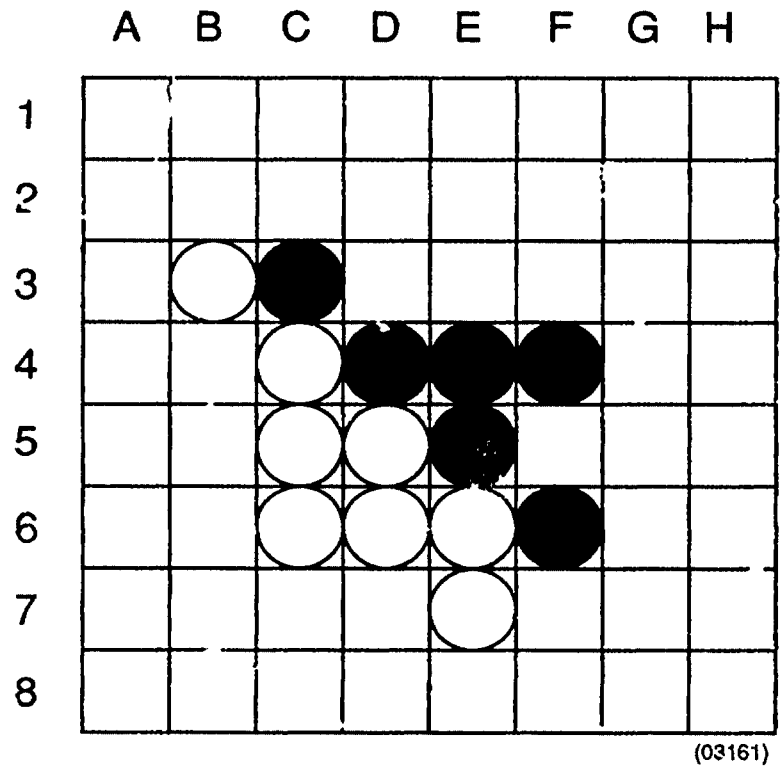


Figure 5-3

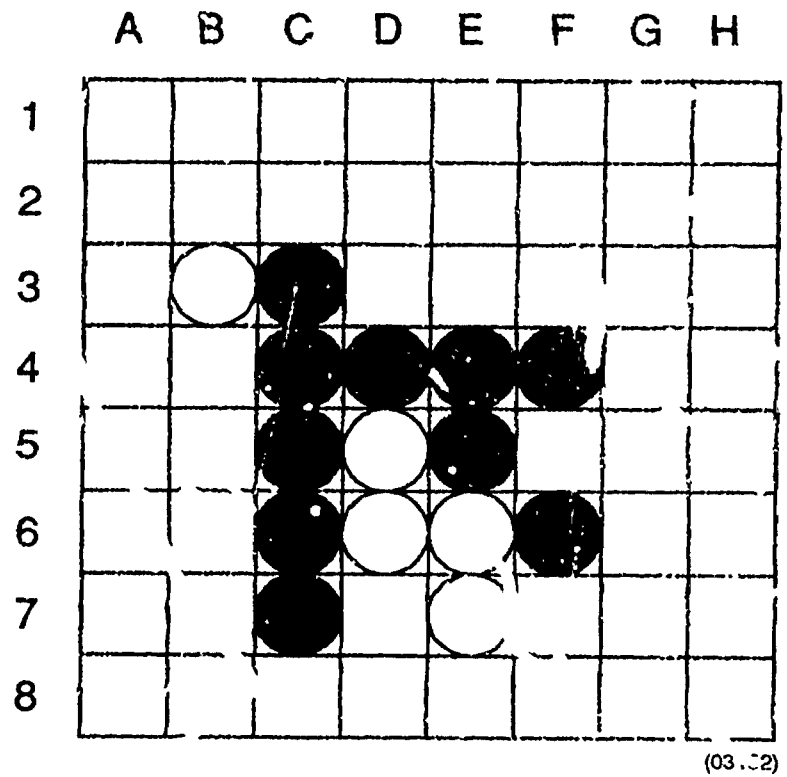


Figure 5-4

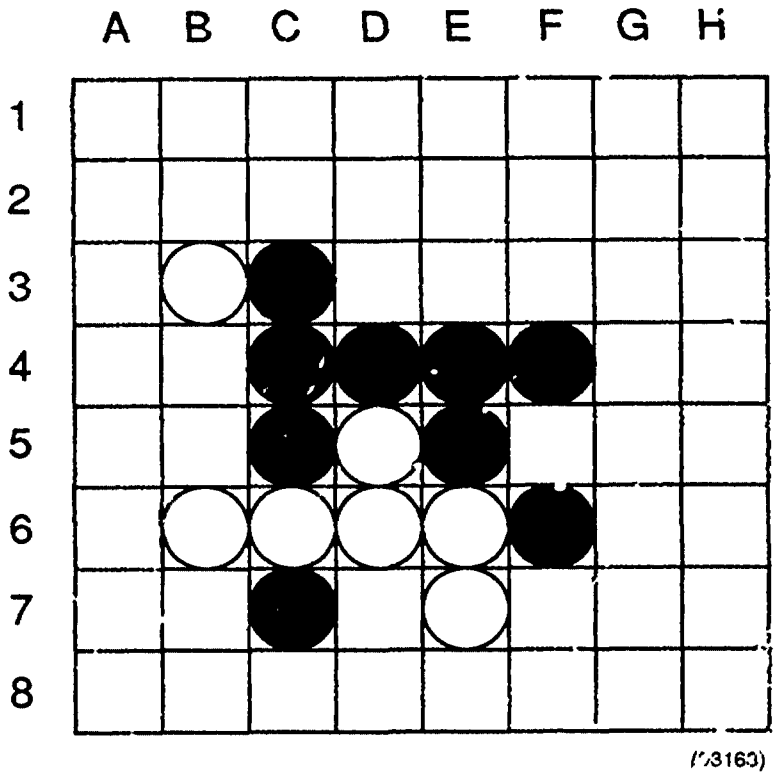


Figure 5-5

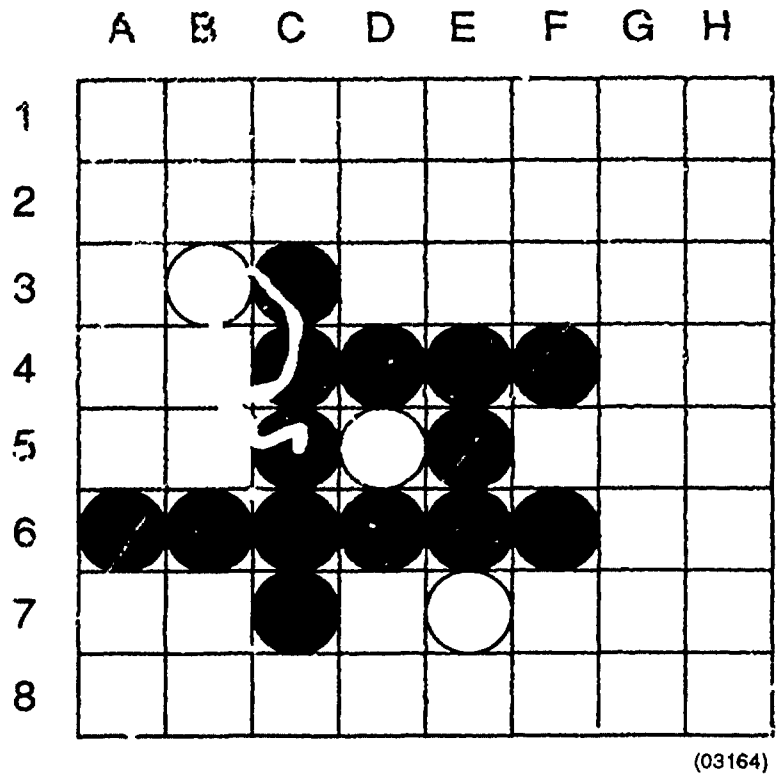


FIGURE 5-6

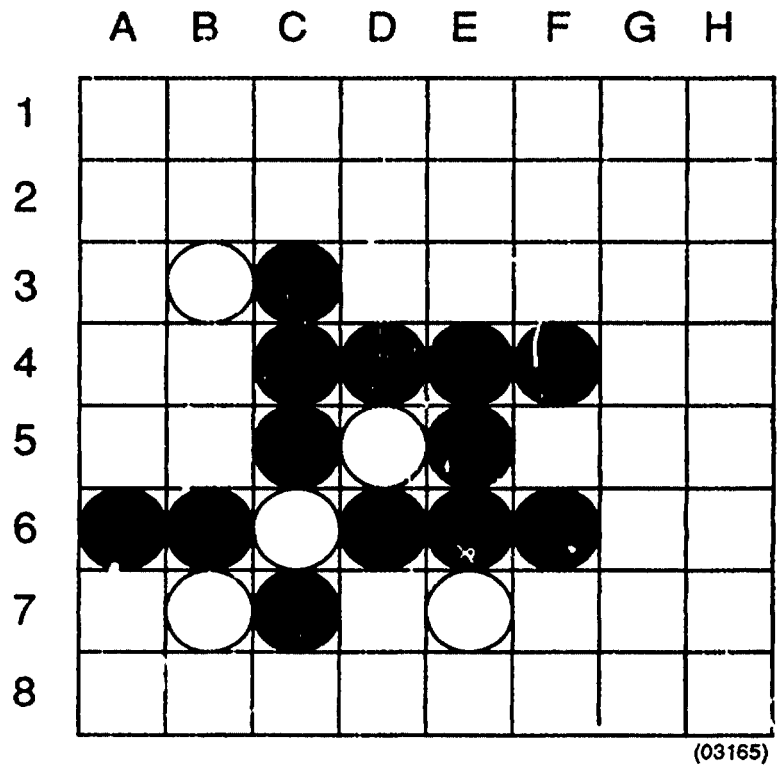


Figure 5-7

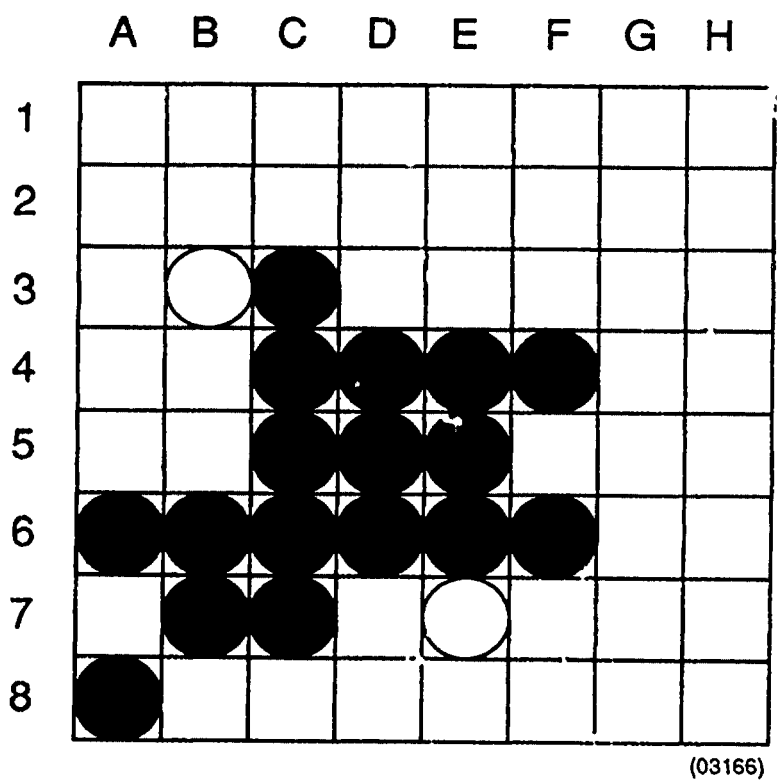


Figure 5-8

is the automatic failure of its upper-level goal of "improve_corner_(A8)," and so it switches sides back to white and backs up the tree to the turnover goal. Examining the success/failure conditions for this node, ARES finds that, since black has occupied the corner at (A8), the opponent has succeeded in defeating white's objective. It does no good to continue to attempt to turn over (C6), since the whole point of this play was to set up the corner at (A8) and this present contingency has resulted in the loss of the target corner.

It is important to notice that, without the success/failure tests, ARES would have gone ahead looking for move sequences that would allow it to occupy (C6) and would not have known that any move sequence that allows black to occupy the corner, even if it results in the success of the subgoal, is a greater success for the opponent. Having realized this implicit failure, however, ARES begins to replan. The first step is to pop back up two levels in hypothetical worlds, returning to the situation before it made the play at (B7) which led to the situation of black occupying the corner at (A8) (See Figure 5-6). Now ARES proceeds to plan for white with a new goal of turning over (C6) AND NOT playing at (B7). As is evident from an inspection of Board 5 (Figure 5-6), there are no white moves other than (B7) that will flip C6 over. Thus, ARES must search back even further, popping up another two levels in boards to where it was before it played at (B6) (see Figure 5-4). At this point its goal is to turn over (C6) AND NOT play (B6). Here, ARES has another alternative: it can achieve this goal by playing at (B7). It may seem peculiar that such a move would be considered since it was a similar play at (B7) which allowed black to occupy the corner in an earlier contingency, but there is no necessary reason to assume that by introducing it now, a similar failure will occur (even though that is exactly what happens). Having discovered this new move, ARES proceeds to create a new level of hypothetical-move board (see Figure 5-9) and places this move on it. Switching sides as before and backing up the goal tree for white, ARES is now looking for a move which will turn (C6) back over. It discovers, as before, that by playing at (A8) it can turn the piece in question (C6) over once again (see Figure 5-10). As before, after switching sides and checking its success/failure tests AREAS finds that its upper level goal has failed and so it must once again attempt to replan. Levels of hypothetical-move boards are backed up to the situation as it was before ARES played at (B7) (see Figure 5-4). Now the goal is to turn over (C6) AND NOT play (B6) AND NOT play (B7). As is obvious from the game-board diagram, there are no available moves which will allow this goal to be achieved. ARES, therefore, pops up two more levels in hypothetical-move

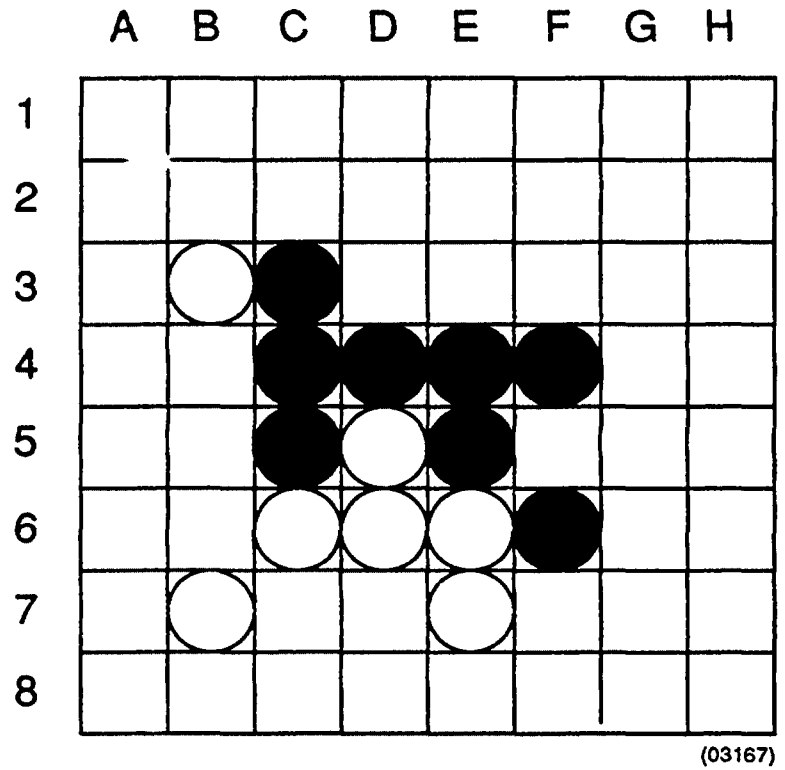


Figure 5-9

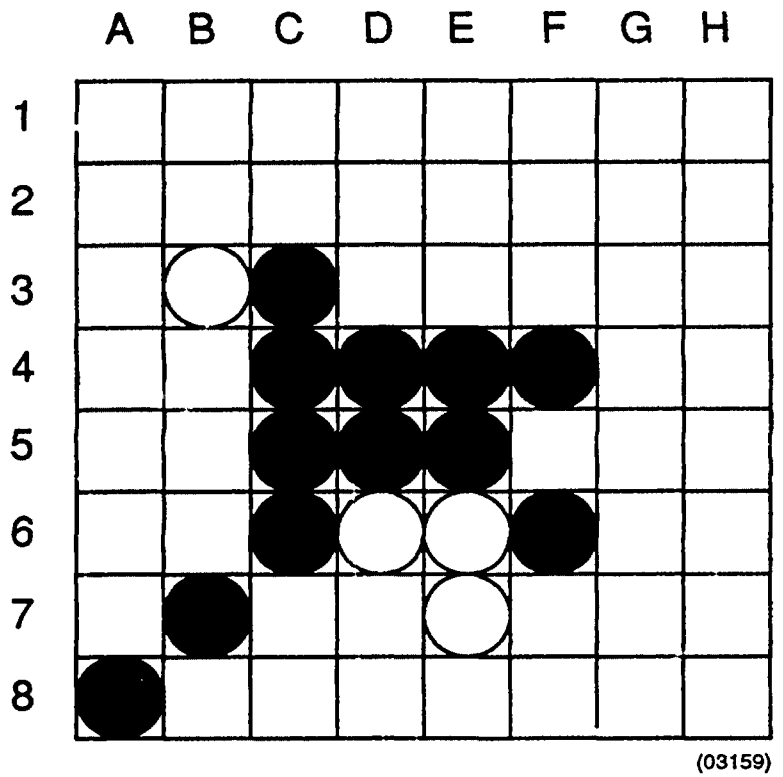


Figure 5-10

boards to the situation as it was before it played at (C6). As there are no other moves that will allow ARES to occupy this square (C6), it abandons this target and will instead try to see if there is another way in which it can improve its position in this corner. ARES will continue to consider move-countermove sequences until it finds a winning combination for itself which black cannot counter. The move/countermove tree that is finally developed becomes ARES current strategic plan. The other contingency trees are maintained along with their corresponding hypothetical-move boards. When black next moves, ARES will compare the adversarial action with its current strategic plan to see if the adversary is behaving as anticipated. If black's move corresponds to ARES' calculated best move for black, the current tactical plan is still valid and ARES can automatically make its next move as indicated in the plan. If black's move does not correspond to what ARES expected black to do, it is possible that one of the following conditions exists:

1. The adversary is unaware of ARES tactical plan or has abandoned the target and will not react to ARES' offensive, or
2. The adversary is aware of ARES' tactical objective but has failed to identify the best move sequence to thwart the plan, or
3. ARES has made an error in the assumption of adversarial countergoals.

ARES first considers that Case 2 is indeed what has happened and it searches through its collection of stored CGTs and hypothetical-move boards to see if black may have blundered and selected a poor countermove. If the given scenario is located among the stored CGTs, the located CGT is used to update the current tactical plan according to the new move sequence contained in the tree. Because ARES has already thoroughly examined this tree, beginning with black's countermove, it is just as certain that black will fail in this sequence as it was in the original tactical plan.

If, on the other hand, ARES is unable to locate the black counter among its stored CGTs, it immediately begins to develop a new tactical plan, taking into consideration the new situation resulting from black's unexpected countermove. By proceeding in this way, ARES is able to deal effectively with the possibility that the adversary may have abandoned ARES' tactical target, or may be embarking on some offensive of its own. In this case it is necessary for ARES to rethink a tactical plan given the possibility that its opponent may have changed the environment in such a way so as to be posing a serious challenge in some other part of the game board.

Such a reconsideration does not automatically mean that the original tactical plan will be abandoned, only that other alternatives will be considered in light of the adversary's new action, and, if the original tactical plan is no longer the most advantageous for realizing higher-level statistic goals, a new tactical plan will be developed.

6. MANEUVER EXAMPLE

In addition to refining and formalizing the work conducted in Phase I, the Phase II effort extended the adversarial-planning mechanism from the two-player, perfect-information, sequential-movement domains examined (primarily Othello) to a more robust and realistic environment. Since the ultimate goal was a planner that can effectively operate in battlefield environments, the researchers decided to create a simple corps-maneuver-planning scenario as the target domain.

The school scenario used at the U.S. Army Command and General Staff College, Ft. Leavenworth, KS, was selected as a basis for the new planning domain. A wargame consisting of seven distinct maneuver units was abstracted from the detailed scenario. This generalized scenario is depicted in Figure 6-1. The following extract from the Corps operations text (lesson 3) describes the scenario the research effort attempted to plan.

> "The corps main attack will concentrate in the south (right) to defeat the 1 GTA by rapid penetration of its main and second defense belts thus destroying the continuity of threat defense and rear services systems. The corps attack will be conducted with three mechanized divisions attacking in sequence from north to south. At D-day, H-hour, the 54th Mech Div attacks in the north (left); at H+12, the 53rd Mech Div attacks in the center. The 52nd Mech Div is the corps main effort and attacks at H+14 in the south (right). The sequence of the corps attack is designed to cause the TRs in the second defense belt to shift north from present positions against the supporting attacks in the north, thus reducing the strength of threat forces in the zone of the main attack (52nd Mech Div). The 54th Mech Div will attack through the 23rd Amd Div on the left (north) to penetrate the main defense belt, then continue the attack to penetrate the second defense belt, fix the 55 TR in zone if it counterattacks, and secure objective 1 (NB294Ø). The 53rd Mech Div attacks through elements of the 23rd Armd Div and 2Ø8th ACR in the center to penetrate the main defense belt, fix the 115 TR, & TD, if it is employed in zone; then secure objective 2 (NB4127). The 52nd Mech Div, the main attack on the right (south), attacks to penetrate the main defense belt, then maneuvers to secure objective 3 (NB592Ø). In the

event that threat forces do not shift north as expected, the main effort will be redesignated to the 54th Mech Div or the 53rd Mech Div.

"The corps flanks will be protected by the CENTAG supporting attacks along each flank. The 54th Mech Div will prepare to protect the corps northern flank (left) with the priority between PL HORSE and PL SHARK; the 52nd Mech Div will prepare to protect the corps southern flank (right) from PL ELK to PL SHARK.

"The 54th Mech Div and 53rd Mech Div relieve friendly defending units, in zone, of responsibility for containment of threat forces along the LD/LC by H+14 and H+18, respectively. The attacking divisions will prepare to assist the rapid passage of exploiting forces as soon as the assigned objective is secure. The 2Ø8th ACR prepares to follow the 54th Mech and 53rd Mech Divs, in zone, in order to initiate phase II of the corps operation by a rapid passage and exploitation. The 25th Armd Div prepares to follow the 52nd Mech Div or 53rd Mech Div, in zone, in order to execute phase II of the corps plan."

To simulate this problem the planning mechanism was adjusted to generate plans of attack for multiple units, each capable of a variety of independent actions at different times. Here, the major difference from the prior implementation is that a single "move" for either the friendly or adversary side consists of a coordinated action of several pieces (more than one unit). Each of these coordinated actions can be countered by multiple enemy responses. As before, an appropriate response consisting of some combination of unit actions is postulated as a contingency plan and tested hypothetically through CGT generation to see how that possible response will fare against possible counters. Once an appropriate response is located the planner posts the action and updates the simulation. Planning is then undertaken for the other side.

In order to decide the relative advantage or disadvantage of any particular course of action, a simple outcome simulator was employed to calculate the extent to which a given unit either defeated or was weakened by any unit with which it came in contact. Failed courses of action for either side arose whenever an engagement led to such a diminishment of the friendly force that a breakthrough could not be prevented.

A prototype knowledge base of goals and countergoals for the domain was created by knowledge-engineering activities conducted internally at PGSC. In addition, the Leavenworth course materials for the modeled domain provided an additional source of both friendly and adversary goals.

The remainder of this section depicts the actual planning sequence conducted by ARES in the corps-maneuver domain. In the series of storyboards which follow, each space is occupied by a token. The storyboards are numbered (1-16), and a script for interpreting the planning sequence follows:

(1) Starting positions. Friendly force on left, OPFOR on right. OPFOR goal is to find an offensive plan against which the friendly force cannot defend.

(2) First contingency plan considers a single massive attack through the central gap.

(3) Friendly attempts to counter with a shallow defense.

(4) After 12 hours, friendly force has lost one unit and others are weakened.

(5) After 24 hours, battle is still continuing in central gap, friendly forces are further weakened.

(6) After 36 hours, OPFOR breaks through on southern edge of gap.

(7) After 48 hours, the friendly defense has clearly failed. Planner will now back up and attempt to plan another defense against the OPFOR's single massive attack.

(8) Alternative friendly defense will be defense in depth (using a 24-hour decision cycle).

(9) After 24 hours, OPFOR makes initial breakthrough in northern region of the gap.

(10) After 48 hours, friendly tank division has moved up from the southwest to challenge advancing OPFOR divisions. Defense holds and a satisfactory plan has been discovered that can stop single massive attack by OPFOR. Planner now backs up and attempts to replan for OPFOR.

(11) New contingency plan for OPFOR utilizes dual avenues of approach.

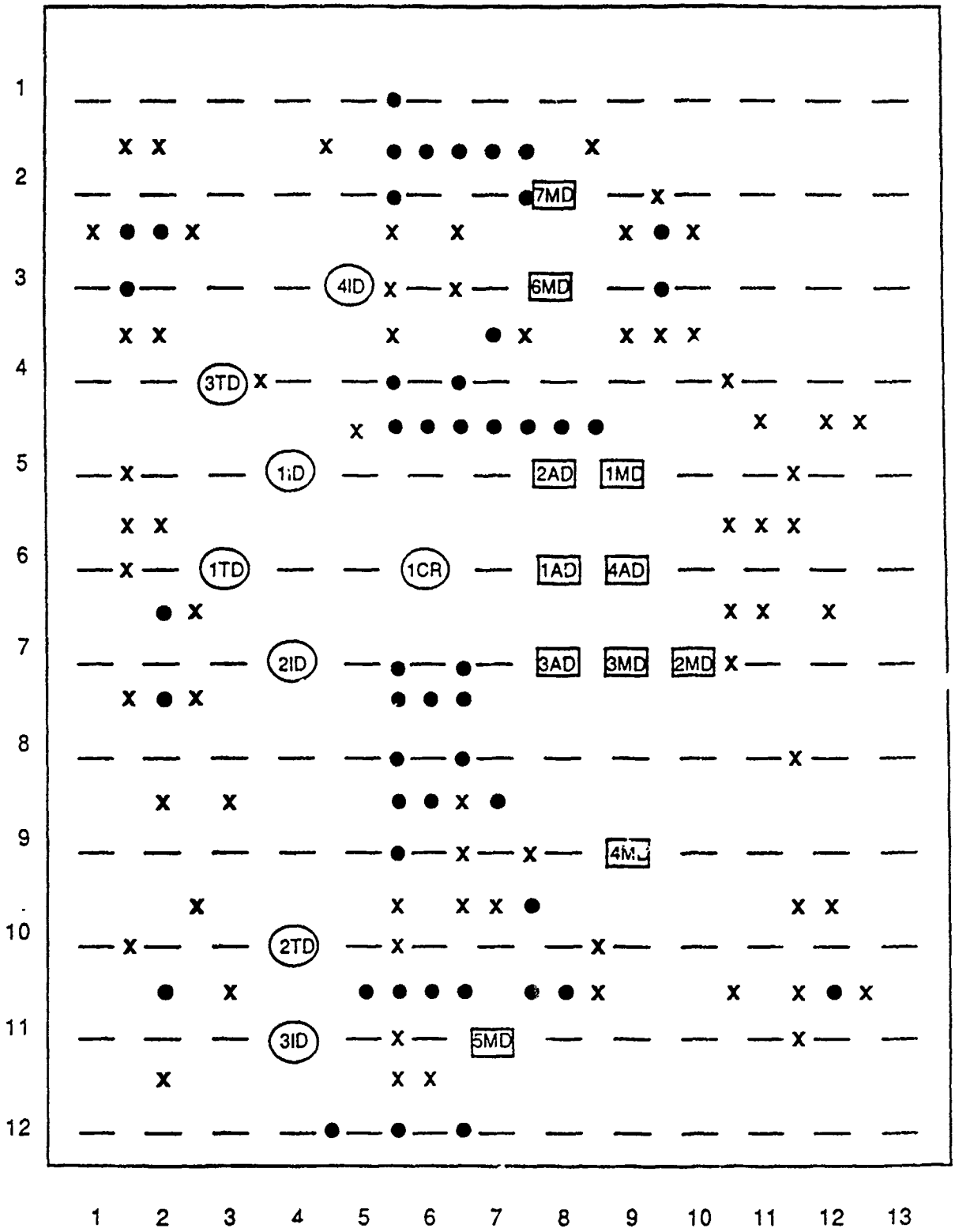
(12) Friendly force counters with defense in depth.

(13) Friendly force is weakening in area of lower OPFOR thrust.

(14) Friendly tank division moves in to meet advancing OPFOR along lower prong.

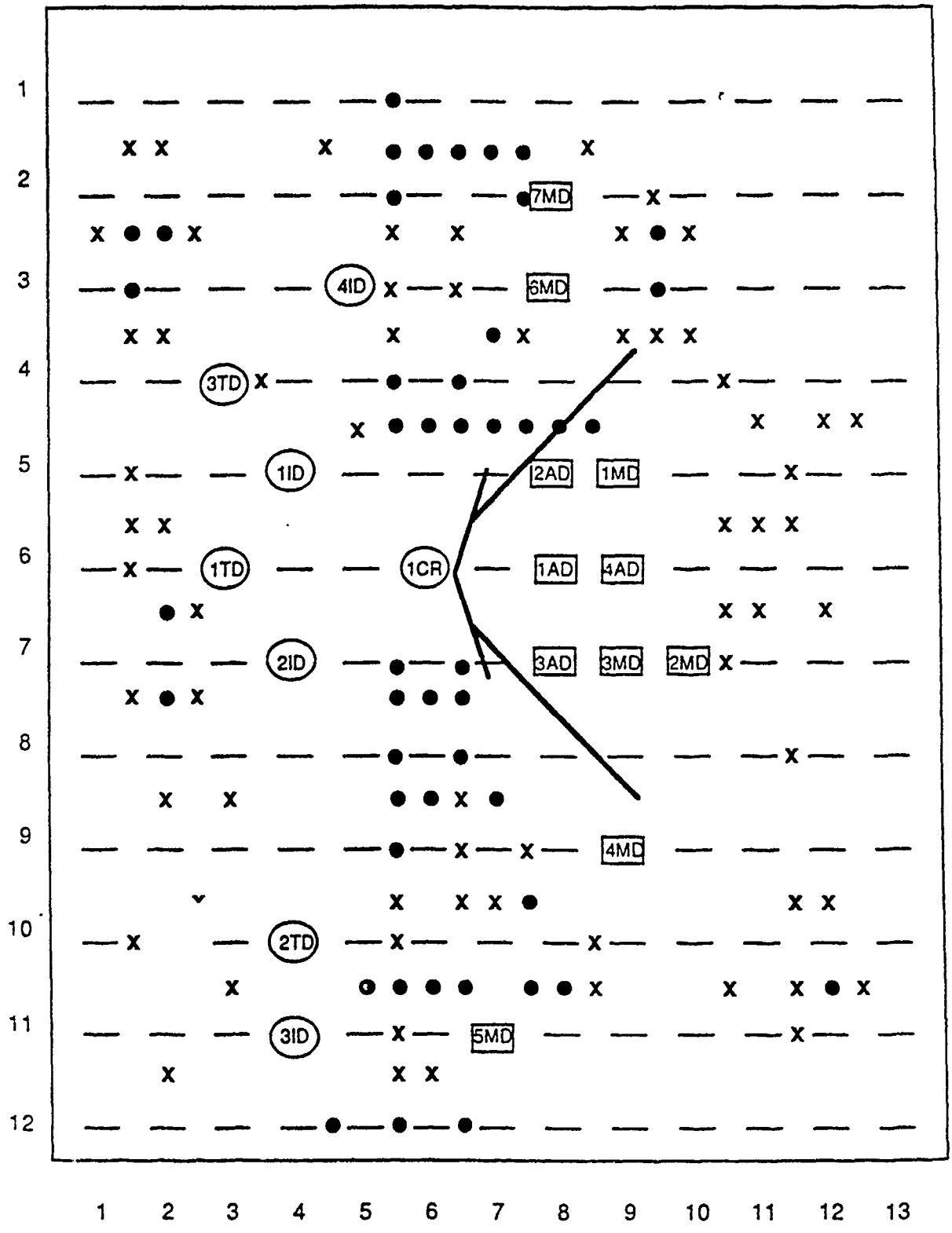
(15) OPFOR breaks through in the north.

(16) OPFOR advance continues in the north. Friendly defense has failed.



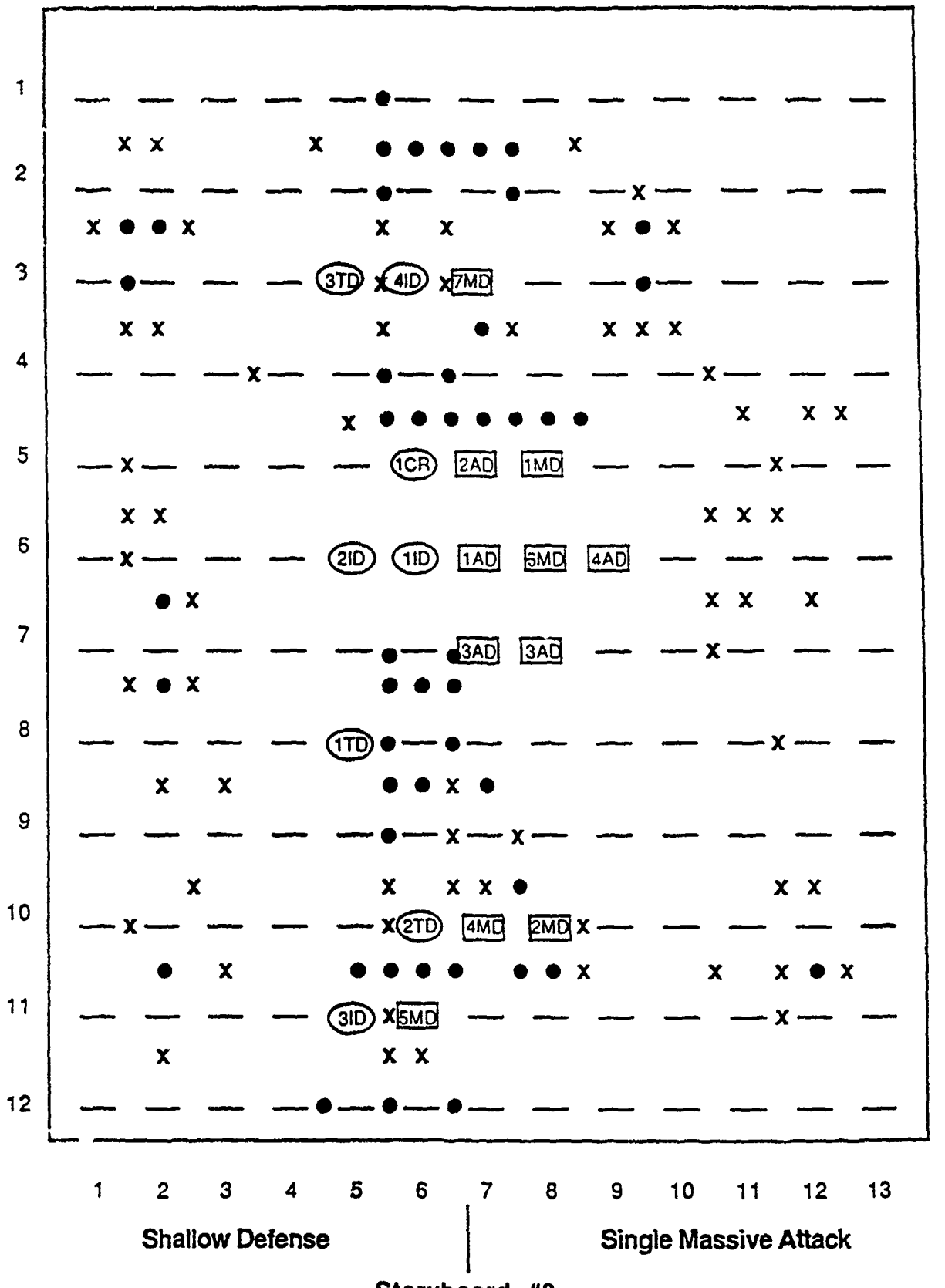
Initial Position

Storyboard #1

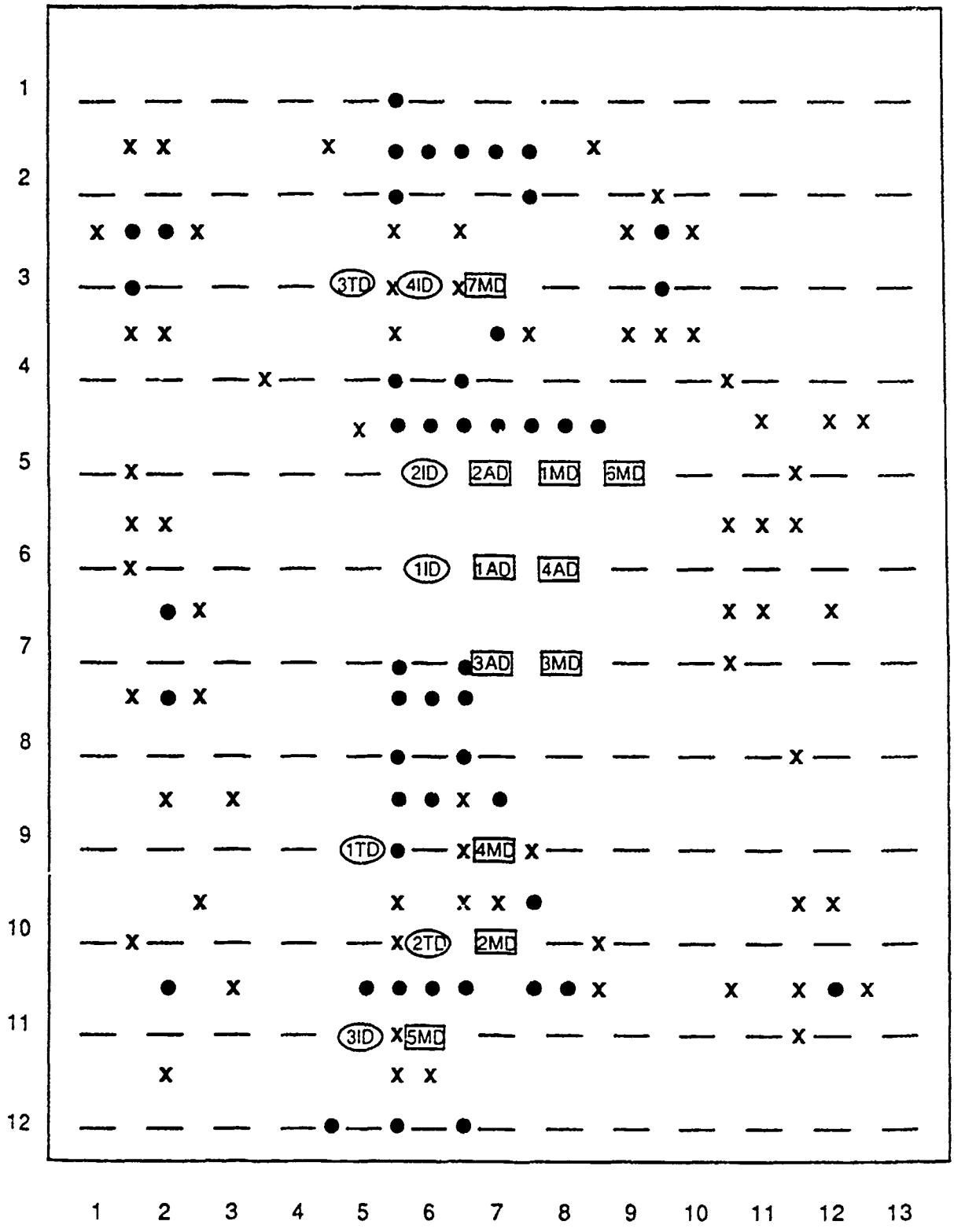


Single Massive Attack

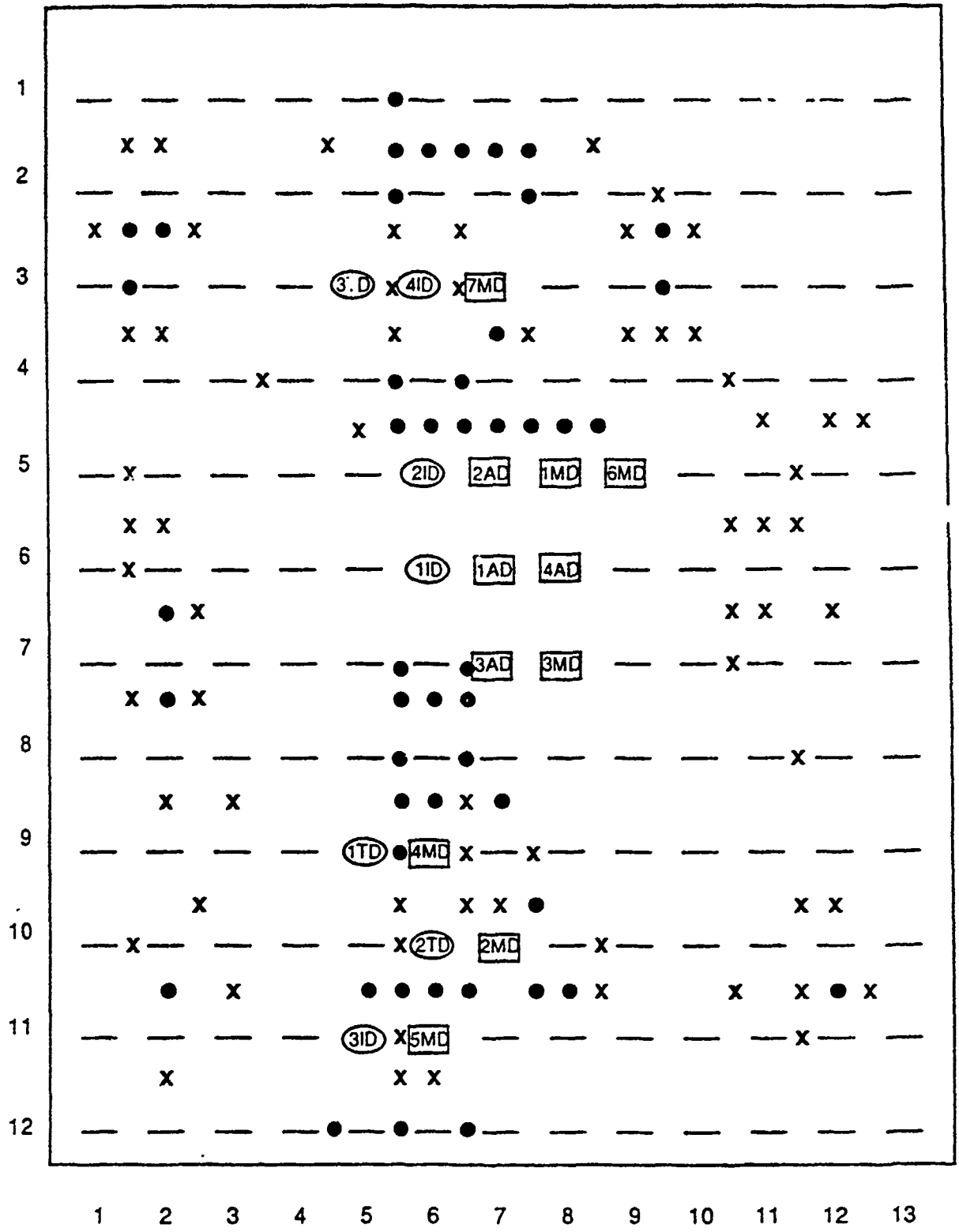
Storyboard #2



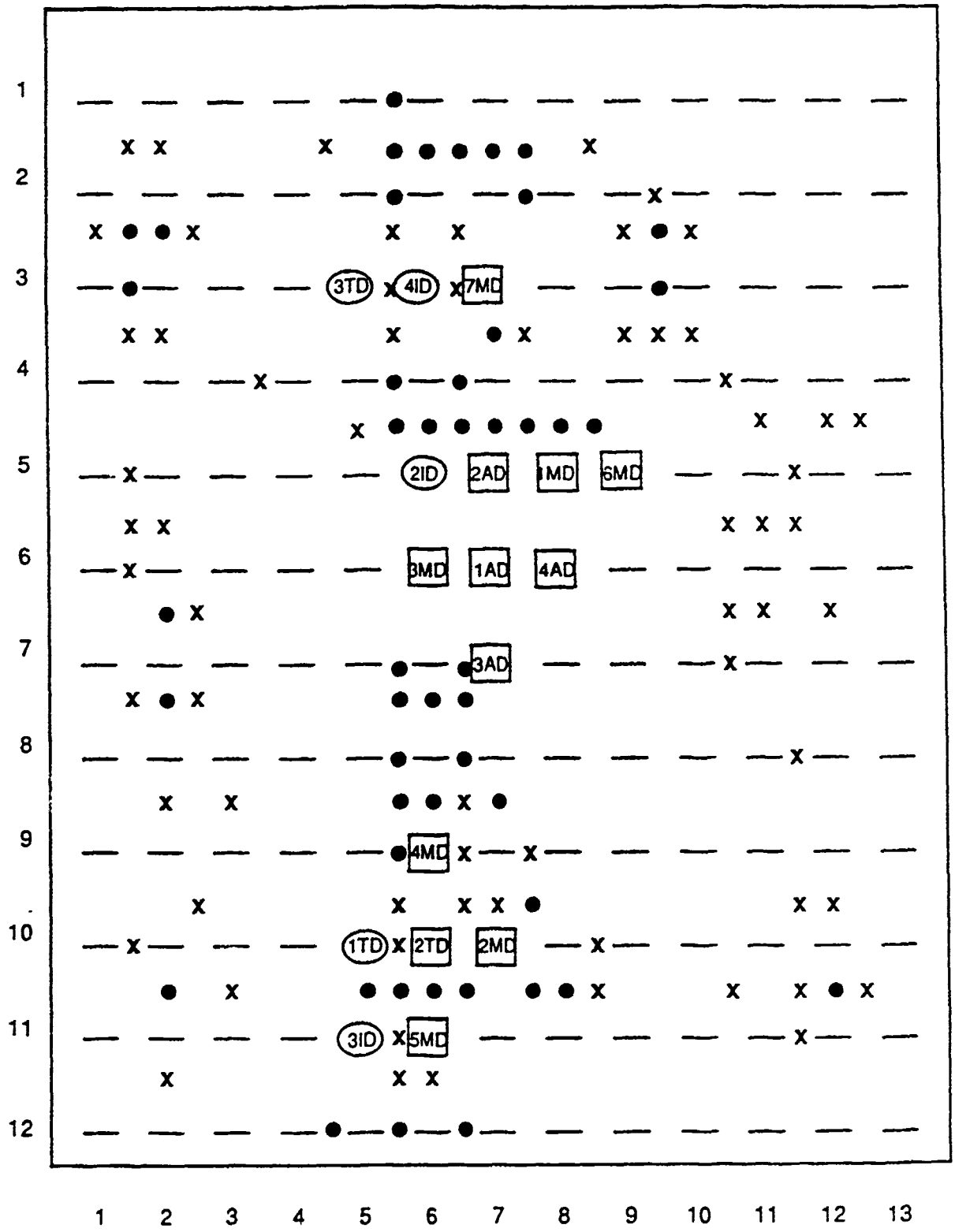
Storyboard #3



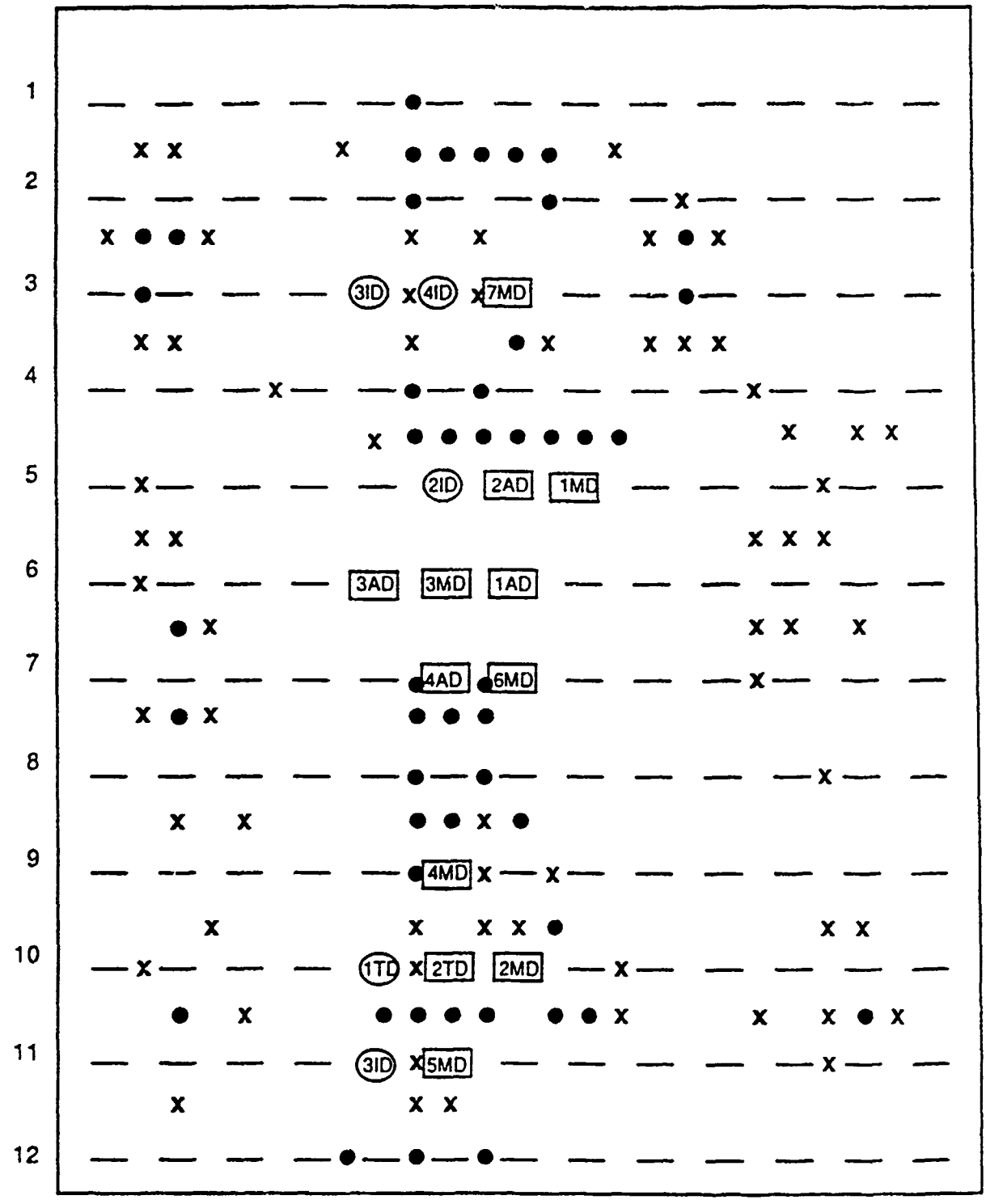
Storyboard #4



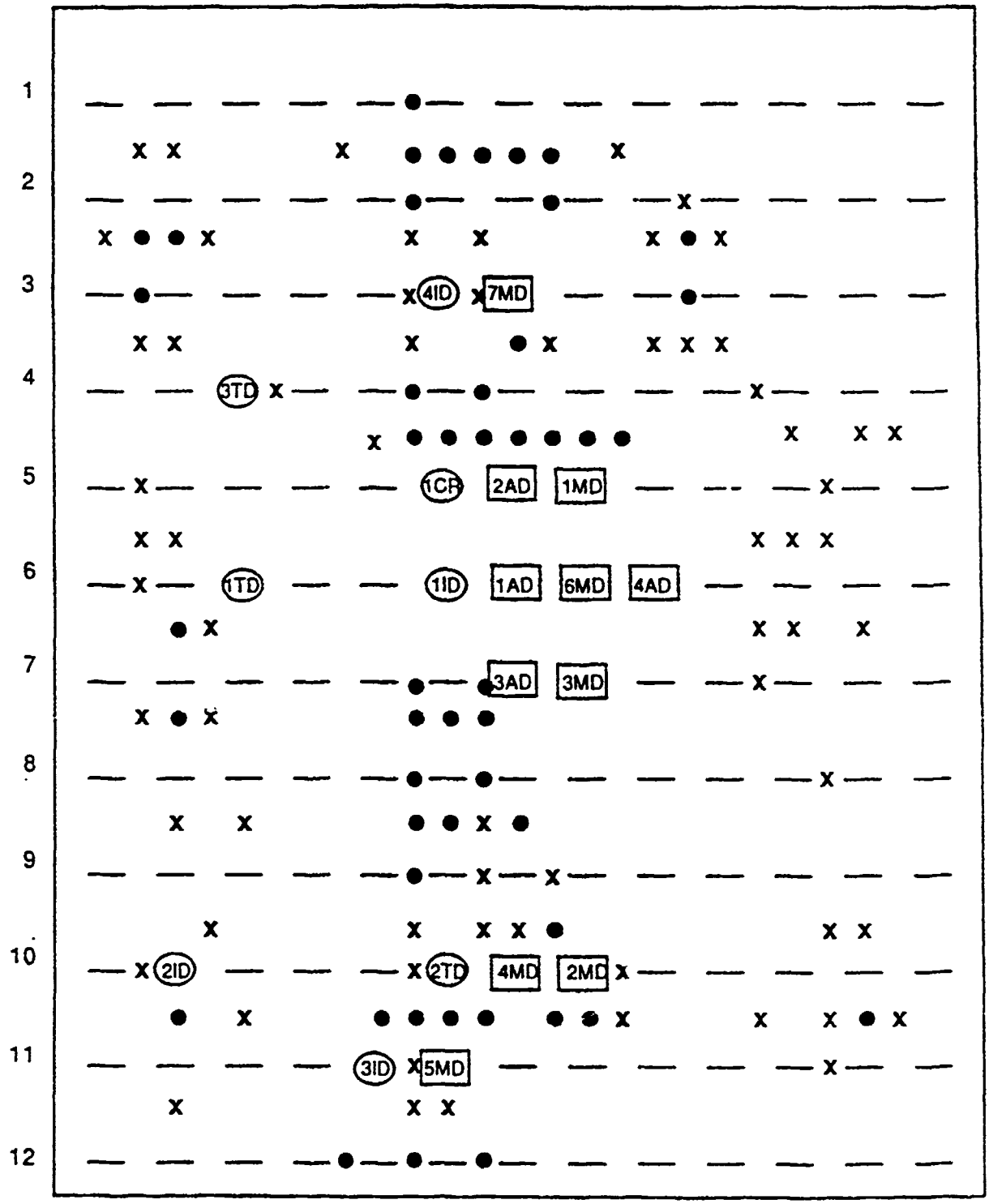
Storyboard #5



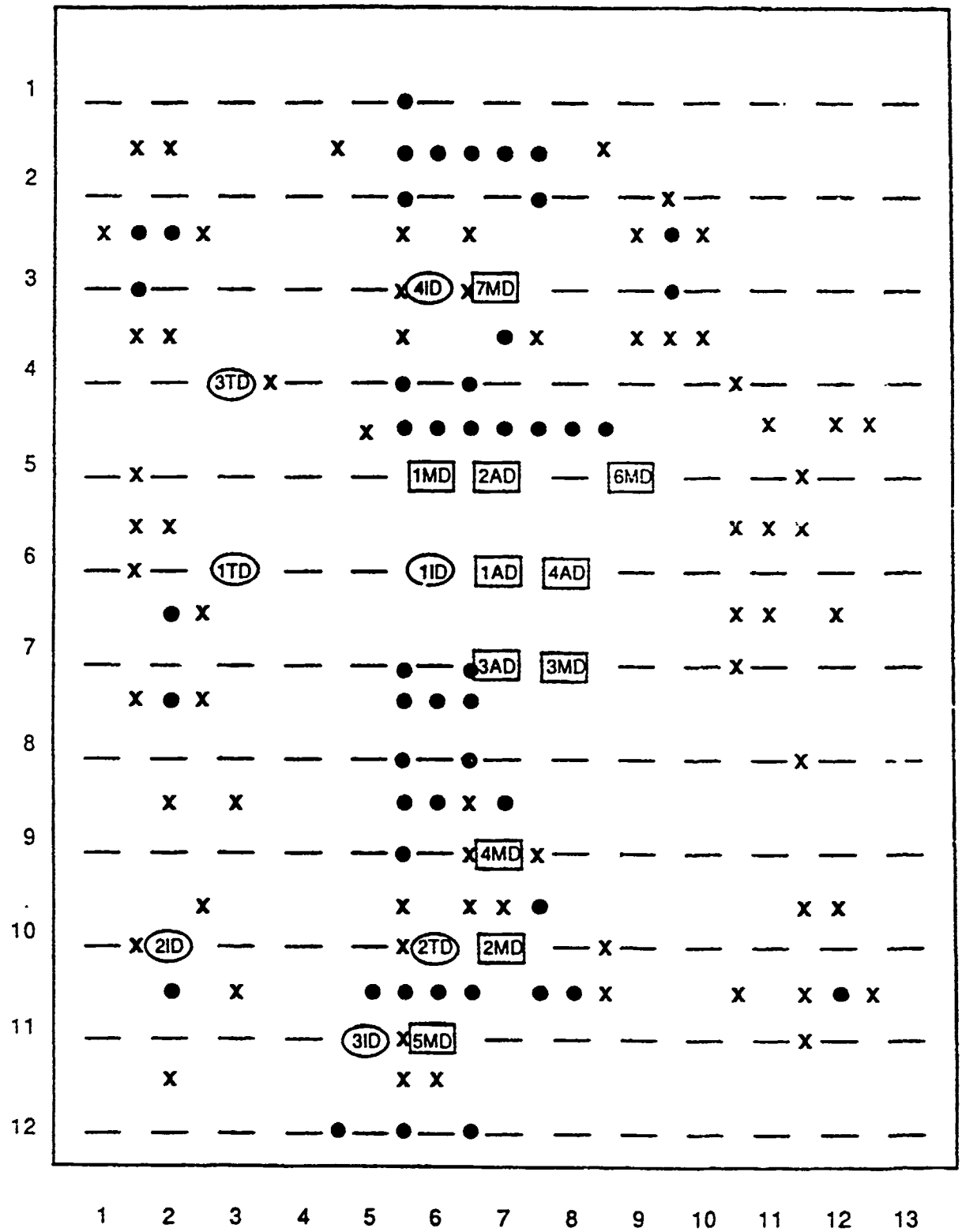
Storyboard #6



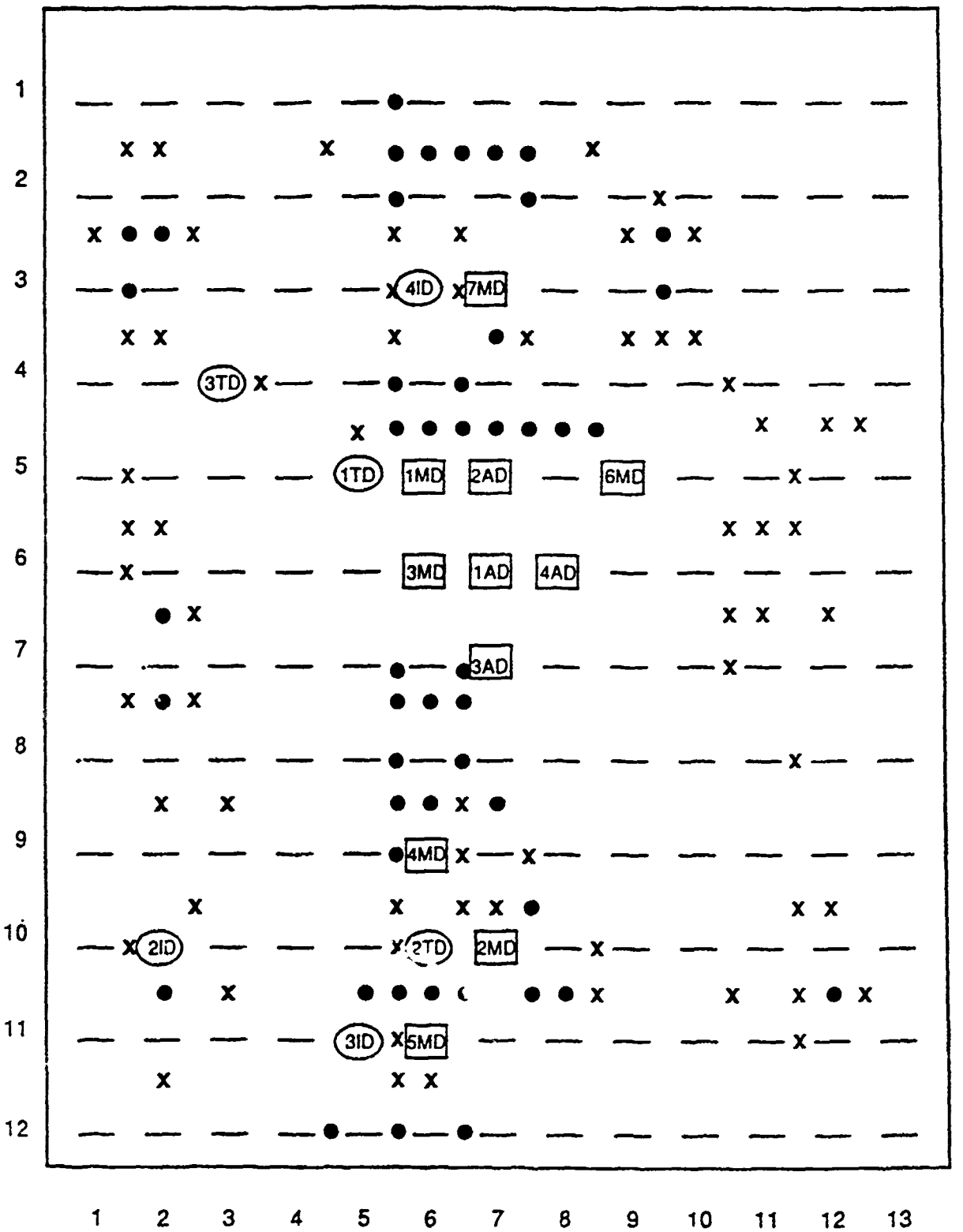
Storyboard #7



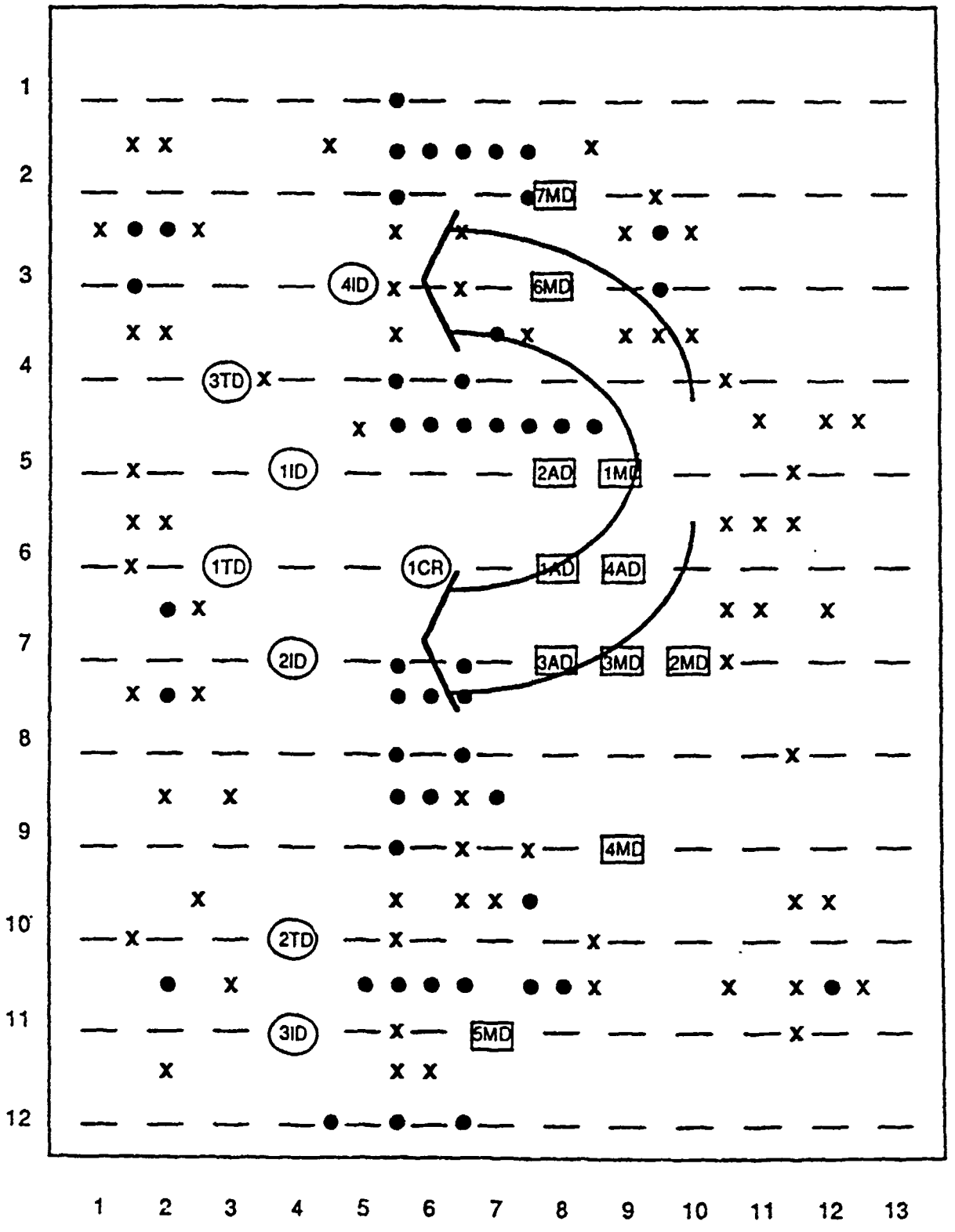
Storyboard #8



Storyboard #9

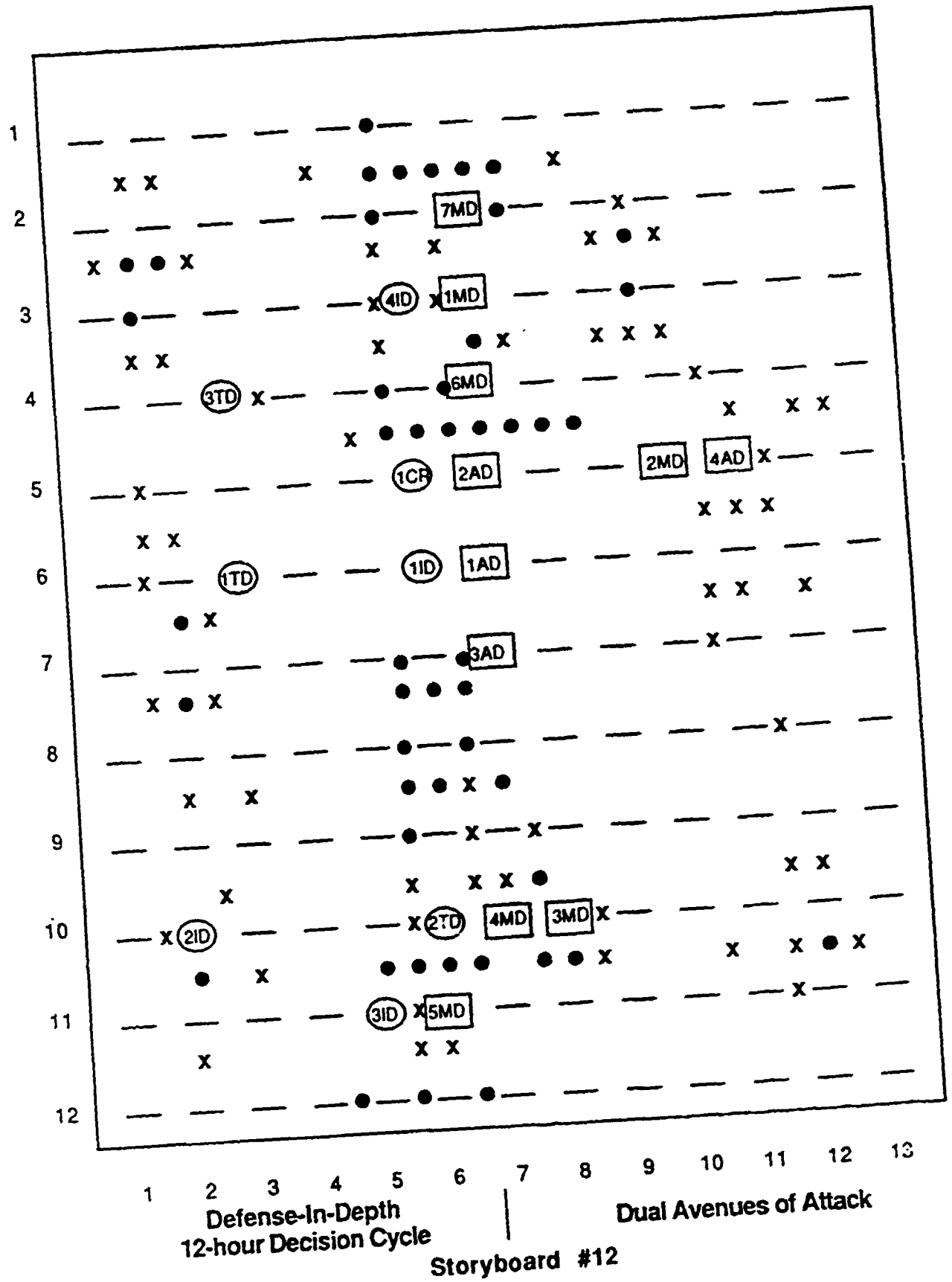


Storyboard #10

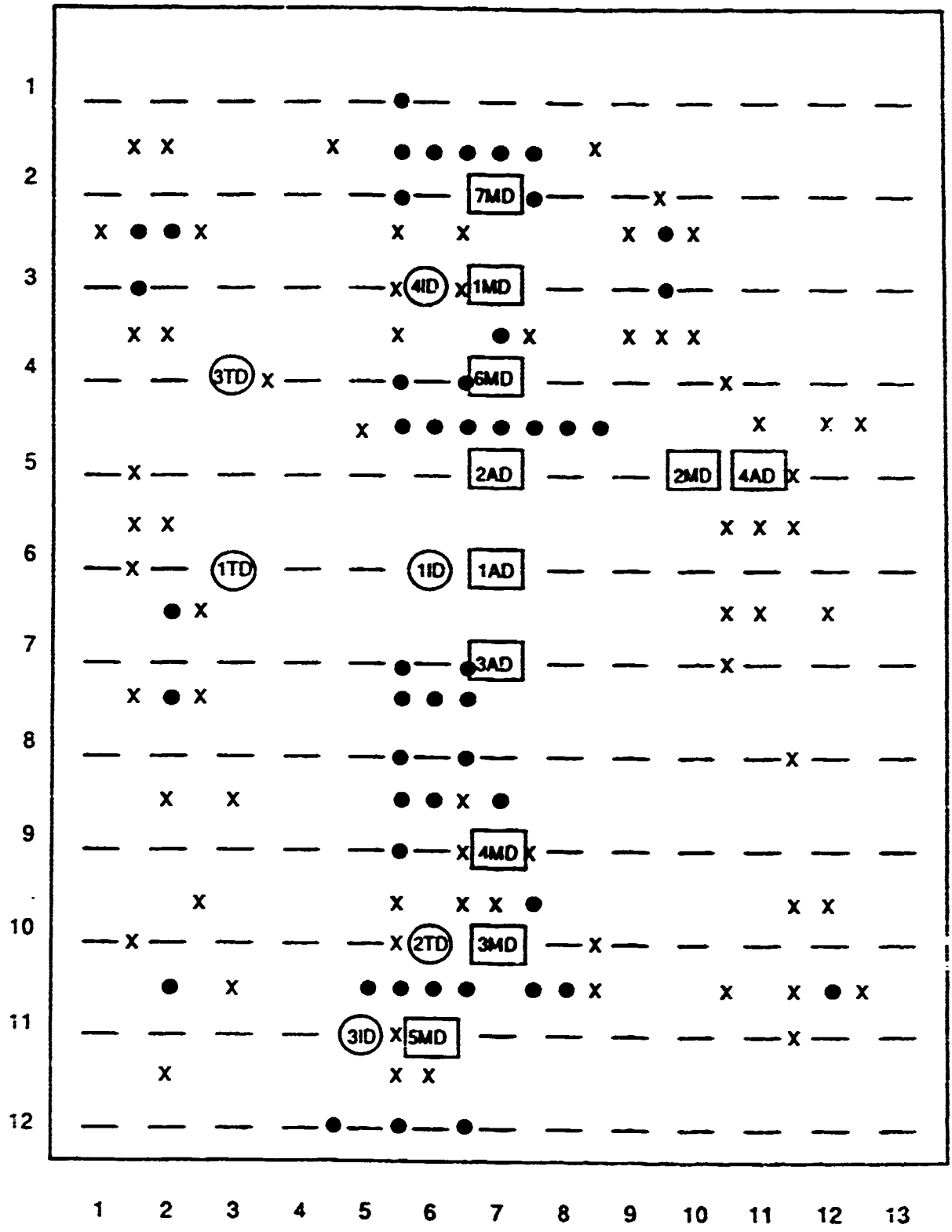


Dual Avenues of Approach

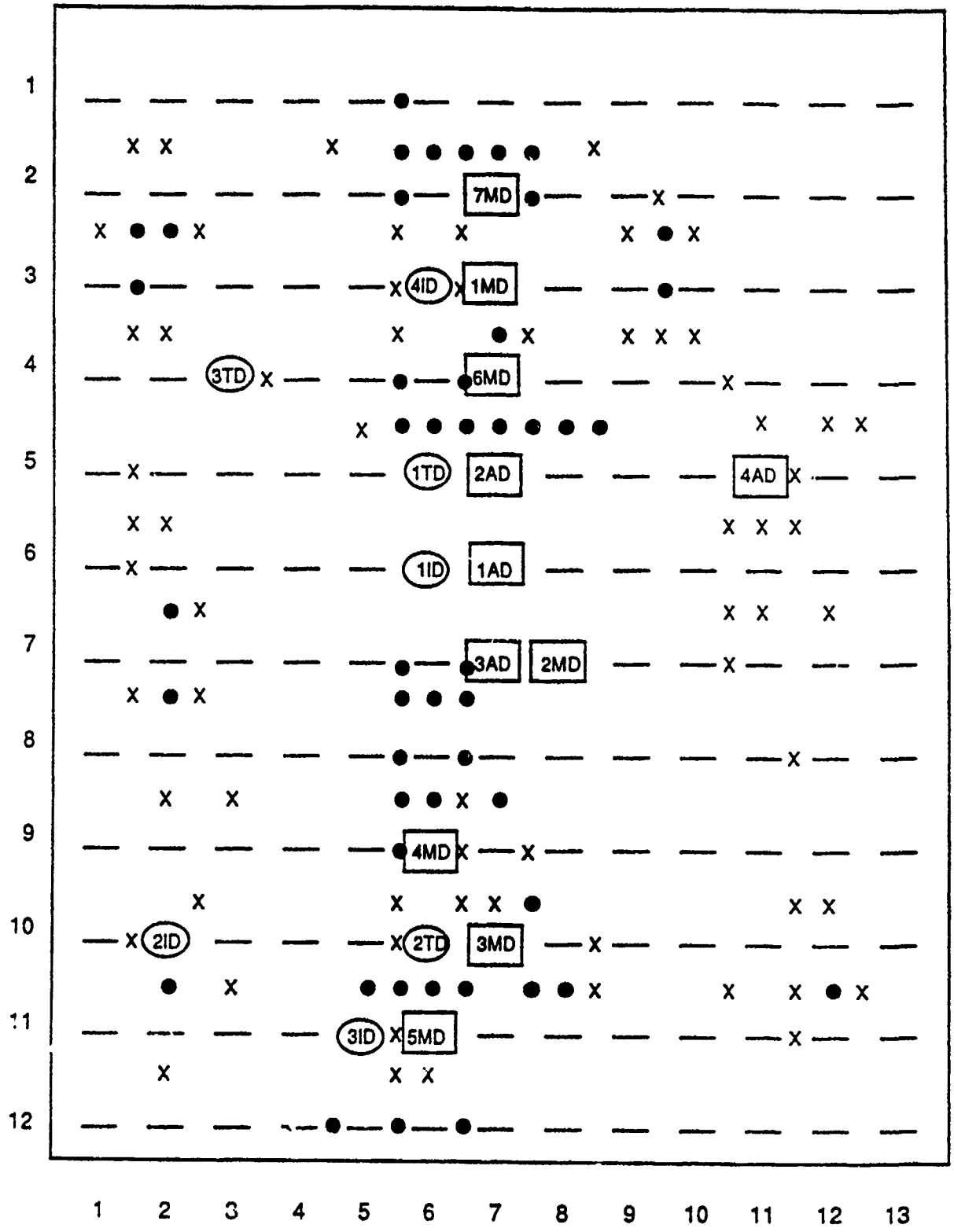
Storyboard #11



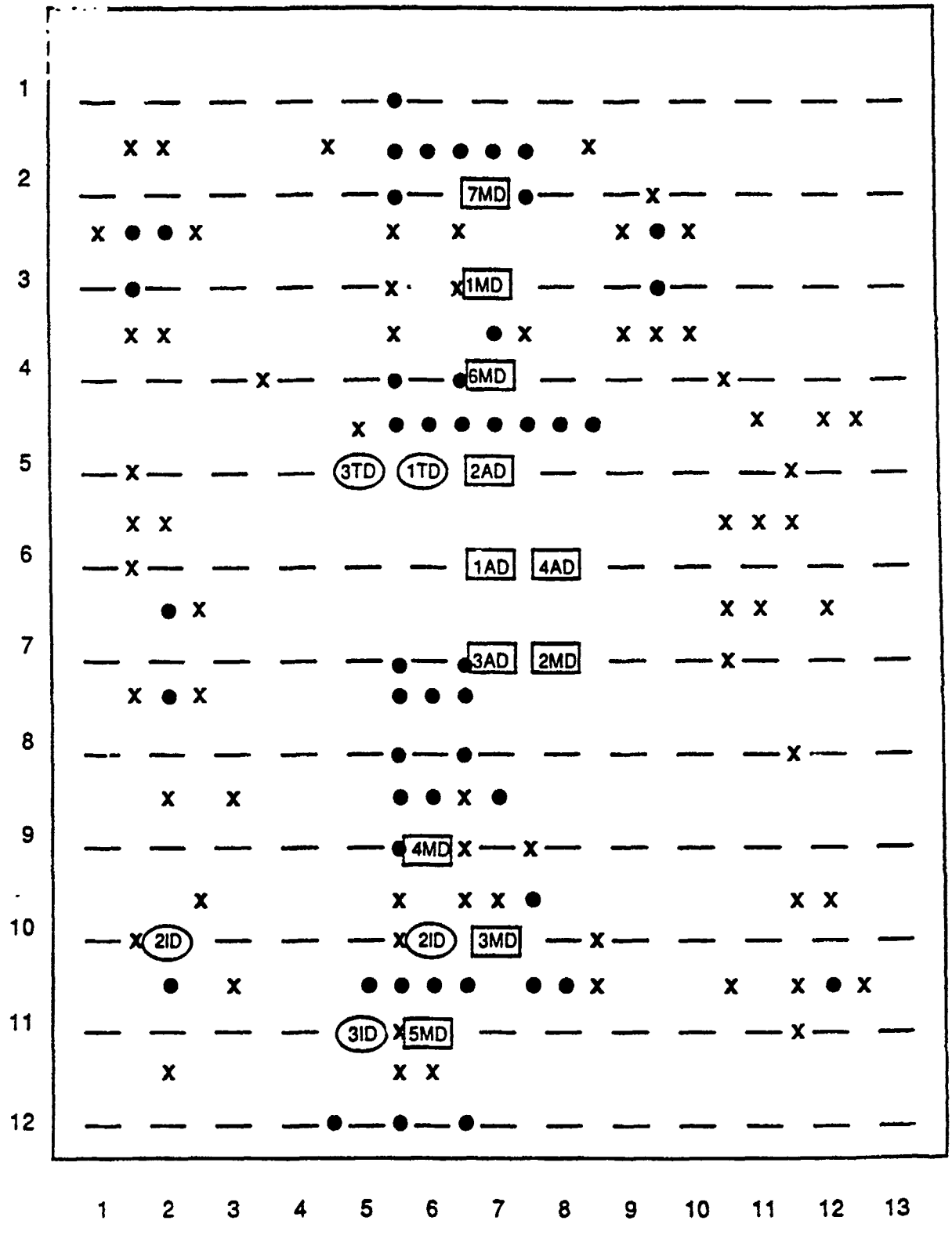
Storyboard #12



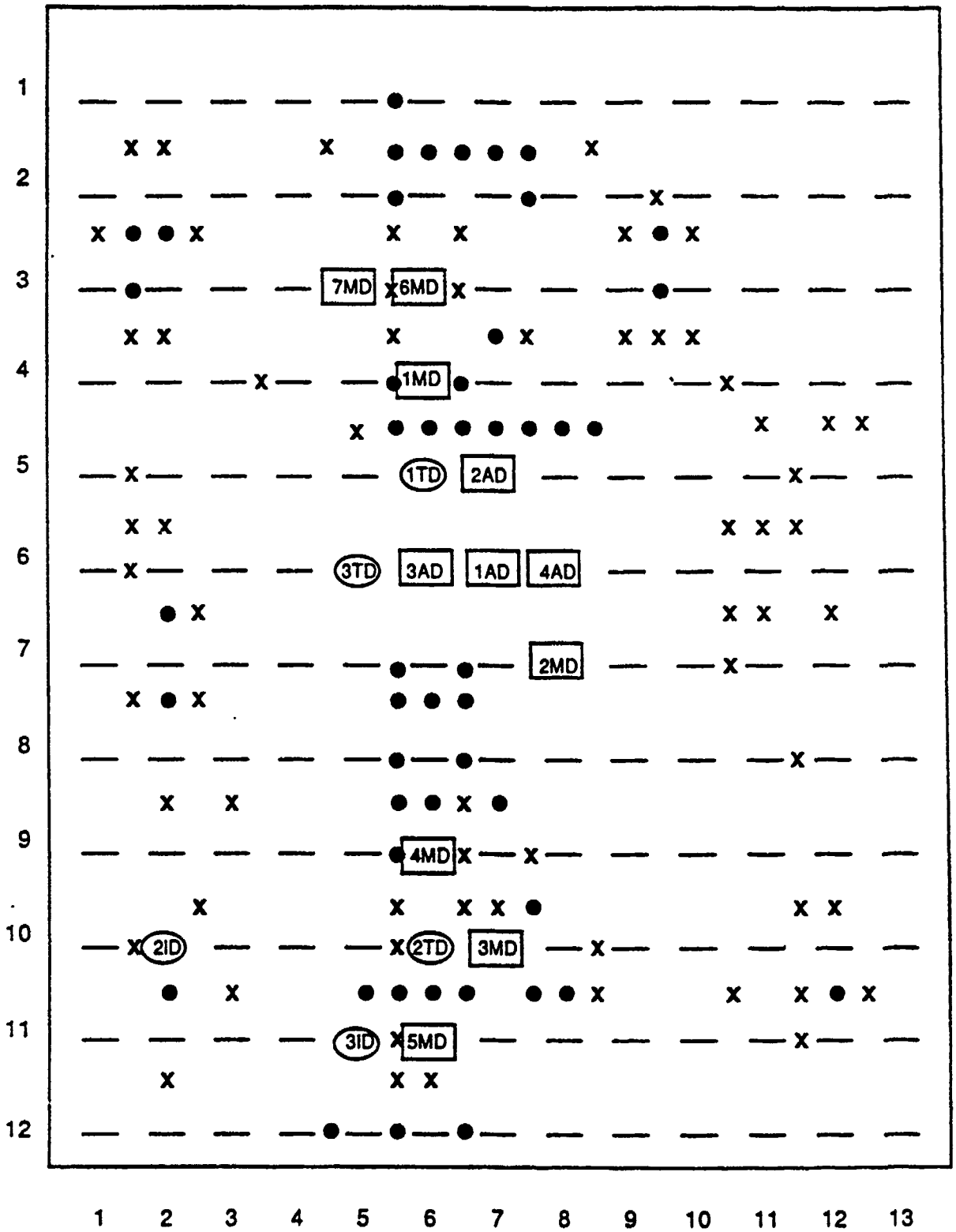
Storyboard #13



Storyboard #14



Storyboard #15



Storyboard #16

7. SUMMARY

The original goals of the Phase II effort were:

1. Develop a Plan Language for Pattern-directed Planning,
2. Develop a Plan Language for Robot Planning,
3. Investigate Adversarial Planning Issues in Robot Problem Solving,
4. Extend the planning system to multi-agent domains, and
5. Investigate approaches to interactive Planning.

All of these objectives have been achieved to a certain extent. The most significant success was the extension of the planning mechanism to the corps-maneuver problems, which involved both multiple agents (Goal #4) and interactive planning (Goal #5). Unfortunately, the researchers discovered that adversarial contingency planning is not as appropriate in low-level, reactive domains such as robotics as it is in higher-level, more-strategic environments. This is undoubtedly due to the fact that the search space examined by the planner, although a mere fraction of that examined by other planners, is yet sizable enough to require significant time for computation.

The efforts to develop a planning "language" were also successful in that a generic plan parser was defined which is capable of developing plans of action based on input goals which can represent actions in any pre-defined domain.

Possibilities for further research can best be broken down into two areas. First, the basic features of the planner's goal representation could easily be formalized into a grammar which would allow for easier processing and manipulation of success and failure tests. Currently, these tests are being added to long lists that develop as a particular course of action develops. Such a grammar would make it possible to make these lists more manageable and to reduce backtracking; deKleer (1985) suggests such a grammar.

A second possibility for further research is in the area of distributed or parallel planning. Currently, the planning process is understood in terms of a sequential linear model. Real-life planning in such domains as Command and Control, however, is conducted in parallel. The primary problem is the difficulty in knowing how to partition

planning bases so as to make them independent. If they are not treated as independent, knowing how changes in the situation affect different components is difficult. Essentially the problem is knowing what information is important to a planner working on some sub-problem.

REFERENCES

Berliner, Hans J., {Some Necessary Conditions For a Master Chess Program," Proc. Third International Joint Conference on Artificial Intelligence. Stanford University Press: Stanford, CA, 1973.

Berliner, Hans J., "Chess As Problem Solving," The Development of A Tactics Analyzer, Doctoral Dissertation, Carnegie Mellon University, 1975.

deKleer, Johan, "Choices Without Backtracking." Proc Ninth International Joint Conference on Artificial Intelligence, 1985.

Lehner, P.E., and McIntyre, James R., "Developing a General Contingency Planner for Adversarial Planning," PAR Technology Corporation Report 84-125.

McCarthy, John, and Hayes, Patrick J., "Some Philosophical Problems from the Standpoint of Artificial Intelligence," in Machine Intelligence 4, edited by Bernard Melzer and Donald Michie, Edinburgh Univ. Press, Edinburgh, Scotland, 1969.

Nilsson, Nils J., Principles of Artificial Intelligence. Tioga Publishing Company, Palo Alto, CA, 1980.

Reitment, W. and Wilcox, B. "Modeling Tactical Analysis and Problem Solving in Go," Procedures of the Tenth Annual Conference on Modeling and Simulation. 2133-2144, 1979.

Sacerdoti, Earl D., "Problem Solving Tactics," Proc. Sixth International Joint Conference on Artificial Intelligence. Tokyo, Japan, 1979.

Wilkins, D., "Using Patterns and Plans to Solve Problems and Control Search," Stanford Artificial Intelligence Laboratory Memo AIM-329, Stanford University, 1979.

APPENDIX A

PROGRAM 1

DEMONSTRATION INSTRUCTION

```
;FL(:BYTE-SIZE 8 :LENGTH-IN-BLOCKS 4 :LENGTH-IN-BYTES 3789 : JTHOR "WARGAME" :CREATI
3AME" :NAME "ADEMOINST" :TYPE "L" :VERSION 3)



;STRUCTIONS FOR RUNNING A DEMONSTRATION OF ARES FOR THE WARGAME
 -- SETTING UP THE DEMONSTRATION
    (1) in the lisp interpreter enter "(login 'wargame)"
    (2) then enter "(dired)", you should then see a listing
        of the top level directory for "wargame"
    (3) using the mouse (left button) select the file 'afinalgame.l'
    (4) then enter "E", this will load the editor, load the file
        into a buffer, and open the buffered file for editing
    (5) once in the editor enter "META-x", this will allow you
        to enter an extended command (look at window at bottom of
;F screen)
    (6) enter "compile buffer" and return, this will compile the present
        buffer into the lisp environment
    (7) after compilation is complete enter "META-CTRL-l", this will
        return you to the directory
    (8) repeat steps (3) to (7) above for the following files in order:
            'afinalterrain.l'
            'afinalgoaldef.l'
            'afinalgoals.l'
    (9) repeat steps (3) to (8) for the file  'afinalares.l'
   (10) after step (9) all necessary files for execution should be compiled
        into the lisp environment and you should still be in the buffer
        'afinalares.l'
   (11) while the arrow is in the main window hit the right button, when the
        menu appears thave the option 'Kill or Save Buffers', when in
        this option kill the buffers 'afinalgame.l', 'afinalgoals.l',
        'afinalgoaldef.l', and 'afinalterrain.l'.   While this step is
        optional, the planner has a tendency to overload virtual memory.
        Doing this step will avoid this problem during a demo!
   (12) while in the file 'afinalares.l' enter "BREAK", this will open up
        a window into the lisp environment
   (13) enter "(display realboard t)", this will clear the screen and display the
        present board position
   (14) enter "(retrieve_game 'ademogame.l)", this step is optional but will cut
        your demo down from two hours to 10 minutes, in particular this file
        contains the results of previous path finding problems making it unnessecar
        to wait while the system does path finding.
   (15) enter "(plan fgoal egoal t)", this will start the planner going
   (16) after each new board position, the planner will break.  you may find the
        following commands useful:
        (a) "RESUME"  --  this will exit the break and continue the planning sessior
        (b) "(display_orders 'hypboard S)" where S can be "friend" or "enemy"
              this will display the most recent orders to either side
        (c) "(display_unit_status 'hypboard U)" where U is any unit identifier
              this will display the present status of any unit
        (d) "(pprint (reverse (first cgt)))"  --  this will display all goals
              the planner processed to get to the present position for side friend
        (e) "(pprint (reverse (second cgt)))"  -- same as (d) for side enemy
```

(f) 'CLEARSCREEN' useful to do before (d) or (e)
(g) '(cursorpos 50)' moves the cursor just under the board
(h) '(display hypboard t)' -- redisplays the present hypothetical board position

(17) note that the planner is quite fast except when it does a backup, backups and restarts usually take 3-5 minutes (JIM we can cut this to a few seconds when we get back to work)
(18) when everything is finished enter 'ABORT', this will put you back in the 'afinalares.l' buffer
(19) enter 'SYSTEM-L' this will put you back in the top level lisp listener
(20) in the lisp listener enter '(logout)'
(21) enter '(si:%halt)'
(22) proceed to turn of the LMABDA machine

PROGRAM 2

TERRAIN

```
LMFL(:BYTE-SIZE 8 :LENGTH-IN-BLOCKS 11 :LENGTH-IN-BYTES 11151 :AUTHOR 'WARGAME' :CI
WARGAME' :NAME 'AFINALTERRAIN' :TYPE 'L' :VERSION 1)



; this file contains an example game
; a game includes a terrain_board and a set of units
; for each side
(defvar terrain_board nil)
; terrain_board will be a global variable that defines
; the board
; the following is how the terrain board is defined
(setq terrain_board
'((5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5
  (5 nil 1 nil 1 nil 1 nil 1 nil 5 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 5
  (5  1  2  2  1  1  1  1  2  1  5  5  5  5  5  1  2  1  1  1  1  1  1  1  1  1  5
  (5 nil 1 nil 1 nil 1 nil 1 nil 5 nil 1 nil 5 nil 1 nil 2 nil 1 nil 1 nil 1 nil 5
  (5  2  5  5  2  1  1  1  1  1  2  1  2  1  1  1  1  2  5  2  1  1  1  1  1  1  5
  (5 nil 5 nil 1 nil 1 nil 1 nil 2 nil 2 nil 1 nil 1 nil 1 nil 5 nil 1 nil 1 nil 1 nil 5
  (5  1  2  2  1  1  1  1  1  1  2  1  1  5  2  1  1  2  2  2  1  1  1  1  1  1  5
  (5 nil 1 nil 1 nil 2 nil 1 nil 5 nil 5 nil 1 nil 1 nil 1 nil 2 nil 1 nil 1 nil 5
  (5  1  1  1  1  1  1  1  1  2  5  5  5  5  5  5  5  1  1  1  1  2  1  2  2  1  5
  (5 nil 2 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 2 nil 1 nil 5
  (5  1  2  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  2  2  1  1  2  5
  (5 nil 2 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 5
  (5  1  1  5  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  2  1  2  1  1  5
  (5 nil 1 nil 1 nil 2 nil 1 nil 5 nil 5 nil 1 nil 1 nil 1 nil 2 nil 1 nil 1 nil 5
  (5  1  2  5  2  1  1  1  1  1  5  5  5  1  1  1  1  1  1  2  1  2  1  1  1  1  5
  (5 nil 1 nil 1 nil 1 nil 1 nil 5 nil 5 nil 1 nil 1 nil 1 nil 1 nil 1 nil 2 nil 1 nil 5
  (5  1  1  2  1  2  1  1  1  1  5  5  2  5  1  1  1  1  1  1  1  1  1  1  1  1  5
  (5 nil 1 nil 1 nil 1 nil 1 nil 5 nil 2 nil 2 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 5
  (5  1  1  1  2  1  1  1  1  1  2  1  2  2  5  1  1  1  1  1  1  1  2  2  1  1  5
  (5 nil 2 nil 1 nil 1 nil 1 nil 2 nil 1 nil 1 nil 2 nil 1 nil 1 nil 1 nil 1 nil 1 nil 5
  (5  1  1  5  1  2  1  1  1  5  5  5  5  1  5  5  2  1  1  1  2  1  2  5  2  1  5
  (5 nil 1 nil 1 nil 1 nil 1 nil 2 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 2 nil 1 nil 5
  (5  1  1  2  1  1  1  1  1  1  2  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  5
  (5 nil 1 nil 1 nil 1 nil 5 nil 5 nil 5 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 1 nil 5
  (5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5
; in additions to the terrain itself that are a number of features of a board
; that are looked at by the knowledge base
; these features are defined a properties below
; corridors of attack are defined below
; in the terrain board defined above there are four corridors
; for each corridor the following things are identified
;   defend_points lists of sets of defensive points for each side
;                 forward most defend points are listed first
;   paths lists the alternative paths of attack for
;         at present these paths are the same for both sides
;         but this should later be changed
;   general_area  is a list of all unit locations in the general area
;                 of the corridor
(putprop 'defend_points '( (friend ( ((5 3)) ((6 3))))
   (enemy ( ((7 3)) ((6 3))))))
```

```
 'corridor1)
(putprop 'paths '( ((5 3) (6 4) (7 3)) ((5 3) (6 2) (7 2))
   ((5 3) (6 3) (7 4)))
 'corridor1)
(putprop 'general_area '( (4 2) (4 3) (4 4) (5 2) (5 3) (5 4) (6 2) (6 3) (6 4)
  (7 2) (7 3) (7 4) (8 2) (8 3) (8 4))
 'corridor1)
(putprop 'center_point '(6 3) 'corridor1)
(putprop 'defend_points '( (friend ( ((5 5) (5 6)) ((6 5) (5 6)) ((5 5) (6 6))
     ((6 5) (6 6)) ) )
   (enemy  ( ((7 6) (8 5)) ((7 6) (7 5)) ((7 6) (6 5)) ) ))
 'corridor2)
(putprop 'paths '( ((5 5) (6 5) (7 5)) ((5 6) (6 7) (7 6)) ((5 6) (6 6) (7 7)))
 'corridor2)
(putprop 'general_area '( (4 5) (4 6) (4 7) (5 5) (5 6) (5 7)
  (6 5) (6 6) (6 7) (7 5) (7 6) (7 7)
  (8 5) (8 6) (8 7) (9 5) (9 6) (9 7) )
 'corridor2)
(putprop 'center_point '(6 6) 'corridor2)
(putprop 'defend_points '( 'friend ( ((5 10)) ((6 10)) ) )
   (enemy  ( ((7 9) (7 10)) ((6 9) (7 10)) ((6 10)) ) ))
 'corridor3)
(putprop 'paths '( ((5 10) (6 9) (7 8)) ((5 9) (6 9) (7 9))
   ((5 10) (6 10) (7 10)))
 'corridor3)
(putprop 'general_area '( (4 8) (5 8) (6 8) (7 8) (8 8)
  (4 9) (5 9) (6 9) (7 9) (8 9)
  (4 10) (5 10) (6 10) (7 10) (8 10) )
 'corridor3)
(putprop 'center_point '(6 9) 'corridor3)
(putprop 'defend_points '( (friend ( ((4 11)) ((5 11)) ((6 11)) ) )
   (enemy  ( ((7 11)) ((6 11)) ((5 11)) ) ) )
 'corridor4)
(putprop 'paths '( ((4 11) (5 11) (6 12) (7 11)) ((4 11) (5 12) (6 11) (7 12)))
 'corridor4)
(putprop 'general_area '( (4 11) (5 11) (6 11) (7 11) (8 11)
  (4.12) (5 12) (6 12) (7 12) (8 12) )
 'corridor4)
(putprop 'center_point '(6 11) 'corridor4)
; for units that must backup the attack or defense of two or more corridors
; support areas are defined
; these support areas represent where these units
; should locate before moving into one of the supported corridors
;separate support_areas are defined for the friendly and enemy side
(putprop 'support_areas '(((corridor1 corridor2)
   ( (2 4) (1 3) (1 4) (1 5) (2 3) (2 4) (2 5) (3 3) (3 4) (3 5) ))
  ((corridor2 corridor3)
   ( (2 7) (1 6) (1 7) (1 8) (2 6) (2 7) (2 8) (3 6) (3 7) (3 8) ))
  ((corridor3 corridor4)
   ( (2 10) (1 9) (1 10) (1 11) (2 9)
     (2 10) (2 11) (3 9) (3 10) (3 11) ))
  ((corridor2 corridor3 corridor4)
   ( (2 8) (2 9) (3 8) (3 9) ))
  ((corridor1 corridor2 corridor3)
   ( (2 5) (2 6) (3 5) (3 6) ) )
  )
```

```
 'friend)
(putprop 'support_areas '(((corridor1 corridor2)
   ( (11 5) (10 4) (10 5) (10 6) (11 4) (11 5) (11 6)
     (12 4) (12 5) (12 6) ))
  ((corridor2 corridor3)
   ( (10 7) (9 6) (9 7) (9 8) (10 6) (10 7) (10 8)
     (11 6) (11 7) (11 8) ))
  ((corridor3 corridor4)
   ( (11 11) (10 10) (10 11) (10 12) (11 10) (11 11) (11 12)
     (12 10) (12 11) (12 12) ))
  ((corridor1 corridor2 corridor3)
   ( (10 5) (10 6) (11 5) (11 6) ))
  ((corridor2 corridor3 corridor4)
   ( (10 8) (10 9) (11 8) (11 9) ) )
  )
 'enemy)
; once the terrain board is specified then units can be placed on the board
; to defin a starting position this is done below
; in effect the following is an example of an initial set up
(setq realboard terrain_board)
(putprop 'time 0 'realboard)
(setq friendly_units nil)
(setq enemy_units nil)
(setq all_units nil)
;in the unit defintions below
;   define_unit sets up the property lists that define each unit
;   put_unit_on_board actual puts the unit on the realboard
(define_unit '1AD 'enemy '1st 'armour 'division 12 9 8.5 4 '(8 6) t)
(put_unit_on_board 'realboard '1AD '(8 6))
(define_unit '2AD 'enemy '2nd 'armour 'division 12 9 8.5 4 '(8 5) t)
(put_unit_on_board 'realboard '2AD '(8 5))
(define_unit '3AD 'enemy '3rd 'armour 'division 12 9 8.5 4 '(8 7) t)
(put_unit_on_board 'realboard '3AD '(8 7))
(define_unit '4AD 'enemy '4th 'armour 'division 12 9 8.5 4 '(9 6) t)
(put_unit_on_board 'realboard '4AD '(9 6))
(define_unit '1MD 'enemy '1st 'infantry 'division 10 9 6 4 '(9 5) t)
(put_unit_on_board 'realboard '1MD '(9 5))
(define_unit '2MD 'enemy '2nd 'infantry 'division 10 9 6 4 '(10 7) t)
(put_unit_on_board 'realboard '2MD '(10 7))
(define_unit '3MD 'enemy '3rd 'infantry 'division 10 9 6 4 '(9 7) t)
(put_unit_on_board 'realboard '3MD '(9 7))
(define_unit '4MD 'enemy '3rd 'infantry 'division 10 9 6 4 '(9 9) t)
(put_unit_on_board 'realboard '4MD '(9 9))
(define_unit '5MD 'enemy '3rd 'infantry 'division 10 9 6 4 '(7 11) t)
(put_unit_on_board 'realboard '5MD '(7 11))
(define_unit '6MD 'enemy '3rd 'infantry 'division 10 9 6 4 '(8 3) t)
(put_unit_on_board 'realboard '6MD '(8 3))
(define_unit '7MD 'enemy '3rd 'infantry 'division 10 9 6 4 '(8 2) t)
(put_unit_on_board 'realboard '7MD '(8 2))
(define_unit '1CR 'friend '1st 'cavalry 'regiment 5 5 7 4 '(6 6) t)
(put_unit_on_board 'realboard '1CR '(6 6))
(define_unit '1TD 'friend '1st 'tank 'division 10 9 7 4 '(3 6) t)
(put_unit_on_board 'realboard '1TD '(3 6))
(define_unit '2TD 'friend '2nd 'tank 'division 10 9 7 4 '(4 10) t)
(put_unit_on_board 'realboard '2TD '(4 10))
(define_unit '3TD 'friend '3rd 'tank 'division 10 9 7 4 '(3 4) t)
```

```
(put_unit_on_board 'realboard '3ID '(3 4))
(define_unit '2ID 'friend '2nd 'infantry 'division 8 10 7 4 '(4 7) t)
(put_unit_on_board 'realboard '2ID '(4 7))
(define_unit '1ID 'friend '1st 'infantry 'division 8 10 7 4 '(4 5) t)
(put_unit_on_board 'realboard '1ID '(4 5))
(define_unit '3ID 'friend '3rd 'infantry 'division 8 10 7 4 '(4 11) t)
(put_unit_on_board 'realboard '3ID '(4 11))
(define_unit '4ID 'friend '4th 'infantry 'division 8 10 7 4 '(5 3) t)
(put_unit_on_board 'realboard '4ID '(5 3))
; at this point the entire board has been defined
; the following is a list of top level goals
; if you understand how the goal defintions work you
; should be able to pick up the meaning of these goals
(setq esgoal1 '(attack_corridors (((corridor1 (7md 6md 1md 2md))
  (corridor3 (3md 4md 3ad 4ad)))
  5)))
(setq esgoal2 '(defend_corridors (((corridor2 (1ad 2ad)) (corridor4 (5md)))
  7)))
(setq egoal1 (list 'andsim (list esgoal1 esgoal2)))
(setq esgoal3 '(attack_corridors (((corridor2 (1ad 2ad 3ad 4ad 1md 3md 6md)))
  5)))
(setq esgoal4 '(defend_corridors (((corridor1 (7md)) (corridor3 (4md 2md)) (corrido
  7)))
(setq egoal2 (list 'andsim (list esgoal3 esgoal4)))
(setq esgoal5 '(attack_corridors (((corridor2 (1ad 2ad 4ad))
   (corridor3 (3ad 4md 3md))) 5)))
(setq esgoal6 '(defend_corridors (((corridor1 (6md)) (corridor4 (5md)))
  7)))
(setq esgoal7 '(support_plan ( ( ( ((corridor1 defend) (corridor2 attack)) (7md))
 ( ((corridor2 attack) (corridor3 attack)) (1md))
 ( ((corridor3 attack) (corridor4 defend)) (2md)))
      2)))
(setq egoal3 (list 'andsim (list esgoal5 esgoal6 esgoal7)))
(setq esgoal8 '(attack_corridors (((corridor1 (6md 7md 1md))
   (corridor2 (1ad 2ad 3ad)))
  5)))
(setq esgoal9 '(defend_corridors (((corridor3 (4md 3md)) (corridor4 (5md)))
  7)))
(setq esgoal10 '(support_plan ( ( ( ((corridor1 attack) (corridor2 attack)) (2md 4
      2)))
(setq egoal4 (list 'andsim (list esgoal8 esgoal9 esgoal10)))
(setq egoal (list 'or (list egoal2 egoal3)))
(setq fgoal1 '(defend_corridors (((corridor1 (4id 3td)) (corridor2 (1id 2id 1cr))
  (corridor3 (1td 2td)) (corridor4 (3id))) 7)))
(setq fsgoal1 '(defend_corridors (((corridor1 (4id)) (corridor2 (1id 1cr))

  (corridor3 (2td)) (corridor4 (3id))) 7)))
(setq fsgoal2 '(support_plan ( ( ( ((corridor1 defend) (corridor2 defend)) (3td) )
        ( ((corridor2 defend) (corridor3 defend)) (1td) )
 ( ((corridor3 defend) (corridor4 defend)) (2id) ))
     2)))
(setq fgoal2 (list 'andsim (list fsgoal1 fsgoal2)))
(setq fsgoal3
      '(support_plan
 ( ( ( ((corridor1 defend) (corridor2 defend) (corridor3 defend)) (3td) )
     ( ((corridor2 defend) (corridor3 defend) (corridor4 defend)) (1td 2id) ))
```

```
   ))')
(setq 'goal3 (list 'andsim (list fsgoal1 fsgoal3)))
(setq fgoal (list 'or (list fgoal2 fgoal3)))
```

PROGRAM 3

GOAL DEFINITION PARAMETERS

```
LMFL(:BYTE-SIZE 8 :LENGTH-IN-BLOCKS 2 :LENGTH-IN-BYTES 1752 :AUTHOR 'WARGAME' :CRE
RGAME' :NAME 'AFINALGOALDEF' :TYPE 'L' :VERSION 1)



; these are the parameters for all goal definition
; at present procedures used in a goal definition can use no
; other parameters  --  this is bad and should be replaced with
; macros someday
(defvar gtmpgoal nil)
; this is the goal name
(defvar gtmpargs nil)
; all arguements of the goal name are in this list
(defvar gtmpbname nil)
; this specifies the board name
(defvar gtmpside nil)
; this is the side being processed
; makegoal
; a simple routine that puts specifed lisp code on property
; list of a goal
(defun makegoal (call)
  (putprop 'type 'specific (car call))
  (putprop 'subgoal nil (car call))
  (putprop 'countergoal 'universe (car call))
  (putprop 'feasible t (car call))
  (putprop 'succeeded_if nil (car call))
  (putprop 'failed_if nil (car call))
  (putprop 'dont_continue_if nil (car call))
  ; the above putprops set default values
  ; the prog below replaces default values with values
  ; specified in goal definition
  (prog (gname cur lst)
(setq gname (car call) lst (cddr call))
(putprop 'args (cadr call) gname)
loop1
(setq cur (car lst) lst (cdr lst))
(and cur (putprop (car cur) (cadr cur) gname))
(and lst (go loop1))
(return gname)))
; process_goal
; retrieves specific property and evals it after setting up
; necessary globals
; this is done by setting up the four goal definition arguements
; and processing type_process
(defun process_goal (bname goal side type_process)
(setq gtmpgoal (car goal) gtmpargs (cadr goal)
      gtmpbname bname gtmpside side)
(eval (get type_process (car goal)))
)
; Note this approach to defining goals should be replaced with
; a sophisticated goal description language
; an important question is however how much such a language should be
; domain dependent
```

PROGRAM 4

GOAL DEFINITION STRUCTURE

```
LMFL(:BYTE-SIZE 8 :LENGTH-IN-BLOCKS 35 :LENGTH-IN-BYTES 35587 :AUTHOR 'WARGAME' :CRI
WARGAME' :NAME 'AFINALGOALS' :TYPE 'L' :VERSION 1)



; in finalgame.l
; the goal definition which is a frame-like representation
; has the following structure
;(makegoal
;  '(goalname (arguements)
;       (type ********)
;       (subgoal ********)
;       (countergoal **********)
;       (feasible_if **********)
;       (succeeded_if ********)
;       (failed_if **********)
;       (dont_continue_if **********)
; these elements of the goal structure can be interpreted as follows
; goalname - is the name of the goal -- it must be a unique atom
; arguements - is a list of the arguements used by the goal
;              when a goal definition is being processed these arguements are
;              contained in the global variable gtmpargs
; type - allows characterization of the goal
;        not used now but will be useful in future versions --
;        default is specific
; subgoal - the eval of the contents of the subgoal slot must
;           evaluate to a list of subgoals -- default is nil
; countergoal - the eval of the contents of the countergoal slot must
;               eval to a list of acceptable countergoals
;               this slot is not used for the wargame because but
;               is very useful for single move-countermove games such
;               as chess  go  othello -- default is 'universe
; feasible - a procedure that should eval to t or nil
;            to indicate whether it is feasible to pursue the
;            goal in the present situation  --  default is t
; succeeded_if - a procedure that should eval to t or nil
;                to indicate whether the goal has been achieved
;                in the present situation -- default is nil
; failed_if - a procedure that should eval to t or nil
;             to indicate whether the goal has failed
;             in the present situation -- default is nil
; dont_continue_if - a procedure that should eval to t or nil
;                    to indicate whether the goal has become irrelevant
;                    but cannot be marked as succeeded or failed -- default is nil
; GENERIC GOALS
; the following are generic goals that can be used in any domain
; they represent knowledge about goals that is independent of knowledge
; about a domain
; andsim   generic
; subgoal is andsim of subgoal of the goals in its arguements
; failed_if any one subgoal is a failure
; succeeded_if all of its subgoals succeed
; dont_continue_if not all but at least one subgoal succeeds
```

```
(makegoal
  '(andsim (goals)

     (type generic)
     ;subgoal is a list that equals all possible combinations of all
     ;subgoals of the goals in the andsim arguement
     (subgoal
       (prog (tmp1 tmp2 tmpbname tmpside)
     (setq tmp1 gtmpargs tmpbname gtmpbname tmpside gtmpside)
   loop1
     (setq tmp2 (cons (process_goal tmpbname (car tmp1) tmpside 'subgo ) tmp2))
     (and (setq tmp1 (cdr tmp1)) (go loop1))
     (setq tmp2 (remove 'no_longer_relevant tmp2))
     (and (null tmp2) (return 'no_longer_relevant))
     ; loop1 gets all subgoals for each goal
     (setq tmp1 (all_combinations tmp2) tmp2 nil)
     ; tmp1 is now list of all combinations of subgo l's
   loop2
     (setq tmp2 (cons (list 'andsim 'car tmp1)) tmp2))
     (and (setq tmp1 (cdr tmp1)) (go loop2))
     ; now each list of subgoals is andsim subgoal
     (return tmp2)
     ; list of andsims is now returned
     ))
     ;if all of the component goals of the andsim succeed then the andsim has succe
     (succeeded_if
       (prog (tmp1 tmpbname tmpside rslt)
     (setq tmp1 gtmpargs tmpbname gtmpbname tmpside gtmpside)
   loop1
     (setq rslt (process_goal tmpbname (car tmp1) tmpside 'succeeded_if))
     (and (null rslt) (return nil))
     (and (setq tmp1 (cdr tmp1)) (go loop1))
     (return (list 'succeeded tmpside))))
     ;if just one subgoal of the component goals has failed then the andsim has fai
     (failed_if
       (prog (tmp1 tmpbname tmpside rslt)
     (setq tmp1 gtmpargs tmpbname gtmpbname tmpside gtmpside)
   loop1
     (setq rslt (process_goal tmpbname (car tmp1) tmpside 'failed_if))
     (and rslt (return (list 'failed tmpside (car tmp1) rslt)))
     (and (setq tmp1 (cdr tmp1)) (go loop1))))
     ;if any of the component subgoals of the andsim should not be continued then
     ;dont continue the andsim
     (dont_continue_if
       (prog (tmp1 tmpbname tmpside rslt)
     (setq tmp1 gtmpargs tmpbname gtmpbname tmpside gtmpside)
   loop1
     (setq rslt (process_goal tmpbname (car tmp1) tmpside 'dont_continue_if))
     '(and (null rslt)
   (setq rslt (process_goal tmpbname (car tmp1) tmpside 'succeeded_if)))
     (and rslt (return 'list 'dont_continue_if tmpside 'car tmp1) rslt)))
     (and (setq tmp1 (cdr tmp1)) (go loop1))))
     )
   )
; orsim   generic
; pursues multiple simultaneous goals and fails if all goals fai
```

```
; succeeded_if any one subgoal is a success
; failed_if all of its subgoals failed
; dont_continue_if not all but at least one subgoal fails
(makegoal
  '(orsim (goals)

     (type generic)
     ;subgoal evals to a list of all combinations of all subgoals
     :of the component goals of goals
     (subgoal
       (prog (tmpl tmp2 tmpbname tmpside)
     (setq tmpl gtmpargs tmpbname gtmpbname tmpside gtmpside)
  loop1
     (setq tmp2 (cons (process_goal tmpbname (car tmp1) tmpside 'subgoal, tmp2))
     'and (setq tmpl (cdr tmpl)) (go loop1))
     (setq tmp2 (remove 'no_longer_relevant tmp2))
     ; loop1 gets all subgoals for each goal

     (setq tmpl (all_combinations tmp2) tmp2 nil)
     ; tmpl is now list of all combinations of subgoals

  loop2
     (setq tmp2 (cons (list 'orsim (car tmpl)) tmp2))
     (and (setq tmpl (cdr tmpl)) (go loop2))
     ; now each list of subgoals is andsim subgoal
     (return tmp2)
     ; list of orsims is now returned
     ))
     :succeeded_if any of the component goals have succeeded
     (succeeded_if
       (prog (tmpl tmpbname tmpside rslt)
     (setq tmpl gtmpargs tmpbname gtmpbname tmpside gtmpside)
  loop1
     (setq rslt (process_goal tmpbname (car tmpl) tmpside 'succeeded_if),
     (and rslt (return (list 'succeeded tmpside (car tmpl) rslt)))
     (and (setq tmpl (cdr tmpl)) (go loop1))))
     ;failed if all of the component goals have failed
     (failed_if
       (prog (tmpl tmpbname tmpside rslt)
     (setq tmpl gtmpargs tmpbname gtmpbname tmpside gtmpside)
  loop1
     (setq rslt (process_goal tmpbname (car tmpl) tmpside 'failed_if))
     (and  (equal rslt t)  (return (list 'failed tmpside (car tmpl))))
     (and rslt (return rslt))
     (and (setq tmpl (cdr tmpl)) (go loop1))
     (return nil)))
     ;dont_continue_if any of the component subgoals should not be continued
     ;or if any of the component goals have failed
     (dont_continue_if
       (prog (tmpl tmpbname tmpside rslt)
     (setq tmpl gtmpargs tmpbname gtmpbname tmpside gtmpside)
  loop1
     (setq rslt (process_goal tmpbname (car tmpl) tmpside 'dont_continue_if))
     '(and (null rslt)
  (setq rslt (process_goal tmpbname (car tmpl) tmpside 'failed_if)))
     (and rslt (return (list 'dont_continue_if tmpside (car tmpl) rslt)))
```

```
      (and (setq tmpl (cdr tmpl)) (go loop1))))
      )
  )
; or
; logical or of goals
(makegoal
  '(or (goals)
     (type generic)
     (subgoal gtmpargs)

     )
  )
; DOMAIN SPECIFIC GOALS
; the following goals are unique to this wargame
; NOTE that these goals do not make use of boardval routines
; (although some supporting utilities do) consequently
; if there is an occassional board conflict with two units on
; the same position (can only occur by an error in backup)
; then planner will still continue without error
; defend_corridors
; goal that specifies corridors to be defended and the units
; to defend with
(makegoal
  '(defend_corridors (list_of_c_and_unit_list by_time)

     (type specific)

     (subgoal
       (prog (tmp1 tmp2)
     (setq tmp1 (first gtmpargs))
  loop1
     (and tmp1
  (setq tmp2 (cons (list 'defend_1_corridor
 (list (caar tmp1) (cadar tmp1)
       (second gtmpargs))) tmp2)))
     (and (setq tmp1 (cdr tmp1)) (go loop1))
     ;above loop decomposes arguements into component goals
     (cond '((> (length tmp2) 1) (return (list (list 'andsim tmp2))))
   ((= (length tmp2) 1) (return tmp2))))
            ;if more than one subgoal make andsim subgoal
     )
     )
  )
(makegoal
  '(defend_1_corridor (corridor list_of_units by_time)

     (type specific)

     (subgoal
       (prog (tmp1 tmp2 tmp3 tmp4 tmp5)
     (setq tmp1 (get_defend_points gtmpbname (car gtmpargs) gtmpside)
   tmp3 (second gtmpargs)
   tmp4 (union
  (second gtmpargs)
  (units_in_area gtmpbname
 (get 'general_area (first gtmpargs))
```

```
gtmpside)))
    ;tmp1 is list of locations to defend
    ;tmp3 is list of units available
    ;tmp4 is tmp3 plus list of units already there

    (and (> (length tmp1) (length tmp4)) (return nil))
    ;if not enough units assigned then failure
    (setq tmp5 tmp3)
    ;tmp5 is initially set to list of all units
  loop0
    (and (member (unit_status gtmpbname (car tmp5) 'location)
  tmp1)
 ((setq tmp2
(cons
  (list
    'defend_location
    (list (car tmp5)
  (unit_status gtmpbname (car tmp5) 'location)))
  tmp2))
  (setq tmp3 (remove (caadar tmp2) tmp3)))
    (and (setq tmp5 (cdr tmp5)) (go loop0))
    ;loop0 checks all units in list_of_units to determine if
    ;unit is already on a defend_point -- if yes then it stays
    ;and defends that location
  loop1
    (and tmp1 tmp3
  (setq tmp2
(cons (list 'defend_location
    (list (closest_unit gtmpbname (car tmp1) tmp3)
  (car tmp1)))
     tmp2)))
    (setq tmp1 (cdr tmp1) tmp3 (remove (caadar tmp2) tmp3))
    (and tmp1 (go loop1))

    (and tmp3
 (setq tmp1
(get_defend_points gtmpbname (car gtmpargs)
   gtmpside)) (go loop1))
            ;loop1 goes through all the units not already on a defend point
            ;and assigns each one to a specific
    ;location that it should move toward and defend
    (return (cond ((> (length tmp2) 1) (list (list 'andsim tmp2)))
   ((= (length tmp2) 1) tmp2)))
    ))
    ;failed_if cannot find defend points   This only happens if
    ;enemy has already broken through
    (failed_if
      (prog (tmp1)
    (setq tmp1 (get_defend_points gtmpbname (first gtmpargs) gtmpside))
    (cond ((null tmp1) (return t))
   (t (return nil)))
    ))
    ;if get past time by_time without failing then has succeeded
    (succeeded_if (> (get 'time gtmpbname) (third gtmpargs)))

    )
```

```
  )
; defend_location
; this goal is to move to a specified location and the defend it
(makegoal
  '(defend_location (unitname location)

     (type specific)

     (subgoal
       (cond
 ((equal (unit_status gtmpbname (first gtmpargs) 'location) (second gtmpargs))
  (list (list 'send (list (list (car gtmpargs) '(defend_in_place))))))
 (t (list (list 'send (list
'list (car gtmpargs)
      (cons 'move (get_path gtmpbname (first gtmpargs)
    (second gtmpargs) 6 t)))))))))
     :if at location stay there else move to location

     (dont_continue_if
       (prog (tmp1)
     (setq tmp1 (boardval (eval gtmpbname) (second gtmpargs)))
     (and tmp1 (setq tmp1 (unit_status gtmpbname tmp1 'side)))
     (cond ((equal tmp1 (opposite_side gtmpside))
    (return t))
   (t (return nil))))
       )
     ;dont_continue_if other side occupies location
     )
  )
; attack_corridors
; assigns a set of units to a set of corridors for an attack
(makegoal
  '(attack_corridors (list_of_c_and_unit_list by_time)

     (type specific)

     (subgoal
       (prog (tmp1 tmp2)
     (setq tmp1 (first gtmpargs))
  loop1
     (and tmp1
  (setq tmp2 (cons (list 'attack_1_corridor
 (list (caar tmp1) (cadar tmp1)
       (second gtmpargs))) tmp2)))
     (and (setq tmp1 (cdr tmp1)) (go loop1))
     (return (list (list 'orsim tmp2)))))
     )
  )
(makegoal
  '(attack_1_corridor (corridor list_of_units by_time)
      (type specific)
      (subgoal
(prog (tmp1 tmp2 tmp3 tmp4)
      (setq tmp1 (paths_in_corridor gtmpbname
    (first gtmpargs) gtmpside)
    tmp2 (second gtmpargs))
```

```
   loop1
      (setq tmp3 (best_unit_for gtmpbname (car tmp1) tmp2))
      ;tmp3 is the preferred unit to go down path
      (setq tmp4 (cons (list 'attack_down_path
     (list tmp3 (car tmp1) (first gtmpargs))) tmp4))
      ;add goal of attack down path car tmp1  with unit tmp3
      (setq tmp1 (reverse (cons (car tmp1) (reverse (cdr tmp1)))))
      ;move path to end of path list
      (setq tmp2 (remove tmp3 tmp2))
      ;remove unit tmp3 from list of units tmp2
      (and tmp2 (go loop1))
      ;as long as there are units left assign them
      (cond ((> (length tmp4) 1)
     (setq tmp4 (list (list 'andsim tmp4))))
    ((= (length tmp4) 1)
     (setq tmp4 tmp4)))
      ;subgoal is andsim of several attacks or just one attack with no andsim
      (return tmp4)))
      (succeeded_if
(prog (tmp1)
      (setq tmp1 (get_defend_points gtmpbname (first gtmpargs) (opposite_side gtmps
      (cond ((null tmp1) (return t))
    (t (return nil)))
      ))
      (failed_if (> (get 'time gtmpbname) (third gtmpargs)))

      ))
; attack_down_path
; identifies specific path of attack through corridor
(makegoal
  '(attack_down_path (unit path corridor)
      (type specific)
      (subgoal
(prog (tmp1 tmp2)
      (setq tmp1 (first_defend_unit gtmpbname (third gtmpargs)
    (opposite_side gtmpside)))
      ;identify first unit defending corridor
      (and (can_attack_unit gtmpbname (first gtmpargs) tmp1)
   (return (list (list 'send
       (list (list (first gtmpargs)
   (list 'attack tmp1)))))))
      ;attack any units in the way
      (and (setq tmp2 (member 'unit_status gtmpbname (first gtmpargs) 'location)
      (second gtmpargs)))
   (return (list (list 'send (list (list (first gtmpargs)
 (cons 'move (cdr tmp2)))))))
      ;if can not attack and member of path then march down path if can
      (return (list
(list 'send (list (list (first gtmpargs)
(cons 'move (append
      (get_path gtmpbname (first gtmpargs)
(car (second gtmpargs))
6 t)
      (cdr (second gtmpargs)))))))))
      ;if nothing else move unit toward beginning of path and march
      ;down path
```

```
      ))
      (succeeded_if
(equal (unit_status gtmpbname (first gtmpargs) 'location)
       (car (reverse (second gtmpargs)))))
      ;succeeded_if made it to the other end
      ))
; support plan
; assigns units to support positions until corridor
; conflict is near resolution then supports
(makegoal
  '(support_plan (list_of_corridors_and_units tempo)
     (type specific)
     (subgoal
       (prog (tmp1 tmp2)
     (setq tmp1 (first gtmpargs))
  loop1
     'setq tmp2
   'cons
     (list 'support_corridors
   (list (first (first tmp1))
 (second (first tmp1))
 (+ (second gtmpargs) (get 'time gtmpbname)))) tmp2))
     (and (setq tmp1 (cdr tmp1)) 'go loop1))
     (return (list (list 'andsim tmp2)))
     )
       .
     )
  )
; support corridors
; specific support corridor goal
; at present makes use of a global variable called supported_corridor that is
; set to nil on newturn  sorry bad form
(makegoal
  '(support_corridors (list_corridors_atkdef units wait_until_time)
     (type specific)
     (subgoal
       (prog (tmp1 tmp2 tmp3 tmp4 subgoals in_corridor)
     (setq tmp1 (first gtmpargs) tmp2 (second gtmpargs) tmp3 tmp2 tmp4 tmp1)
     ;tmp1 and tmp4 list of corridors
     ;tmp2 and tmp3 list of units
  loop0
     (cond ((and (null (member (unit_status gtmpbname
    (first tmp3) 'location)
       (support_area tmp1 gtmpside)))
 ;if unit not in support area
 (member (car tmp3) (active_units gtmpside))
 ;   unit is alive
 (equal_intersect
   (unit_status gtmpbname
(first tmp3) 'retreat_direction)
   (support_area tmp1 gtmpside)))
        ;   unit previously was in support area
    (setq in_corridor
  (which_corridor? gtmpbname (first gtmpargs)
   (first tmp3)))
    ;then identify which corridor it is now supporting
```

```
   (and (equal (second in_corridor) 'defend)
(setq subgoals
      (cons (list 'defend_1_corridor
  (list (first in_corridor)
(list (first tmp3))
100))
    subgoals)))
   ;and defend that corridor
   (and (equal (second in_corridor) 'attack)
(setq subgoals
      (cons (list 'attack_1_corridor
  (list (first in_corridor)
(list (first tmp3))
100))
    subgoals)))
   ;or attack that corridor
   (setq tmp2 (remove (car tmp3) tmp2))
   ;remove the unit from list of units to consider for new assignments
   ))
    (and (setq tmp3 (cdr tmp3)) (go loop0))
    ;subgoals now has all previous subgoals included
    ;tmp2 is now a list of all other units to assign
    ;tmpl is still list of corridors
    (and (null tmp2) (go loop2))
    ;if no supporting units remain then skip over next loop1

 loop1
    (cond ((and (equal (second (first tmpl)) 'defend)
(null (= (get 'time gtmpbname) (get 'time 'realboard)))
(null (member (list (first 'first tmpl)) gtmpside)
      supported_corridors))
(need_help gtmpbname (first tmpl) gtmpside 3))
   :check if defense of corridor in car tmpl needs help
   (and tmp2
(setq subgoals
      (cons
(list 'defend_1_corridor
      (list (first 'first tmpl))
    (list (first tmp2))
    100))
subgoals)))
   ;send help
   (setq supported_corridors
  (cons (list (first (first tmpl)) gtmpside)
supported_corridors))
   :record fact that help is sent
   (setq tmp4 (remove (car tmpl) tmp4) tmp2 (cdr tmp2))
  ;remove corridor from list of corriodrs
  ((and 'equal (second (first tmpl)) 'attack)
(null (= (get 'time gtmpbname) (get 'time 'realboard)))
(null (member (list 'first 'first tmpl)) gtmpside)
      supported_corridors))
(need_help gtmpbname 'first tmpl) gtmpside 3))
   :check if attack in car tmpl should be supported
   (and tmp2
(setq subgoals
```

```
      (cons
 (list 'attack_1_corridor
       (list (first (first tmp1))
     (list (first tmp2))
     100))
 subgoals)))
    ;send support to attack
    (setq supported_corridors
  (cons (list (first (first tmp1)) gtmpside)
supported_corridors))
    ;record fact that corridor is being supported
    (setq tmp4 (remove (car tmp1) tmp4) tmp2 (cdr tmp2)))
   ;remove corridor from list of corridors to be checked
   )
     (and (setq tmp1 (cdr tmp1)) (go loop1))
     ;get next corridor
     ;assign a unit to defense or attack
     ;never more than one unit one for a corridor

  loop2
     (and tmp2
  (setq subgoals
(cons
  (list 'move_to_support_pos
(list (first gtmpargs)
      (first tmp2)))
  subgoals)))
     ;all remaining units should move to support corridors
     (and (setq tmp2 (cdr tmp2)) (go loop2))
     ;send other units to support waiting area
     (cond ((> (length subgoals) 1) (return (list 'list 'gnosis subgoals))))
   ((= (length subgoals) 1) (return subgoals))
   (t (return nil)))
     )
       )
     (dont_continue_if (>= (get 'time gtmpbname) (third gtmpargs))
       )
     )
  )
; this is old version of support corridors
; support corridors
; specific support corridor goal
; at present makes use of a global variable called supported_corridor that is
; set to nil on newturn  sorry bad form
'(makegoal
  '(support_corridors (list_corridors_atkdef units wait_until_time)
     (type specific)
     (subgoal
       (prog (tmp1 tmp2 tmp3 tmp4)
     (setq tmp1 (first gtmpargs) tmp3 (second gtmpargs) tmp2 tmp4 tmp1)
     ;tmp1 and tmp4 list of corridors
     ;tmp2 and tmp3 list of units
  loop0
     (and (null (member (unit_status gtmpbname (first tmp2) 'location)
(support_area tmp1 gtmpside)))
  (equal_intersect (unit_status gtmpbname (first tmp2) 'retreat_direction)
```

```
  (support_area tmp1 gtmpside))
 (setq tmp2 (remove (car tmp3) tmp2)))
    (and (setq tmp3 (cdr tmp3)) (go loop0))
    ;remove any units not waiting in support area but previously
    ;have been in support area
    ;tmp2 is list of remaining units
    (and (null tmp2) (return 'no_longer_relevant))
    ;if no supporting units remain then goal is irrelevant

 loop1
    (cond ((and (equal (second (first tmp1)) 'defend)
(null (= (get 'time gtmpbname) (get 'time 'realboard)))
(null (member (list (first (first tmp1)) gtmpside)
      supported_corridors))
(need_help gtmpbname (first tmp1) gtmpside 3))
   ;check if defense of corridor in car tmp1 needs help
   (and tmp2
(setq tmp3
      (cons
(list 'defend_1_corridor
      (list (first (first tmp1))
    (list (first tmp2))
    100))
tmp3)))
   ;send help
   (setq supported_corridors
 (cons (list (first (first tmp1)) gtmpside)
supported_corridors))
   ;record fact that help is sent
   (setq tmp4 (remove (car tmp1) tmp4) tmp2 (cdr tmp2)))
  ;remove corridor from list of corriodrs
  ((and 'equal (second (first tmp1)) 'attack)
(null (= (get 'time gtmpbname) (get 'time 'realboard)))
(null (member (list (first (first tmp1)) gtmpside)
      supported_corridors))
(need_help gtmpbname (first tmp1) gtmpside 3))
   ;check if attack in car tmp1 should be supported
   (and tmp2
(setq tmp3
      (cons
(list 'attack_1_corridor
      (list (first (first tmp1))
    (list (first tmp2))
    100))
tmp3)))
   ;send support to attack
   (setq supported_corridors
 (cons (list (first (first tmp1)) gtmpside)
supported_corridors))
   ;record fact that corridor is being supported
   (setq tmp4 (remove (car tmp1) tmp4) tmp2 (cdr tmp2)))
  ;remove corridor from list of corridors to be checked
  )
    (and (setq tmp1 (cdr tmp1)) (go loop1))
    ;get next corridor
    ;assign a unit to defense or attack
```

```
    ;never more than one unit one for a corridor

  loop2
     (and tmp2
  (setq tmp3
(cons
  (list 'move_to_support_pos
(list (first gtmpargs)
      (first tmp2)))
  tmp3)))
     ;all remaining units should move to support corridors
     (and (setq tmp2 (cdr tmp2)) (go loop2))
     ;send other units to support waiting area
     (cond ((> (length tmp3) 1) (return (list (list 'andsim tmp3))))
   ((= (length tmp3) 1) (return tmp3))
   (t (return nil)))
      )
        )
     (dont_continue_if (>= (get 'time gtmpbname) (third gtmpargs))
       )
     )
  )

; move_to_support
; sends unit to position where it can reinforce any of the
; corridors it is supposed to support
(makegoal
  '(move_to_support_pos (list_of_corridors_&_goals unit)

      (type specific)
      (subgoal
(prog (tmp1 tmp2)
      (setq tmp1 (support_area (first gtmpargs) gtmpside))
      (cond ((member (unit_status gtmpbname (second gtmpargs) 'location)
     tmp1)
     (return (list (list 'send (list (list (second gtmpargs) '(no order))))))
    (t
     (setq tmp2 (get_path gtmpbname (second gtmpargs)
  (car tmp1) 6 t))
     (return
       (list
 (list 'send
       (list
 (list (second gtmpargs)
       (cons 'move tmp2)))))))
    )
   )
)
      )
  )
; send     specific
; uses   unction send_order to send orders for each unit
(makegoal
   (send (orders)

    (type specific)
```

```
    (subgoal
      (list (list 'send gtmpargs)))
    (action
      (prog (tmp1)
    (setq tmp1 gtmpargs)
 loop1
    (send_order (first (car tmp1)) (cadr (car tmp1)))
    (and (setq tmp1 (cdr tmp1)) (go loop1))
    (return nil)))
    ))
; SOME DOMAIN UTILITIES
; TEMPORARILY HERE FOR DEBUGING PURPOSES
(defun all_combinations (list_of_lists)
  (prog (tmp1 tmp2 tmp3 tmp4)
(setq tmp1 list_of_lists tmp2 (first tmp1))
     loop1
(setq tmp3 (cons (list (first tmp2)) tmp3))
(and (setq tmp2 (cdr tmp2)) (go loop1))
;this loop sets up initial list
(setq tmp1 (cdr tmp1) tmp2 (car tmp1))
(and (null tmp1) (return tmp3))
; continue to nest loop only if more than one list_of_lists
     loop2
(setq tmp4 (cons (cons (car tmp2) (car tmp3)) tmp4))
(and (setq tmp2 (cdr tmp2)) (go loop2))
(setq tmp2 (car tmp1))
(and (setq tmp3 (cdr tmp3)) (go loop2))
(setq tmp1 (cdr tmp1) tmp2 (car tmp1) tmp3 tmp4 tmp4 nil)
(and tmp1 (go loop2))
; this is main loop will set tmp3 to list of lists that reflects
; all possible combinations of the initial lists in list_of_lists
(return tmp3)
)
  )
(defun get_defend_points (bname corridor side)
  (prog (tmp1 tmp2 tmp3 tmp4)
(setq tmp1 (get 'defend_points corridor))
;gets list of friend and enemy defend points
(cond ((equal side 'friend)
       (setq tmp1 (cadr (first tmp1))))
      ((equal side 'enemy)
       (setq tmp1 (cadr (second tmp1)))))
;loop1 goes through each set of defend points and
;looks for enemy
;last defend points before enemy is set up as the
;defend points return
(setq tmp2 (car tmp1))
     loop1
(and (setq tmp3 (boardval (eval bname) (car tmp2)))
     (equal (unit_status bname tmp3 'side)
    (opposite_side side))
     (cond ((> (length tmp4) 1)
    (return (reorder_by_strength bname tmp4 side)))
   (t (return tmp4))))
     ;if multiple defend points then order them weakest first
(and (setq tmp2 (cdr tmp2)) (go loop1))
```

```
;if no enemy units in defend_line tmp2 get next defend line
(setq tmp4 (car tmpl) tmpl (cdr tmpl) tmp2 (car tmpl))
;tmp4 is set to last defend_line
(and tmpl (go loopl))
;go check next defend line
(and (> (length tmp4) 1)
     (setq tmp4 (reorder_by_strength bname tmp4 side)))
(return tmp4)
;if no enemy found then return farthers defend point
)
  )
; reorder_by_strength
; function will return a set of defend points order in terms
; of their need for defense
(defun reorder_by_strength (bname defend_points side)
  (prog (tmpl tmp2 tmp3 values)
(setq tmpl defend_points values '(10 20 30 40 50 60 70 80 90 100 1000))
     loopl
(and (null (boardval (eval bname) (first tmpl)))
     (setq tmp2 (cons (first tmpl) tmp2)
   defend_points (remove (first tmpl) defend_points)))
(and (setq tmpl (cdr tmpl)) (go loopl))
     loop2
(setq tmpl defend_points)
     loop3
(and tmpl
     (setq tmp3 (boardval (eval bname) (first tmpl)))
     (equal (unit_status  bname tmp3 'side)
    side)
     (< (* (unit_status bname tmp3 'proficiency)
   (unit_status bname tmp3 'defend_strength))
(first values))
     (setq tmp2 (cons (first tmpl) tmp2)
   defend_points (remove (first tmpl) defend_points)))
(and (setq tmpl (cdr tmpl)) (go loop3))
(and (setq values (cdr values)) (go loop2))
(return (reverse tmp2))
)
  )
;which_corridor?
; function used only by support_corridors goal definition
; it determines which corridor a unit is already supporting
(defun which_corridor? (bname cor&goals unitname)
  (prog (tmpl tmp2 maxdist tdist)
(setq tmpl cor&goals)
     loopl
(cond ((member (unit_status bname unitname 'location)
       (get 'general_area (first (first tmpl))))
       (return (first tmpl)))
      )
(and (setq tmpl (cdr tmpl)) (go loopl))
;determine if already in general area of one of corridors
(setq tmpl cor&goals maxdist 20)
     loop2
(setq tdist (< tance (unit_status bname unitname 'location)
      (get 'center_point (first (first tmpl)))))
```

```
(and (< tdist maxdist)
     (setq tmp2 (first tmp1) maxdist tdist))
(and (setq tmp1 (cdr tmp1)) (go loop2))
;if not in general area then unit should already be going
;to the closest corridor
;tmp2 is the closets cor&goal
(return tmp2)
)
  )
;routine to find paths in corridor
(defun paths_in_corridor (bname corridor side)
  (prog (tmp1 tmp2 tmp3)
(setq tmp1 (get 'paths corridor))
;paths is a property of a corridor
'setq tmp2 (get_defend_points bname corridor (opposite_side side)))
:defend_points for other side are ordered by weakest point first
     loop1
(and (member (first tmp2) (first tmp1))
     (setq tmp3 (cons (first tmp1) tmp3)))
(and (setq tmp1 (cdr tmp1)) (go loop1))
(setq tmp1 (union tmp3 (get 'paths corridor)))
;loop1 reorders list of paths so that all paths going through
;weakest enemy defend point are first in the list
(and    ual side 'enemy)
     (prog (tmp4)
loop11
   (setq tmp4 (cons (reverse (car tmp1)) tmp4))
   (and (setq tmp1 (cdr tmp1)) (go loop11))
   (setq tmp1 tmp4)))
;reverse direction of all paths for side enemy
(return (reverse tmp1))
))
;routine to select a unit from a set of unit to go down path
(defun best_unit_for (bname path units)
  (prog (tmp1 tmp2)
(setq tmp1 units)
     loop1
(and (member (unit_status bname (car tmp1) 'location) path)
     (setq tmp2 (cons (car tmp1) tmp2)))
;if on path then its automatically a possible best unit
(and (setq tmp1 (cdr tmp1)) (go loop1))
;collect list of units already on path
(and tmp2 (= (length tmp2) 1) (return (car tmp2)))
(and tmp2 (return (closest_unit bname (car (reverse path)) tmp2)))
;if one or more units on path pick unit farthest along
(return (closest_unit bname (first path) units))
;if not units on path pick unit closest to start of path
))
;select the most forward defending unit
(defun first_defend_unit (bname corridor defend_side)
  (prog (tmp1 tmp2 tmp3)
(setq tmp1 (get 'defend_points corridor))
(cond ((equal defend_side 'friend)
       (setq tmp1 (reverse (cadr (first tmp1)))))
      ((equal defend_side 'enemy)
       (setq tmp1 (reverse (cadr (second tmp1))))))
```

```
     loop1
(setq tmp2 (car tmp1))
     loop2
(and (member (boardval (eval bname) (car tmp2))
     (active_units defend_side))
     (setq tmp3 (cons (boardval (eval bname) (car tmp2)) tmp3)))
(and (setq tmp2 (cdr tmp2)) (go loop2))
;tmp3 is now all units on forward most defend point
(and tmp3
     (prog (tmp4 tmp5)
   (setq tmp4 500)
loop11
   (and (< (* (unit_status bname (car tmp3) 'defend_strength)
      (unit_status bname (car tmp3) 'proficiency))
   tmp4)
(setq tmp4 (* (unit_status bname (car tmp3) 'defend_strength)
      (unit_status bname (car tmp3) 'proficiency))
    tmp5 (car tmp3)))
   (and (setq tmp3 (cdr tmp3)) (go loop11))
   (setq tmp3 tmp5)
   ;select weakest of forward most units
   ))
(and tmp3 (return tmp3))
;if a unit is found return it
(and (setq tmp1 (cdr tmp1)) (go loop1))
;if no unit found go down to next defend points
))
; support_area
; specifies locations that could be used to support multiple corridors
(defun support_area (cors&goals side)
  (prog (tmp1 tmp2 tmp3)
(setq tmp2 cors&goals)
     loop1
(setq tmp1 (cons (first (first tmp2)) tmp1))
(and (setq tmp2 (cdr tmp2)) (go loop1))
;get list of corridors
(setq tmp2 (get 'support_areas side))
     loop2
(setq tmp3 (first (first tmp2)))
(and (equal (union tmp1 tmp3) tmp1)
     (return (second (first tmp2))))
(and (setq tmp2 (cdr tmp2)) (go loop2))
(return (print 'error_in_support_area))
)
  )
; need help
; determines if a defense is in trouble or attack succeeding
(defun need_help (bname cor&goal side trouble_ratio)
  (prog (tmp1 tmp2 dfnd_strngth atck_strngth)
;if the other side is not attacking then no support is needed
(and (equal (second cor&goal) 'attack)
     (setq side (opposite_side side)))
;checking an attack is same as checking defense for other side
;that is if attacking is succeeding send units to exploit it
(setq tmp1 (length (get_defend_points bname (car cor&goal) side)))
(setq trouble_ratio (- trouble_ratio (* .3 (sub1 tmp1))))
```

```
;decrease trouble_ratio threshold for corridors wider than one unit
(setq tmp1 (units_in_area bname (get 'general_area (first cor&goal)) side)
      tmp2 (units_in_area bname (get 'general_area (first cor&goal))
  (opposite_side side))
      dfnd_strngth 0 atck_strngth 0)
;tmp1 and tmp2 are defend and attacking units in general area of corridor
     loop1
(and tmp1 (setq dfnd_strngth
(+ (* (unit_status bname (car tmp1) 'defend_strength)
      (unit_status bname (car tmp1) 'proficiency))
   dfnd_strngth)))
(and (setq tmp1 (cdr tmp1)) (go loop1))
;dfnd_strngth is total strength of defending units
     loop2
(and tmp2 (setq atck_strngth
(+ (* (unit_status bname (car tmp2) 'attack_strength)
      (unit_status bname (car tmp2) 'proficiency))
   atck_strngth)))
(and (setq tmp2 (cdr tmp2)) (go loop2))
;atck_strngth is total stength of attacking units
(cond ((< dfnd_strngth .5) (return t))
      ((< (quotient atck_strngth dfnd_strngth) trouble_ratio)
       (return nil))
      ((and (equal (second cor&goal) 'defend)
    (null (are_attacking bname (first cor&goal) (opposite_side side)))
    ((<= (quotient atck_strngth dfnd_strngth) 5))
       (return nil))
      ;if not yet attacked then return nil unless about to be overwhelmed
      (t (return t)))
;if attack to defend ratio is not greater than acceptable trouble_ratio
;then no support should be provided
)
  )
; units_in_area
; returns list of all units in area of specified side
(defun units_in_area (bname area side)
  (prog (tmp1 tmp2)
(setq tmp1 (active_units side))
     loop1
(and (member (unit_status bname (car tmp1) 'location)
     area)
     (setq tmp2 (cons (car tmp1) tmp2)))
(and (setq tmp1 (cdr tmp1)) (go loop1))
(return tmp2)
)
  )
; are_attacking
; determines if enemy units in corridor are attacking
(defun are_attacking (bname corridor side)
  (prog (tmp1)
(setq tmp1 (units_in_area bname (get 'general_area corridor) side))
     loop1
(and (null tmp1) (return nil))
(and (member 'list (car tmp1) 'attack) unit_actions) (return t))
(setq tmp1 (cdr tmp1))
(go loop1)
```

```
)
  )
;determines if two units can fight
(defun can_attack_unit (bname attacker defender)
  (prog (tmpl)
(setq tmpl (member (unit_status bname defender 'location)
   (zone_of_control bname attacker)))
(and tmpl (return t))
(return nil)))
(defun active_units (side)
  (cond ((equal side 'friend) friendly_units)
((equal side 'enemy) enemy_units)))
(defun closest_unit (bname location list_of_units)
  (prog (tmpl tmp2 dist tmpdist)
(setq dist 8 tmpl list_of_units)
     loopl
(and (< (setq tmpdist (distance (unit_status bname (car tmpl) 'location) location))
dist)
     (setq dist tmpdist tmp2 (car tmpl)))
(and (setq tmpl (cdr tmpl)) (go loopl))
(return tmp2)))
; equal_intersection
; returns the intersection of two lists using equal rather than eq
(defun equal_intersect (list1 list2)
  (prog (tmpl)
(and (or (null list1) (null list2)) (return nil))
     loopl
(and (member (car list1) list2)
     (setq tmpl (cons (car list1) tmpl)))
(and (setq list1 (cdr list1)) (go loopl))
(return tmpl)))
; UNUSED GOALS
; makegoals that are not presently used but embedded concept may eventually be used
'(makegoal
  '(prevent
     (type generic)
     (countergoal (list parg1))
     (subgoal nil)))
'(makegoal
  '(execute_orders
     (type generic)
     (action (prog (tmpl)
   (setq tmpl gtmpargs)
loopl
   (send_order (second (car tmpl)) (third (car tmpl)))
   (and (setq tmpl (cdr tmpl)) (go loopl))
   (execute_all_orders gtmpboard)))
     )
  )
```

PROGRAM 5

CONTINGENCY GOAL TREE

```
LMFL(:BYTE-SIZE 8 :LENGTH-IN-BLOCKS 14 :LENGTH-IN-BYTES 14073 :AUTHOR "WARGAME" :CF
WARGAME" :NAME "AFINALARES" :TYPE "L" :VERSION 1)



(defvar cgt nil)
; the cgt is the contingency goal tree that the planner works with
; as its core representation of a plan
(defvar friendcgts nil)
(defvar enemycgts nil)
; friendcgts and enemycgts is used to save successful sequences for
; friendly and enemy
; plan
; core planner routine
; fgoal is the top level friendly goal
; egoal is the top level enemy goal
; debug mode = t will activate various stopping points
(defun plan (fgoal egoal debug_mode)
  (prog (rslt)
(initialize_board 'hypboard 'realboard)
;sets hypboard to realboard position
(setq supported_corridors nil unit_actions nil)
;these global variables are used by rule base -- bad form
(display hypboard t)
;displays the intial position of search
(setq cgt (list (list fgoal) (list egoal)
(list 1)
(list (list fgoal)) (list (list egoal))))
;the cgt is structured as a list with the following sublists
;   a list of all friendly goals most recent first
;   a list of all enemy goals most recent first
;   a list of the depth in the contingency goal tree of
;     each friend and enemy goal
;   a list of friendly goals that have been or are being examined
;   a list of enemy goals that have been or are being examined
     ; loop1 is for adding nodes to the cgt
     loop1
(cond ((check_for_failure (first cgt) (third cgt) 'friend)
       (setq rslt '(failed friend)))
      ((check_for_failure (second cgt) (third cgt) 'enemy)
       (setq rslt '(failed enemy)))
      (t (setq rslt (add_gpair))))
;check if any goal on any side has failed
;if not add a new goalpair to cgt using function add_gpair
;note that if new goalpair was added rslt will equal new_node
(cond ((and (equal (car rslt) 'new_node)
   (can_act (first (first cgt)))
   (can_act (first (second cgt))))
       (update_board 'hypboard (first (first cgt)) (first (second cgt)) debug_mode
;if new goal pair was added then check if game can be updated and if so do it
;note that (first (first cgt)) is the most recent friendly goal
;     and that (first (second cgt)) is the most recent enemy goal
```

```
(and (equal (car rslt) 'new_node) (go loop1))
;if new goal pair was added to cgt then process that new goal pair
;    to get another new goal pair
      ; loop2 is used for backtracking and finding alternative goals
      loop2
(and (caar cgt) (play_out (remove_1_gpair cgt) nil nil))
;this returns to position before last goal pair
(and (modify_last_goal rslt)
      (or
        (and (can_act (first (first cgt))) (can_act (first (second cgt)))
     (update_board 'hypboard (first (first cgt)) (first (second cgt))
   debug_mode t))
        t)
      (go loop1))
;modify_last_goal tries to replace most recent goal for the side that failed
;if an alternative goal is found then the board is updated as necessary and
;                                    return to the add node loop
(setq cgt (remove_1_gpair cgt))
(and (car cgt) (go loop2))
; if an alternative goal is not found then remove last goal pair from cgt
; and try to find alternative for the new last goal on cgt
(return rslt)
; whichever side finally failed to correct the last failure has
; by definition failed
; rslt will either be (failed friend) or (failed enemy)
)
  )
; add_gpair
; this routine trys to add a new goal pair to the present cgt
(defun add_gpair ()
  (prog (fgoals egoals levels frejs erejs newfgoal newegoal level acted_flag)
(setq fgoals (first cgt) egoals (second cgt) levels (third cgt)
      frejs (fourth cgt) erejs (fifth cgt))
;the cgt is decomposed into its component parts
      ;loop1 moves up the last branch of the cgt to find a goal pair that
      ;will generate a pair of subgoals
      loop1
 ;the cond below tries to get a new friendly goal and a new enemy goal
(cond ((and (null acted_flag) (can_act (car fgoals)) (can_act (car egoals)))
       (setq newfgoal nil newegoal nil acted_flag t))
      ;goal pairs that resulted in a board update may not have subgoals
      ;acted_flag simply flags if this cgt has a board update in it
      ((and acted_flag (or (can_act (car fgoals)) (can_act (car egoals))))
       (setq newfgoal nil newegoal nil))
      ;once again goal pairs that resulted in a board update may not have subgoals

      ((or
 (check_dont_continue fgoals levels 'friend)
 (check_dont_continue egoals levels 'enemy))
       (setq newfgoal nil newegoal nil))
      ;if either the friendly or the enemy goal is no longer active
      ;   then the goal pair may not generate a sub goal pair

      (t (setq newfgoal (get_sub 'hypboard (car fgoals) nil  friend)
       newegoal (get_sub 'hypboard (car egoals) nil 'enemy)))
      ;otherwise try to get new goals for both sides
```

```
      )
;the cond below processes the results of the above cond
(cond
  ((and newfgoal newegoal)
   (setq cgt (list (cons newfgoal (first cgt))
   (cons newegoal (second cgt))
   (cons (add1 (car levels)) (third cgt))
   (cons (list newfgoal) (fourth cgt))
   (cons (list newegoal) (fifth cgt))))
   (return (list 'new_node (list newfgoal newegoal))))
   :if new goals for both sides then add these to the cgt and return

  ((and (null newfgoal) (null newegoal))
   (setq level (car levels))
   (prog ()
      loop11
 (setq fgoals (cdr fgoals) egoals (cdr egoals) levels (cdr levels)
       frejs (cdr frejs) erejs (cdr erejs))
 (and (<= level (car levels)) (go loop11))
 (return t))
   (and levels (go loop1))
   (return (print "bad goals returned to top level sans resolution")))
   ;if no new goals then move up one level on the cgt and try to generate a
   ; new subgoal pair from there

  ((and newfgoal (null newegoal))
   (return '(failed enemy)))
  ((and (null newfgoal) newegoal)
   (return '(failed friend)))
  ;if only one side can continue but not the other then the side that
  ;cant continue must have failed
   )
)
  )
: modify last goal
; for the side that failed this routine will try to replace
; the last goal in the cgt
(defun modify_last_goal (because_of)
  (prog (fgoals egoals levels frejs erejs newgoal level cgt_copy)
(setq cgt_copy cgt level (car (third cgt_copy)))
;make a copy of the cgt and set level to depth of last
;goal pair in the cgt
     loop1
;the first part of this loop (next four lines) finds the
        ;parent goal pair of the goal pair that is to be replaced
(setq cgt_copy (remove_1_gpair cgt_copy))
;move back to the previous goal
(and (null (third cgt_copy)) (return nil))
;if cgt is empty then return nil
(and (<= level (car (third cgt_copy)))
     (go loop1))
:if after moving back level did not increase then
;the parent of the goal pair has not yet been found
:at this point the parent goal pair is the last node
:in cgt_copy
(setq fgoals (first cgt) egoals (second cgt) levels (third cgt)
```

```
      frejs (fourth cgt) erejs (fifth cgt))
;break up present cgt into its component parts
;the cond below tries to find an alternative goal for the
;goal that failed
(cond
  ((equal because_of '(failed friend))
   (setq newgoal
 (get_sub 'hypboard (car (first cgt_copy))
  (car frejs) 'friend))
   (and (null newgoal) (return nil))
   (setq fgoals (cons newgoal (cdr fgoals)))
   (setq frejs (cons (cons newgoal (first frejs)) (cdr frejs))))
   ;if friend failed then try replacing the friend goal
  ((equal because_of '(failed enemy))
   (setq newgoal
 (get_sub 'hypboard (car (second cgt_copy))
  (car erejs) 'enemy))
   (and (null newgoal) (return nil))
   (setq egoals (cons newgoal (cdr egoals)))
   (setq erejs (cons (cons newgoal (first erejs))
     (cdr erejs))))
   ;if enemy failed then try replacing the enemy goal
      )

(setq cgt (list fgoals egoals levels frejs erejs))
;reconstruct the cgt
(return (list 'newgoal (second because_of) newgoal))
;return the result of modify_last_goal processing
)
  )
; get_sub
; this routine actually controls the execution of subgoal processing
; bname is the boardname
; goal is the goal for which a subgoal is desired
; rejs is the list of subgoals that have already been tried
; side indicates that it is an enemy or friendly goal
(defun get_sub (bname goal rejs side)
  (prog (tmpl)
(setq tmpl (process_goal bname goal side 'subgoal))
;the above gets all subgoals of goal
     loopl
(setq tmpl (remove (car rejs) tmpl))
(and (setq rejs (cdr rejs)) (go loopl))
;remove previously tried goals from the list of subgoals
(return (car tmpl))
;return the first subgoal that has not already been tried
)
  )
; check_for_failure
; determines if the most recent goal in the cgt or any of its parents have failed
; goals is the list of all friend or enemy goals in the cgt
; levels is a list that indicates the depth in the tree of each element of goals
; side is friend or enemy
(defun check_for_failure (goals levels side)
  (prog (t_l level)
     loopl
```

```
(and (setq tmpl (process_goal 'hypboard (car goals) side 'failed_if))
     (return (list 'failed (car goals))))
;check if first goal in list of goals has failed
(setq level (car levels))
;if goal has not failed then level is depth of this goal in cgt
     loop2
(setq goals (cdr goals) levels (cdr levels))
;pop last goal from list of goals
(and (null goals) (return nil))
;if all goals have been poped then no failure was found
(and (<= level (car levels)) (go loop2))
;if last goal is not at a higher level in the cgt
;it is not a parent of the goal that was just checked
(go locpl))
  )
; check_dont_continue
; determines if the most recent goal in the cgt or any of its parents
; should not be continued
; success or failure cannot be inferred from this processing
(defun check_dont_continue (goals levels side)
  (prog (tmpl level)
     loop1
(and (setq tmpl (process_goal 'hypboard (car goals) side 'dont_continue_if))
     (return (list 'dont_continue (car goals))))
;check if first goal in list of goals should be discontinued
(setq level (car levels))
;if not then level is depth of goal in cgt
     loop2
(setq goals (cdr goals) levels (cdr levels))
;pop last goal from list of goals
(and (null goals) (return nil))
;if all goals have been poped then no goal was found to discontinue
(and (<= level (car levels)) (go loop2))
;if last goal is not at a higher level in the cgt
;it is not a parent of the goal that was just checked
(go loop1))
  )
; can_act
; determines if the goal is sufficiently specific so as to be able to update the s
; this function is somewhat domain dependent but can be generalized
(defun can_act (goal)
  (prog (tmpl tmp2)
(setq tmpl goal)
     loop1
(and (null (member (car tmpl) '(andsim orsim send))) (return nil))
;if any of the componenet goals embedded in goal
;is not an andsim orsim  or send then it is not executable
(and (member (car tmpl) '(andsim orsim))
     (setq tmpl (cadr tmpl))
     (prog ()
loop11
   (setq tmp2 (cons (car tmpl) tmp2))
   (and (setq tmpl (cdr tmpl)) (go loop11)))
     .
;the above and gets all the component goals of andsim  or  orsim
;and puts them in tmp2
```

```
(setq tmp1 (car tmp2) tmp2 (cdr tmp2))
;tmp2 is list of all remaining goals and tmp1 is present goal to check
(and tmp1 (go loop1))
;as long as there is a goal to check keep checking
(return t)
;if all discovered componenet goals are executable then entire goal is
;executable
)
  )
(defun update_board (bname fgoal egoal debug_mode display_board)
  (get&send_orders bname fgoal 'friend)
  ;send orders defined if friendly goal
  (get&send_orders bname egoal 'enemy)
  ;send orders defined in enemy goal
  (execute_orders bname display_board debug_mode)
  ;execute_orders is a routine defined in domain
  )
; get&send_orders
; assuming goal can be acted upon will send all orders embedded in goal
(defun get&send_orders (bname goal side)
  (prog (tmp1 tmp2)
(setq tmp1 (list goal))
;tmp1 starts as list of goals to process
     loop1
(and (null tmp1) (return t))
;if no more goals then done
(cond ((equal (caar tmp1) 'send)
       (process_goal bname (car tmp1) side 'action)
       (setq tmp1 (cdr tmp1))
       (go loop1)))
;if goal is to send an order then send order using action slot of
;goal definition
(and (member (caar tmp1) '(andsim orsim))
     (prog ()
   (setq tmp2 (cadar tmp1) tmp1 (cdr tmp1))
loop11
   (setq tmp1 (cons (car tmp2) tmp1))
   (and (setq tmp2 (cdr tmp2)) (go loop11))
   (return t))
     (go loop1)
     )
;if first goal is concatenation of goals then get
;component goals  put them in tmp1  and process these goals
))
; remove_1_gpair
; returns all but the last goal pair of a cgt
(defun remove_1_gpair (cgt_like)
  (prog (cgt_to_return)
(or cgt_like (return nil))
     ;this loop gets the cdr of each element of cgt_like and puts
     ;it into cgt_to_return
     loop1
(cond ((atom (car cgt_like)) (return 'error_in_remove_1_pair))
      (t (setq cgt_to_return (cons (cdr (car cgt_like)) cgt_to_return))))
(and (setq cgt_like (cdr cgt_like)) (go loop1))
;a cgt has five elements so above loop goes five times
```

```
(return (reverse cgt_to_return))
;return cgt sans last goal pair
)
  )
; play_out
; steps through all the nodes of a cgt and updates the board accordingly
(defun play_out (cgt_copy debug_mode display_board)
  (prog (fgoals egoals)
(setq fgoals (reverse (first cgt_copy))
      egoals (reverse (second cgt_copy)))
;get friendly and enemy goals in the cgt
(initialize_board 'hypboard 'realboard)
;reset board to initial position
(setq supported_corridors nil unit_actions nil)
;reset global lists
     loopl
(and (car cgt_copy)
     (can_act (first fgoals)) (can_act (first egoals))
     ;check if goal pair is executable
     (or (get&send_orders 'hypboard (first fgoals) 'friend) t)
     (or (get&send_orders 'hypboard (first egoals) 'enemy) t)
     ;send orders
     (execute_orders 'hypboard display_board debug_mode)
            ;execute orders
     )
(and (setq fgoals (cdr fgoals) egoals (cdr egoals)) (go loopl))
;pop last goal and go back to loopl
)
  )
```

PROGRAM 6

WARGAME

```
LMFL(:BYTE-SIZE 8 :LENGTH-IN-BLOCKS 46 :LENGTH-IN-BYTES 46619 :AUTHOR "WARGAME" :CR
WARGAME" :NAME "AFINALGAME" :TYPE "L" :VERSION 1)



; this is the file that defines the wargame
; this file plus a terrain - unit definition file would be sufficient
; for manually playing this game
; Note that since this is only an example game and not a key element
; of the ARES project the documentation here is not as detailed as
; commenting in the planner itself
;  GLOBAL VARIABLES
(defvar globalarg nil)
; whenever a global arguement is needed always use this
; may be useful for instance if want to pass a single parameter
; into a mapcar
(defvar initial nil)
(defvar realboard nil)
(defvar hypboard nil)
; the game allows use of any of three boards referred to as
; initial realboard and hypboard
; the game can be played on any of these three boards
; this is done for the sake of the planner and not the game
(defvar friendly_units nil)
(defvar enemy_units nil)
(defvar all_units nil)
;list of active friendly and enemy units
; must be explicitly saved for hypothetical search
(defvar previous_get_paths nil)
;list of all get_path results this saves much time
; this varialbe is used only by get_path
(defvar enemy_orders nil)
(defvar friend_orders nil)
;global list of active orders
(defvar list_of_unit_properties nil)
(setq list_of_unit_properties '(side id type size attack_strength defend strength ;
     movement_allowance location is_active retreat_direction
     location_status previous_locations))
;global list of all properties that may be attached to a unit
(defvar supported_corridors nil)
; a variable used only by goal support_corridors to identify if a corridor has
; already been supported  this is a bad cluge to be repaired later
; newturn sets this variable to nil
(defvar unit_actions nil)
; a variable used only by goal support_corridors to identify
; what is currently happening in a corridor
; this is also a bad cluge to be repaired later
;  BOARD ACCESS ROUTINES
; the following routines are for the value of any board position
; or for defining the relationship between any two positions
; note that a board is defined as a matrix-like list that
; where each location is either a possible unit location or a terrain
; location
```

```
; see example of board in file finalterrain.l
; boardval
; returns value of unit location loc from board
(defun boardval (board loc)
  (prog (x y x2 y2 bd)
(setq x (car loc) y (cadr loc))
(setq x2 1 y2 1 bd (cdr board))
loop1
(and (equal y2 y) (setq bd (car bd)) (go loop2))
(setq y2 (add1 y2) bd (cddr bd))
(go loop1)
loop2
(and (equal x2 x) (return (cadr bd)))
(setq x2 (add1 x2) bd (cddr bd))
(go loop2)
))
; boardset
; returns a board that is same as input board
; but has value val at location loc
(defun boardset (board loc val)
  (prog (x y x2 y2 board1 board2 lst1 lst2)
(setq x (car loc) y (cadr loc))
(setq x2 1 y2 1 board1 board nil
      lst1 nil lst2 nil)
     loop1
(cond ((> y2 12)
       (setq board2 (cons (car board1) board2))
       (return (reverse board2)))
      ((equal y2 y)
       (setq y2 (add1 y2) board2
     (cons (car board1) board2) board1 (cdr board1)
     lst1 (car board1) board1 (cdr board1))
       (go loop2))
      (t
       (setq y2 (add1 y2)
   board2 (cons (car board1) board2) board1 (cdr board1)
     board2 (cons (car board1) board2) board1 (cdr board1))))
(go loop1)
     loop2
(cond ((equal x2 x)
       (setq x2 (add1 x2)
     lst2 (cons (car lst1  lst2) lst1) (cdr lst1)
     lst2 (cons val lst2) lst1 (cdr lst1)))
      ((> x2 12) (setq lst2 (cons (car lst1) lst2)
  board2 (cons (reverse lst2) board2))
(go loop1))
      (t (setq x2 (add1 x2)
       lst2 (cons (car lst1) lst2) lst1 (cdr lst1)
       lst2 (cons (car lst1) lst2) lst1 (cdr lst1))))
(go loop2)
))
; cost_of_move
; returns movement cost for going in any particular direction
;   bname is the name of the board
;   unitname is the name of the unit
;   loc is the starting location
```

```
;   dir is the direction of the move
;   side is the side on the move
;   terrain_only is a flag which if t will ignore extra cost for moving
;                through other side zone of control
(defun cost_of_move (bname unitname loc dir side terrain_only)
  (prog (x y bd cst)
(setq x (car loc) y (cadr loc)
      x (* x 2) y (* y 2)
      x (+ x (car dir)) y (+ y (cadr dir)))
(setq bd (eval bname))
;set expanded board location  and other parameters
(cond
  ((equal (unit_status bname unitname 'type) 'helicopter)
   (setq cst 1))
  (t (setq cst (anyval bd (list x y)))))
(cond
  ((= (+ (abs (car dir)) (abs (cadr dir))) 2)
   (setq cst (* 1.414 cst))))
;if diagonal move cost of move is 1.414 times terrain value
(cond
  ((and (null terrain_only)
(within_zone_of_control bname (opposite_side side)  loc))
   (return (add1 cst)))
  (t (return cst)))
))
; anyval
; uses all locations unit or terrain and gets corresponding value
; this will be referred to as the extended_board
;   board is an actual board
;   loc is the location on the board
(defun anyval (board loc)
  (prog (x y x2 y2 bd)
(setq x (car loc) y (cadr loc))
(setq x2 1 y2 1 bd board)
loop1
(and (equal y2 y) (setq bd (car bd)) (go loop2))
(setq y2 (add1 y2) bd (cdr bd))
(go loop1)
loop2
(and (equal x2 x) (return (car bd)))
(setq x2 (add1 x2) bd (cdr bd))
(go loop2)
))
; terrain_between
; for any two unit locations returns the terrain_value between those two locations
(defun terrain_between (bname from to)
  (prog (dir)
(setq dir (-list to from))
(setq from (list (* (car from) 2) (* (cadr from) 2)))
(return (anyval (eval bname) (+list from dir)))
)
  )
; display
; displays the board in board
;   clrscrn_flg will clear the whole screen first if t
(defun display (board clrscrn_flg)
```

```
  (prog (tmpflg)
(cond (clrscrn_flg (send *standard-output* :clear-screen))
      (t (cursorpos 0 0) (terpri) (cursorpos 0 0)))
(and (atom board) (return 'bad_arguement_to_display))
(setq tmpflg t)
(loop for j from 1 to 25 do
      (princ '   ')
      (loop for i from 1 to 27 do
   (displayl board (list i j))
   (cond (tmpflg (princ '  '))
  (t (princ ' '))))
      (and (null tmpflg) (princ (addl (quotient (subl j) 2))))
      (terpri) (terpri)
      (setq tmpflg (null tmpflg)))
(princ '      ')
(loop for i from 1 to 13 do
      (princ i) (cond ((< i 10) (princ '     '))
      (t (princ '    '))))
(or clrscrn_flg (cursorpos 50))))
; displayl
: displays a single board position
(defun displayl (board place)
  (prog (tmpl)
(setq tmpl (anyval board place))
(cond ((equal tmpl 5) (princ '*'))
      ((equal tmpl 2) (princ 'x'))
      ((equal tmpl 1) (princ ' '))
      ((null tmpl) (princ '---'))
      (t (princ tmpl)))))
;  UNIT DEFINITION AND ACCESS ROUTINES
: define_units
; sets up a new unit as atom 'name' with large property list
(defun define_unit (name side id type size attack_strength defend_strength
    proficiency movement_allowance location is_active)
  (putprop 'side (list side side side) name)
  (putprop 'id (list id id id) name)
  (putprop 'type (list type type type) name)
  (putprop 'size (list size size size) name)
  (putprop 'attack_strength
   (list attack_strength attack_strength attack_strength) name)
  (putprop 'defend_strength
   (list defend_strength defend_strength defend_strength) name)
  (putprop 'proficiency
   (list proficiency proficiency proficiency) name)
  (putprop 'movement_allowance
   (list movement_allowance movement_allowance movement_allowance) name)
  (putprop 'location (list location location location) name)
  (putprop 'is_active (list is_active is_active is_active) name)
  (putprop 'retreat_direction
   (list (list location) (list location)
 (list location)) name)
  (putprop 'location_status (list 'no_conflict nil nil) name)
  (putprop 'previous_locations (list nil nil nil) name)
  ;above creates the required property lists
  ;car is initial status  cadr is real status caddr is status on the hypboard
```

```
  (cond ((equal side 'friend)
 (setq friendly_units (cons name friendly_units)
       all_units (cons name all_units)))
((equal side 'enemy)
 (setq enemy_units (cons name enemy_units)
       all_units (cons name all_units))))
  ; above updates list of friend and enemy units in play
  )
; put_unit_on_board
; after a unit is defined, this routine will put it on the specified board
;    bname is the name of the board
;    unitname is the name of the unit
;    location is the location that it will start at
(defun put_unit_on_board (bname unitname location)
  (prog ()
(and (boardval (eval bname) location)
     (return 'unit_already_there))
(set bname (boardset (eval bname) location unitname))
(set_status bname unitname 'location_status 'no_conflict)
(set_status bname unitname
    'previous_locations (list location location))
)
  )
; unit_status
; gets the specified status value for the board specified
;   bname is the name of the board
;   unitname is the name of the unit
;   property is the unit property to retrieve
(defun unit_status (bname unitname property)
  (cond ((equal bname 'initial) (car (get property unitname)))
((equal bname 'realboard) (cadr (get property unitname)))
((equal bname 'hypboard) (caddr (get property unitname))))
  )
; display_unit_status
; displays status of all properties of unit that may be of interest to player
(defun display_unit_status (bname unitname)
  (print (list bname 'status 'of unitname))
  (terpri)
  (princ "   side  ")
  (princ (unit_status bname unitname 'side))
  (terpri)
  (princ "   id   ")
  (princ (unit_status bname unitname 'id))
  (princ "        type ")
  (princ (unit_status bname unitname 'type))
  (princ "        size ")
  (princ (unit_status bname unitname 'size))
  (terpri)
  (princ " attack_strength ")
  (princ (unit_status bname unitname 'attack_strength))
  (princ "      defend_strength ")
  (princ (unit_status bname unitname 'defend_strength))
  (terpri)
  (princ " proficiency ")
  (princ (unit_status bname unitname 'proficiency))
  (princ "       movement_allowance ")
```

```
  (princ (unit_status bname unitname 'movement_allowance))
  (terpri)
  (princ "   location ")
  (princ (unit_status bname unitname 'location))
  (princ "        is_active ")
  (princ (unit_status bname unitname 'is_active))
  (terpri)
  )
; display_status
; displays the status for all units of any unit property
(defun display_status (bname property)
  (prog (tmp1 tmp2)
(setq tmp1 friendly_units tmp2 enemy_units)
(terpri)
(princ "      enemy                          friendly")
(terpri)
(princ "     ")
(cond ((car tmp2)
       (princ (list (car tmp2)
    (unit_status bname (car tmp2) property))))
      (t (princ "             "))))
(princ "                      ")
(cond ((car tmp1)
       (princ (list (car tmp1)
    (unit_status bname (car tmp1) property)))))
(setq tmp1 (cdr tmp1) tmp2 (cdr tmp2))
(and (or tmp1 tmp2) (go loop1))
)
  )
; set_stat
; used to modify real or hypothetical status of a single unit property
;   bname is the name of the board
;   unitname is the name of the unit
;   property is the name of the property to be modified
;   status is the new status
(defun set_status (bname unitname property status)
  (cond ((equal bname 'realboard)
 (putprop property
  (list (unit_status 'initial unitname property)
status
(unit_status 'hypboard unitname property))
  unitname))
((equal bname 'hypboard)
 (putprop property
  (list (unit_status 'initial unitname property)
(unit_status 'realboard unitname property)
status)
  unitname)))
  )
; reset_status
; resets unit status and board status of of_bname to to_bname
;   of_bname is the name of the board to change
;   to_bname is the name of the board that of_bname should be
;            equivalent to
(defun reset_status (of_bname to_bname)
```

```
  (prog (tmp1 tmp2)
(setq tmp1 list_of_unit_properties tmp2 all_units)
     loop1
(set_status of_bname (car tmp2) (car tmp1)
    (unit_status to_bname (car tmp2) (car tmp1)))
(and (setq tmp1 (cdr tmp1)) (go loop1))
(setq tmp1 list_of_unit_properties)
(and (setq tmp2 (cdr tmp2)) (go loop1))
; above loop set all properties of a unit to be as in to_bname
(set of_bname (eval to_bname))
(putprop 'time (get 'time to_bname) of_bname)
; of_bname is now reset
(setq tmp2 all_units friendly_units nil enemy_units nil)
     loop2
(cond ((and (equal (unit_status of_bname (car tmp2) 'side) 'friend)
    (null (dead_unit of_bname (car tmp2))))
       (setq friendly_units (cons (car tmp2) friendly_units)))
      ((and (equal (unit_status of_bname (car tmp2) 'side)  enemy)
    (null (dead_unit of_bname (car tmp2))))
       (setq enemy_units (cons (car tmp2) enemy_units))))
(and (setq tmp2 (cdr tmp2)) (go loop2))
; above loop sets up list of friendly_units and enemy_units to be
: at status of of_bname
; note that this routine can be used with of_bname=to_bname
; to set friendly and enemy units list properly
(return (list 'status of_bname 'reset 'to to_bname))))
; clear_orders
; this routine clears the orders for all units
; that is sets present order to (no order)
(defun clear_orders ()
  (prog (tmp1)
(setq tmp1 (union friendly_units enemy_units))
(and (null tmp1) (return (print 'error_in_clear_orders)))
loop1
(send_order (car tmp1) '(no order))
(and (setq tmp1 (cdr tmp1)) (go loop1))
)
  )
; initialize_board
; copys board and clears orders
(defun initialize_board (of_bname to_bname)
  (reset_status of_bname to_bname)
  (clear_orders))
;  MOVEMENT FUNCTIONS
; these functions define procedures for moving units
; move_1_space
; moves a unit one space if feasible
; otherwise returns a diagnostic msg
;   bname is the name of the board
;   unitname is the name of the unit to move
;   to is the location to move to
(defun move_1_space (bname unitname to)
  (prog (dir mcost side from tmp1 conflict_flg)
(or (setq from (unit_status bname unitname 'location)) (return 'no_unit_there))
(setq dir (list  - (car to) (car from) (- (cadr to) (cadr from))))
(cond ((or (> (abs (car dir)) 1) (> (abs (cadr dir)) 1)) (return 'not_one_jumb)))
```

```
; sets up from location and direction of move, rejects if illegal move
(setq tmpl (list (boardval (eval bname) to)
 (and (equal (+ (abs (car dir)) (abs (cadr dir))) 2)
      (boardval (eval bname) (list (+ (car from) (car dir))
   (cadr from))))
 (and (equal (+ (abs (car dir)) (abs (cadr dir))) 2)
      (boardval (eval bname) (list (car from)
   (+ (cadr from) (cadr dir))))))
 ))
loop1

(and (car tmpl) (cond ((null (equal
       (unit_status bname (car tmpl) 'side)
       (unit_status bname unitname 'side)))
       (return 'enemy_unit_in_the_way))
      't (and (null cnflct_flg) (setq cnflct_flg t)))))
(and (setq tmpl (cdr tmpl)) (go loop1))
; if enemy unit in way then move is rejected if friendly unit then cnflct_flg is set
(setq side (unit_status bname unitname 'side))
(setq mcost (cost_of_move bname unitname from dir side nil))
(cond ((> mcost (unit_status bname unitname 'movement_allowance))
       (return 'to_costly))
      (t
       (cond ((within_zone_of_control bname (opposite_side side) from)
      (set_status bname unitname 'proficiency
(sub1 (unit_status bname unitname 'proficiency)))
      ))
       (change_location bname unitname from to)
       (set_status bname unitname 'movement_allowance
   (- (unit_status bname unitname 'movement_allowance) mcost)))))
; if mov  is too costly then it is rejected else the move is made
(set_status bname unitname 'retreat_direction
    (cons to (unit_status bname unitname 'retreat_direction)))
; retreat direction is a list of each past position
(cond ((null cnflct_flg) (return t))
      (t (return 'possible_problem)))
; if to location already has a unit of same side then a possible problem is returned
; but move is still made
)
  )
; change_location
; this is the routine that actually changes all location variables
;   bname is the name of the board
;   unitname is the name of the unit
;   from is the starting location
;   to is the ending location
; function will allow two units on top of each other but will record
;          possible conflict
(defun change_location (bname unitname from to)
  (prog (tmpl)
(setq tmpl (another_at_location bname unitname))
(set_status bname unitname 'location to)
(cond ((boardval (eval bname) to)
       (set_status bname unitname 'location_status 'possible_conflict))
      (t (set_status bname unitname 'location_status 'no_conflict)
 (set bname (boardset (eval bname) to unitname))))
```

```
(set bname (boardset (eval bname) from tmp1))
)
  )
; seperate_moves
; takes a set of move sequences for multiple units and seperates them into a series
; of independent and time sequenced single move steps
(defun seperate_moves (bname side)
  (prog (tmp1 tmp2 tmp3 tmp4 unitname moves units)
(cond ((equal side 'friend) (setq units friendly_units))
      ((equal side 'enemy) (setq units enemy_units)))
loop1
(cond ((null (equal (car (present_order (car units))) 'move)) (go loop2)))
(setq unitname (car units) moves (cdr (present_order unitname)))
(and (car moves)
     (setq tmp1 (cons (list 'move_1_space bname unitname (car moves)) tmp1)))
(and (cadr moves)
     (setq tmp2 (cons (list 'move_1_space bname unitname (cadr moves)) tmp2)))
(and (caddr moves)
     (setq tmp3 (cons (list 'move_1_space bname unitname (caddr moves)) tmp3)))
(and (cadddr moves)
     (setq tmp4 (cons (list 'move_1_space bname unitname (cadddr moves)) tmp4)))
loop2
(and (setq units (cdr units)) (go loop1))
(return (list tmp1 tmp2 tmp3 tmp4))))
;execute_move
; takes a set of move sequences for multiple units and plays them out on
; the game board
;   bname is the name of the board
;   display_board is a flag which if t will display board after moves
(defun execute_moves (bname display_board)
  (prog (tmp1 tmp2 rslt frnd_moves enemy_moves units backup_rslt)
(setq frnd_moves (seperate_moves bname 'friend)
      enemy_moves (seperate_moves bname 'enemy))
;frnd_moves and enemy_moves is now a set of independent single moves
     loop1
(setq tmp1 (car frnd_moves) tmp2 (car enemy_moves))
     loop2
(and tmp1 (setq rslt (move_1_space bname (caddr (car tmp1)) (cadddr (car tmp1)))))
(setq tmp1 (cdr tmp1))
(and tmp2 (setq rslt (move_1_space bname (caddr (car tmp2)) (cadddr (car tmp2)))))
(setq tmp2 (cdr tmp2))
(and (or tmp1 tmp2) (go loop2))
(setq frnd_moves (cdr frnd_moves) enemy_moves (cdr enemy_moves))
(and (or frnd_moves enemy_moves) (go loop1))
;above loops execute all moves
(setq units (append enemy_units friendly_units))
     loop3
(and (equal (unit_status bname (car units) 'location_status) 'possible_conflict)
     (setq tmp1 (another_at_location bname (car units)))
     (setq backup_rslt (backup bname (car units)))
     (cond ((equal (car backup_rslt) 'still have_conflict)
    (set bname
 (boardset (eval bname)
   (unit_status bname (car units) 'location
   (car units)))
    (setq units (cons (car units) (cons (cadr backup_rslt) (cdr units))))
```

```
   (t t))
     (set bname (boardset (eval bname)
  (unit_status bname tmp1 'location)
  tmp1)))
(and (setq units (cdr units)) (go loop3))
;above loop will move_back units to avoid double locations
(and display_board (print 'all_moves_executed))
(return t)
))
; another_at_location
; during moving of units more than one unit may be on a single unit location
; this routine determines if more than one unit is at the location of unitname
;   bname is the name of the board
;   unitname is the name of the unit
(defun another_at_location (bname unitname)
  (prog (tmp1 units)
(setq tmp1 (unit_status bname unitname 'location)
      units (delete unitname (union friendly_units enemy_units)))
loop1
(and (null units) (return nil))
(and (equal tmp1 (unit_status bname (car units) 'location)) (return (car units)))
(setq units (cdr units))
(go loop1)
(return nil)
)
  )
; backup
; this routines backs a unit up along the path it just took
; by backing up the problem of having multiple units at a single location
; is resolved
(defun backup (bname unitname)
  (prog (tmp1 tmp2 flg)
(setq tmp1 (cdr (unit_status bname unitname 'retreat_direction)))
(and (null tmp1) (print 'impossible_backup_problem))
     loop1
(cond ((null tmp1)
       (set_status bname tmp2 'location_status 'possible_conflict)
       (set_status bname unitname 'location_status 'no_conflict)
       (return (list 'still_have_conflict tmp2))))
;if can not backup further then return this fact and the name of the other
;unit that is at location    this unit will now have to backup
(and (null (setq tmp2 (boardval (eval bname) (car tmp1)))) (setq flg t))
(change_location bname unitname (unit_status bname unitname 'location) (car tmp1))
(set_status bname unitname 'retreat_direction
    (cdr (unit_status bname unitname 'retreat_direction)))
(setq tmp1 (cdr tmp1))
(and (null flg) (go loop1))
;this loop keeps backing up one space at a time until unit has
;moved into an empty location
(set_status bname unitname 'location_status 'no_conflict)
(return (list 'no_problem_in_backup))))
; ZONE OF CONTROL functions
; these routines define a units zone_of_control
; adjacent_squares
; returns all squares adjacent to a given location
;   location is a unit location
```

```
(defun adjacent_squares (location)
  (mapcar
    '(lambda (k)
       (list (+ (car location) (car k))
     (+ (cadr location) (cadr k))))
       '((1 0) (-1 0) (0 1) (0 -1) (1 1) (1 -1) (-1 1) (-1 -1))))
; zone_of_control
; returns zone of contorl of unitname
; the zone of control should be equal to all locations adjacent to
; the unit that can be reached by that unit in one jump
;   bname is the name of the board
;   unitname is the name of the unit
(defun zone_of_control (bname unitname)
  (setq globalarg (list bname unitname (unit_status bname unitname 'location)
(u  _status bname unitname  'side)))
      'lobalarg was defined using defvar
  ;   is used to pass an arguement into the mapcar
  ;this is obviously a bad cluge
  (cons (unit_status bname unitname 'location)
(remove nil
(mapcar
  '(lambda (k)
     (and (< (cost_of_move (car globalarg) (cadr globalarg)
   (caddr globalarg) k (cadddr globalarg) t) 4.5)
  (list (+ (car (caddr globalarg)) (car k))
(+ (cadr (caddr globalarg)) (cadr k)))))
  '((1 0) (-1 0) (0 1) (0 -1) (1 1) (1 -1) (-1 1) (-1 -1))))))
; within_zone_of_control
; determines if a location is within zone of control of side
; this is equivalent to the location being within the zone of control
; of any of the units of the specified side
;   bname is the name of the board
;   side is either friend or enemy
;   location is a unit location
(defun within_zone_of_control (bname side location)
  (prog (tmpl)
(cond ((equal side 'enemy) (setq tmpl enemy_units))
      ((equal side 'friend) (setq tmpl friendly_units))
      (t (return 'error-in-within-zone-of-control)))
loop1
(and (null tmpl) (return nil))
(cond ((member location (zone_of_control bname (car tmpl)))
       (return t))
      (t (setq tmpl (cdr tmpl))))
(go loop1)))
;  ATTACK AND DEFEND ROUTINES
; these routines execute an attack and update unit status and board
; positions appropriately
; execute_all_attacks
; executes enemy and friendly attacks    enemy first
;     bname is the name of the board
(defun execute_all_attacks (bname)
  (execute_attacks bname 'enemy)
  (execute_attacks bname 'friend))
; execute_attacks
; gets all of the attacks for one side and executes them
```

```
;    bname is the name of the board
;    side is the side executing attacks
; the actual attacks are defined by the orders for the side side
(defun execute_attacks (bname side)
  (prog (tmpl units dfndlst)
(cond ((equal side 'friend) (setq units friendly_units))
      ((equal side 'enemy) (setq units enemy_units))
      (t (return 'error_in_execute_attack)))
     loop1
(cond ((null
 (equal (car (setq tmpl
   (present_order (car units))))
'attack))
       (go loop2)))
;if it is an attack cond below will add it to dfndlst
(cond ((member (cadr tmpl) dfndlst)
       (setq dfndlst
     (add_to_dfndlst (cadr tmpl) (car units) dfndlst)))
      (t
       (setq dfndlst
     (cons (cadr tmpl)
   (cons (list (car units)) dfndlst)))))
     loop2
(and (setq units (cdr units)) (go loop1))
;above loops set up list a list of alternatir   efending unit
;and list of attackers pairs
;such as (funit1 (eunit1 eunit2) funit2 (euni     nit4)) where funit1
;is to be attacked by eunit1 and eunit2 and fur    is to attacked by
;eunit3 and eunit4
     loop3
(and dfndlst (attack bname (cadr dfndlst) (car dfndlst)))
(and (setq dfndlst (cddr dfndlst)) (go loop3))
;loop3 actual executes each attack
(return t)
))
; add_to_dfndlst
; local routine that adjusts list of attackers in dfndlst
(defun add_to_dfndlst (defender attacker dfndlst)
  (prog (tmpl)
     loop1
(cond ((equal defender (car dfndlst))
       (setq tmpl (cons defender tmpl))
       (setq tmpl (cons (cons attacker (cadr dfndlst)) tmpl)))
      (t (setq tmpl (cons (car dfndlst) tmpl))
 (setq tmpl (cons (cadr dfndlst) tmpl))))
(and (setq dfndlst (cddr dfndlst)) (go loop1))
(return (reverse tmpl))))
; attack
;this is the main routine for determing the outcome of an attack
(defun attack (bname attackers defender)
  (prog (actual_atck_strgth actual_dfnd_strgth attack_rslt tmpl
 terrain_value terrain_mult defend_pos_mult)
(setq actual_atck_strgth 0 actual_dfnd_strgth 0)
; thses two variables are used to
        ;determine ratio for battle outcomes
(setq tmpl attackers)
```

```
     loop1
(or (can_attack? bname (car tmp1) defender)
    (print (list 'illegal_attacker (car tmp1)))
    (delete (car tmp1) attackers))
(and (setq tmp1 (cdr tmp1)) (go loop1))
;set attackers to be sublist of attackers
        ;that can defend unitary defend group
(setq tmp1 attackers)
     loop2
(setq terrain_value
      (terrain_between bname
       (unit_status bname (car tmp1) 'location)
       (unit_status bname defender 'location)))
(cond ((equal (unit_status bname (car tmp1) 'type) 'helicopter)
       (setq terrain_mult 1.0))
      ((equal terrain_value 1) (setq terrain_mult 1.0))
      ((equal terrain_value 2) (setq terrain_mult .75))
      ((equal terrain_value 3) (setq terrain_mult .65))
      ((equal terrain_value 4) (setq terrain_mult .4)))
(setq actual_atck_strgth
      (+ (* (* (unit_status bname (car tmp1) 'attack_strength)
       (unit_status bname (car tmp1) 'proficiency))
    terrain_mult)
 actual_atck_strgth))
(and (setq tmp1 (cdr tmp1)) (go loop2))
; set actual_attack_strength to be total of all unit attack strength
(setq defend_pos_mult
      (cond
((equal (car (present_order defender)) 'move) .5)
((null
   (equal
     (unit_status bname defender 'location)
     (cadr (unit_status bname
defender
'previous_locations))))
 .75)
(t 1.0)))
(setq actual_dfnd_strgth
      (+ (* (* (unit_status bname defender 'defend_strength)
       (unit_status bname defender 'proficiency))
    defend_pos_mult)
 actual_dfnd_strgth))
;total defend strength now set
(setq attack_rslt
      (battle_outcome actual_atck_strgth actual_dfnd strgth))
(setq attack_rslt
      (list
(quotient (car attack_rslt) (length attackers))
(cadr attack_rslt)))
;effect on each attacking unit is now defined
(setq tmp1 attackers)
     loop4
(set_status bname (car tmp1)
    'proficiency
    (- (unit_status bname (car tmp1) 'proficiency)
       (car attack_rslt)))
```

```
(and (dead_unit bname (car tmpl)) (remove_from_game bname (car tmpl)))
(and (setq tmpl (cdr tmpl)) (go loop4))
;attackers status now reset
(set_status bname defender
    'proficiency
    (- (unit_status bname defender 'proficiency)
       (cadr attack_rslt)))
(and (dead_unit bname defender)
     (remove_from_game bname defender))
;defender status now reset
))
; can_attack?
; determines if attacker can legally attack defender
; a unit can only attack another unit that is within the attacking
; unit zone of control
;   bname is the name of the board
;   attacker is the name of the potential attacking unit
;   defender is the name of the potential defending unit
(defun can_attack? (bname attacker defender)
  (prog ()
(and (member (unit_status bname defender 'location)
     (zone_of_control bname attacker))
     (return t))
(return nil)))
; dead_unit
; determines if unit is no longer active
; when a units proficiency drops to 0 then it is
; removed from the board
(defun dead_unit (bname unitname)
  (< (unit_status bname unitname 'proficiency) .01))
; remove_from_game
; removes a unit from play of game
;   bname is the name of the board to remove unit from
;   unitname is the name of the unit to remove
(defun remove_from_game (bname unitname)
  (prog (tmpl)
(setq tmpl (unit_status bname unitname 'side))
(cond ((equal tmpl 'friend)
       (setq friendly_units
     (remove unitname friendly_units)))
      ((equal tmpl 'enemy)
       (setq enemy_units
     (remove unitname enemy_units))))
:above cond removes unit for list of active units
(set_status bname unitname 'is_active nil)
:a property of a unit is whether or not it is active
:this is set to nil
(set bname
     (boardset (eval bname)
       (unit_status bname unitname 'location)
       nil))
;the unit is removed from the board bname
)
  )
; battle_outcome
; determines the results of an attack in terms of proficiency loss
```

```
(defun battle_outcome (atck_strgth dfnd_strgth)
  (prog (ratio)
(setq ratio (quotient atck_strgth dfnd_strgth))
(cond ((> ratio 10) (return (list 0 10)))
      ((> ratio 9) (return (list 0 9)))
      ((> ratio 8) (return (list 0 8)))
      ((> ratio 7) (return (list 1 7)))
      ((> ratio 6) (return (list 1 6)))
      ((> ratio 5) (return (list 2 5)))
      ((> ratio 4) (return (list 2 4)))
      ((> ratio 3) (return (list 3 3)))
      ((> ratio 2) (return (list 4 2)))
      ((> ratio 1.5) (return (list 4 1.5)))
      ((> ratio 1) (return (list 4 1)))
      ((> ratio .5) (return (list 5 1)))
      ((> ratio .3) (return (list 7 1)))
      (t (return (list 9 0))))))
; THESE ROUTINES DEFINE TOP LEVEL GAME AND ACTIVITIES DURING A TURN
; display_orders
; will display all orders for given side
(defun display_orders (side)
  (prog (tmpl tmp2)
(cond ((equal side 'friend) (setq tmpl friendly_units))
      ((equal side 'enemy) (setq tmpl enemy_units))
      (t (return (print 'error_in_display_orders))))
(print (list side 'ORDERS 'ARE))
     loop1
        (print (car tmpl))
(princ "  ")
(setq  tmp2 (present_order (car tmpl)))
(princ tmp2)
(and (setq tmpl (cdr tmpl)) (go loop1))
))
; send_order
; will send an order to identified unit
;   unitname is the name of the unit
;   order is the new order for that unit
(defun send_order (unitname order)
  (prog (tmp1)
(cond
  ((equal (car (get 'side unitname)) 'friend)
   (setq tmpl 'friend_orders))
  ((equal (car (get 'side unitname)) 'enemy)
   (setq tmpl 'enemy_orders))
  (t (return (print 'error_in_send_order))))
(set tmpl
     (cons (list unitname order)
   (delete (list unitname
 (present_order unitname))
   (eval tmpl))))
(return
  (list 'present_order unitname
(present_order unitname)))
)
  )
; present_order
```

```
; returns the present order for specified unit
;   unitname is the name of the unit
;   friend_orders is global list of present orders for side friend
;   enemy_orders is global list of present orders for side enemy
(defun present_order (unitname)
  (prog (tmpl)
(and (null unitname)
     (return 'no_unit_in_present))
(cond
  ((equal (car (get 'side unitname)) 'friend)
   (setq tmpl friend_orders))
  ((equal (car (get 'side unitname)) 'enemy)
   (setq tmpl enemy_orders))
  (t (return (print 'error_in_present_order))))
;tmpl is set to list of orders
loopl
(and (equal (caar tmpl) unitname)
     (return (cadar tmpl)))
(and (setq tmpl (cdr tmpl)) (go loopl))
;loopl will return an order if one is found
(return '(no_order))
;if no order found the return no order
)
  )
; new_turn
; initializes the units for a new turn
; resets the movement_allowance
;         list of previous_locations
;         the order to no order
;         golobal list of supported_corridors to nil
;         tiime of board to time + 1
(defun new_turn (bname)
  (prog (tmpl)
(setq tmpl (union friendly_units enemy_units))
     loopl
(set_status bname (car tmpl) 'movement_allowance
    (unit_status 'initial (car tmpl) 'movement_allowance))
;movement allowance is reset to intial value usually 4
(set_status
  bname (car tmpl) 'previous_locations
  (cons (unit_status bname (car tmpl) 'location)
(unit_status bname (car tmpl)
     'previous_locations)))
;a list of locations at previous turns is saved
(send_order (car tmpl) '(no_order))
;unit has no order at beginning of turn this default could be removed
;leaving unit with standing orders
(and (setq tmpl (cdr tmpl)) (go loopl))
;loopl resets status for each unit
(setq supported_corridors nil)
;supported_corridors is global variable used by knowledge base
;its existence is a poor cluge
(putprop 'time (add1 (get 'time bname)) bname)
;update the time
(return
  (list 'NEW_TURN
```

```
'TIME_IS (get 'time bname)))))
; execute_orders
; executes all orders for both sides
;   bname is the name of the board to execute order ags.?.'
;   display_board is a flag which if t will display the ...
(defun execute_orders (bname display_board debug_mode'
  (execute_all_attacks bname)
  ; all attack goals on both sides are executed first
  (execute_moves bname nil)
  ; all moves on both sides are executed second
  ; units may not move and attack on the same turn
  (setq unit_actions (update_unit_actions))
  ; records what all the active units just did
  (and display_board (display (eval bname) nil))
  (and display_board
       (print (list 'supported_corridors_are supported_corridors)))
  (and display_board debug_mode
      (break))
  ;this break is used for showing the planner is action should be
  ;removed for actual planning
  (new_turn bname)
  ;after all orders executed update the board to new time
  )
; update_unit_actions
; returns list of each action of each unit
; this is used by knowledge base as a global variable
; very much a cluge
(defun update_unit_actions ()
  (prog (tmp1 tmp2)
(setq tmp1
      (union friendly_units enemy_units))
     loop1
(setq tmp2
      (cons (list (car tmp1)
  (car (present_order (car tmp1))))
  . tmp2))
(and (setq tmp1 (cdr tmp1)) (go loop1))
(return tmp2)
)
  )
;  DOMAIN SPECIFIC UTILITIES
;  some utilities that can be used by the knowledge base
; opposite_side
;returns opposing side of side specified
(defun opposite_side (side)
  (cond ((equal side 'friend) 'enemy)
((equal side 'enemy) 'friend)))
; get_path
; function to find a path from the present location of unitname
;           to the location in to
;   bname is the name of the board
;   unitname is the name of the unit
;   to is the location to move to
;   max_cost bounds the depth of the
;            search in terms of movement costs
;   stop_at_enemy is a flag which if nil will not account for fact
```

```
;                  that the enemy may be blocking ones path
(defun get_path (bname unitname to max_cost stop_at_enemy)
  ;if max_cost under 6 then may not get any path
  (prog (tmpl unit_locs)
(setq unit_locs (list_unit_locs bname (union friendly_units enemy_units)))
(setq tmpl previous_get_paths)
     loopl
(and (equal (list unit_locs unitname to max_cost stop_at_enemy)
    (car (first tmpl)))
     (return (cadr (first tmpl))))
(and (setq tmpl (cdr tmpl)) (go loopl))
;if path has been previously calculated then just retrieve it
;if this starts to use up to much memory then replace eval bname with somethin
; more limited or do io
(setq tmpl (find_path bname unitname
      (list (unit_status bname unitname 'location)) to 0
      '((1 1) (0 1) (1 0) (-1 0) (0 -1) (1 -1) (-1 1) (-1 -1))
      max_cost stop_at_enemy))
(setq tmpl (cdr (reverse (cadr tmpl))))
;path has now been found if within maxcost distance
(and (null tmpl)
     (setq tmpl (move_toward bname unitname to stop_at_enemy)))
;if no optimal path found then simply find a move_toward path
(and tmpl (setq previous_get_paths
(cons (list (list unit_locs unitname to max_cost stop_at_enemy) tmpl)
      previous_get_paths)))
; save path in list of previous get_paths
(return tmpl)
)
  )
: list_unit_locs
; unique board identier equal to list of all units and their location
(defun list_unit_locs (bname unitnames)
  (prog (tmpl)
    loopl
(setq tmpl (cons (list (car unitnames)
       (unit_status bname (car unitnames) 'location))
 tmpl))
(and (setq unitnames (cdr unitnames)) (go loopl))
(return (reverse tmpl))
)
  )
; move_toward
; trys to find a non_optimal path until within max_cost distance
; this is used by get_path if find path cannot find a complete
; path within maximum allowed movement cost
; this routine reflects the inelegance of the path finding algorithm
: presently being used
(defun move_toward (bname unitname to stop_at_enemy)
  (prog (tmpl tmp2 tmp3 tmp4 count)
(setq tmpl (unit_status bname unitname 'location) count 4)
     loopl
(setq tmp2 (-list to tmpl))
(cond ((and (= (car tmp2) 0) (= (cadr tmp2) 0))
       (setq tmp2 '(0 1)))
      ((and (> (car tmp2) 0) (= (cadr tmp2) 0))
```

```
       (setq tmp2 '(1 0)))
      ((and (= (car tmp2) 0) (< (cadr tmp2) 0))
       (setq tmp2 '(0 -1)))
      ((and (< (car tmp2) 0) (= (cadr tmp2) 0))
       (setq tmp2 '(-1 0)))
      ((and (> (car tmp2) 0) (> (cadr tmp2) 0))
       (setq tmp2 '(1 1)))
      ((and (> (car tmp2) 0) (< (cadr tmp2) 0))
       (setq tmp2  '(1 -1)))
      ((and (< (car tmp2) 0) (> (cadr tmp2) 0))
       (setq tmp2  '(-1 1)))
      ((and (< (car tmp2) 0) (< (cadr tmp2) 0))
       (setq tmp2 '(-1 -1))))
(and (enemy_in_way bname unitname tmp1 tmp2) (return tmp4))
(setq
  tmp3
  (cdr
    (reverse (cadr (find_path bname
      unitname
      (list tmp1)
      (+list tmp1 tmp2)
      0
      '((0 1) (1 0) (-1 0) (0 -1)
(1 1) (1 -1) (-1 1) (-1 -1))
      4 stop_at_enemy)))))
(and tmp3 (setq tmp4 (union tmp4 tmp3)))
(and (> (setq count (sub1 count)) 0)
     (setq tmp1 (+list tmp1 tmp2))
     (go loop1))
(return tmp4)))
; find_path
; will find a path for unitname to location to
; this is a recursive search routine that returns results
;      equivalent to exhaustive search
;   bname is the name of the board
;   unitname is the name of the unit to move
;   path_so_far is the path taken to this point
;               initially it is the list of the unit location
;   to is the location of the destination
;   cost_so_far is the total movement cost of the path_so_far
;   dirs_to_check is a list of allowable directions to move in
;   max_cost is tha maximum allowable movement cost of a path
;   stop_at_enemy is flag to determine is path search should account
;                 for enemy position
(defun find_path (bname unitname path_so_far
  to cost_so_far dirs_to_check
  max_cost stop_at_enemy)

  (prog (dirs best_so_far next_loc next_loc_cost tcost rslt)
(cond ((and (cdr path_so_far)
    (enemy_in_way bname unitname (cadr path_so_far)
  (-list (car path_so_far) (cadr path_so_far))))
    (return '(100 no_path))))
;check if enemy in way on last move
;this can happen depending on how find_path was first called
(and (equal (car path_so_far) to)
```

```
     (return (list cost_so_far path_so_far)))
;if destination found then a legal path has been found so it is returned
(setq tcost (+ max_cost .01) dirs dirs_to_check)
     loop1
(setq next_loc (+list (car dirs) (car path_so_far))
      next_loc_cost (cost_of_move bname unitname (car path_so_far) (car dirs)
  (unit_status bname unitname 'side) nil))
(and (> tcost (+ (+ cost_so_far next_loc_cost)
 (distance (+list (car path_so_far) (car dirs)) to)))
     (> 5 next_loc_cost)
     (null (member next_loc path_so_far))
     (cond (stop_at_enemy
    (or (equal (+list (car path_so_far) (car dirs)) to)
(null (enemy_in_way bname unitname (car path_so_far) (car dirs)))))
   (t t))
     (setq rslt
   (find_path bname unitname (cons next_loc path_so_far) to
      (+ next_loc_cost cost_so_far)
      dirs_to_check max_cost stop_at_enemy)))
;recursively calls find_path if next_loc may lead to good path
(and rslt (< (car rslt) tcost)
     (setq best_so_far rslt tcost (car rslt) rslt nil))
;if new path is cheaper, then use it as the standard
(and (setq dirs (cdr dirs)) (go loop1))
(cond ((null best_so_far) (return '(100 no_path)))
      (t (return best_so_far)))
; returns either no path of the best path so far
)
  )
; best_dir
; sets the direction to check in the correct general direction
; makes find_path more efficient
;   from is the starting unit location
;   to is the ending unit location
(defun best_dir (from to)
  (prog (tmp1 tmp2)
(setq tmp1 (-list to from) tmp2 '((0 1) (1 0) (1 1) (0 -1) (-1 0) (-1 -1) (1 -1) (-1
(cond ((and (= (car tmp1) 0) (> (cadr tmp1) 0))
       (setq tmp2 (union '((0 1) (1 1) (-1 1)) tmp2)))
      ((and (> (car tmp1) 0) (= (cadr tmp1) 0))
       (setq tmp2 (union '((1 0) (1 1) (1 -1)) tmp2)))
      ((and (= (car tmp1) 0) (< (cadr tmp1) 0))
       (setq tmp2 (union '((0 -1) (-1 -1) (1 -1)) tmp2)))
      ((and (< (car tmp1) 0) (= (cadr tmp1) 0))
       (setq tmp2 (union '((-1 0) (-1 -1) (-1 1)) tmp2)))
      ((and (> (car tmp1) 0) (> (cadr tmp1) 0))
       (setq tmp2 (union '((1 1) (1 0) (0 1)) tmp2)))
      ((and (> (car tmp1) 0) (< (cadr tmp1) 0))
       (setq tmp2 (union '((1 -1) (1 0) (0 -1)) tmp2)))
      ((and (< (car tmp1) 0) (> (cadr tmp1) 0))
       (setq tmp2 (union '((-1 1) (-1 0) (0 1)) tmp2)))
      ((and (< (car tmp1) 0) (< (cadr tmp1) 0))
       (setq tmp2 (union '((-1 -1) (-1 0) (0 -1)) tmp2)))
      )
(return tmp2)
)
```

```
  )
'(defun dest_dir (from to)
  '((1 1) (1 0) (0 1) (1 -1) (0 -1) (-1 1) (-1 0) (-1 -1)))
; enemy_in_way
; this function will determine if an enemy unit blocks movement in direction dir
; if a unit is in the way it will return this fact otherewise it returns nil
;   bname is the name of the board
;   unitname is the name of the unit
;   from should be the location moving from
;   dir is the direction of the proposed move
(defun enemy_in_way (bname unitname from dir)
  (prog (tmpl)

(setq tmpl
      (list
(boardval (eval bname) (+list from dir))
(and (equal (+ (abs (car dir)) (abs (cadr dir))) 2)
     (boardval (eval bname)
       (list (+ (car from) (car dir))
     (cadr from))))
(and (equal (+ (abs (car dir)) (abs (cadr dir))) 2)
     (boardval (eval bname)
       (list (car from)
     (+ (cadr from) (cadr dir)))))
))
loopl

(and (car tmpl)
     (null (equal
     (unit_status bname (car tmpl) 'side)
     (unit_status bname unitname 'side)))
     (return 'enemy_unit )._the_way))
(and (setq tmpl (cdr tmpl)). (go loopl))
(return nil)))
; distance
; euclidian distance measure
; will be less than or equal to actual travel distance between from and to
(defun distance (from to)
  (sqrt (+ (expt (car (-list from to)) 2)
   (expt (cadr (-list from to)) 2))))
; GENERAL UTILITIES
;   these utilies are not necessarily tied to this game
; +list
; for two lists of numbers of equal size
; returns list of the respective sums of those numbers
(defun +list (lst1 lst2)
  (prog (sumlst)
(or (equal (length lst1) (length lst2)) (return 'unequal_lists))
(or lst1 (return 'no_list))
loopl
(setq sumlst (cons (+ (car lst1) (car lst2)) sumlst))
(and (setq lst1 (cdr lst1) lst2 (cdr lst2)) (go loopl))
(return (reverse sumlst))
)
  )
; -list
```

```
; for two lists of numbers of equal size returns list of the
; respective subtraction of those numbers
(defun -list (lst1 lst2)
  (prog (diflst)
(or (equal (length lst1) (length lst2)) (return 'unequal_lists))
(or lst1 (return 'no_list))
loop1
(setq diflst (cons (- (car lst1) (car lst2)) diflst))
(and (setq lst1 (cdr lst1) lst2 (cdr lst2)) (go loop1))
(return (reverse diflst))
)
  )
; THESE FUNCTIONS ARE TO SAVE AND RETRIEVE FILES OF GAME SITUATIONS
; save_game
; saves the present game in the file filename
;    realboard is always the name of the present active board
; note that because previous_get_paths is a long list this could be an
; extensive file
(defun save_game (filename)
  (with-open-file (*standard-output* filename 'out)
    (prog (tmp1 tmp2)
(write realboard)
(write all_units)
(write friendly_units)
(write enemy_units)
(write list_of_unit_properties)
(write previous_get_paths)
(setq tmp1 all_units tmp2 list_of_unit_properties)
     loop1
(write (get (car tmp2) (car tmp1)))
(and (setq tmp2 (cdr tmp2)) (go loop1))
(and (setq tmp1 (cdr tmp1)) (setq tmp2 list_of_unit_properties) (go loop1))
)
    )
  )
; retrieve_game
; retrieves a game saved by save_game
; note that because previous_get_paths is a long list this could be an
; extensive file
(defun retrieve_game (filename)
  (with-open-file (*standard-input* filename 'in)
    (prog (tmp1 tmp2)
  (setq realboard (read))
  (setq all_units (read))
  (setq friendly_units (read))
  (setq enemy_units (read))
  (setq list_of_unit_properties (read))
  (setq previous_get_paths (read))
  (setq tmp1 all_units tmp2 list_of_unit_properties)
       loop1
  (putprop (car tmp2) (car tmp1) (read))
  (and (setq tmp2 (cdr tmp2)) (go loop1))
  (and (setq tmp1 (cdr tmp1)) (setq tmp2 list_of_unit_properties) (go loop1))
  )
    )
  )
```

```
; save_board
; this function will save the display of a board in a file
; this allows board to be printed later
;   bname is the name of the board
;   file is the name of the file to save to
(defun save_board (bname file)
  (with-open-file (*standard-output* file 'out)
    (prog (tmpflg board)
  (and (null (atom bname)) (return 'bad_arguement_to_save_board))
  (setq board (eval bname))
  (setq tmpflg t)
  (loop for j from 1 to 25 do
(princ "   ")
(loop for i from 1 to 27 do
      (displayl board (list i j))
      (cond (tmpflg (princ "  "))
    (t (princ " "))))
(and (null tmpflg) (princ (addl (quotient (subl j) 2))))
(terpri) (terpri)
(setq tmpflg (null tmpflg)))
  (princ "      ")
  (loop for i from 1 to 13 do
(princ i) (cond ((< i 10) (princ "     "))
(t (princ "    "))))
  )))
```

APPENDIX A

PROGRAM LISTINGS

Final Report

AI Planning II

F30602-85-C-0106
CLIN: 0002ABB

23 June 1988

Prepared by:

PAR Government Systems Corporation (PGSC)
1840 Michael Faraday Drive, Suite 300
Reston, Virginia 22090

PGSC Report 88-44